Towards Optimized Flexible Multi-ASIP Architectures for LDPC/Turbo Decoding

Purushotham Murugappa Velayuthan

N° d'ordre : 2012telb0260

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

en habilitation conjointe avec l'Université de Bretagne-Sud

Ecole Doctorale -sicma

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

par

Purushotham Murugappa Velayuthan

Towards Optimized Flexible Multi-ASIP Architectures for LDPC/Turbo Decoding

soutenue le 17 décembre 2012 devant la commission d'examen :

Composition du Jury :

Président :	M. Guy Gogniat,	Professeur à l'Université de Bretagne-Sud
Rapporteurs :	M. Gerd Ascheid,	Professeur à l'Université RWTH Aachen
	M. François Verdier,	Professeur à l'Université de Nice
Examinateurs :	M. Gilles Sassatelli,	Directeur de Recherche CNRS au LIRMM
	M. Fabien Clermidy,	Ingénieur chercheur (HDR) au CEA-LETI
Directeur :	M. Michel Jézéquel,	Professeur à Télécom Bretagne
Encadrant :	M. Amer Baghdadi,	Professeur à Télécom Bretagne

To life and its infinite possibilities.

Acknowledgements

First, I would like to express my gratitude to my supervisor Amer Baghdadi, for his constant motivation and patience during this entire thesis work. I also thank him for both personal and technical help that he graciously offered irrespective of the time of the day it was needed. I always will cherish the freedom he offered (as well as the pressure he exerted at times) in bringing out the best in me. I also thank him for the kindness that he showed during those endless arguments, discussions and obvious misunderstandings that we have had.

I would also like to thank Michel Jézéquel, the director of my thesis for his support during this period.

I extend my gratitude to my collegues Atif, Rachid, Camilo, Jean-Noel and many others to have supported me and made me feel welcomed every day at work. A special thanks also goes to my dear friends Ashwani and Tarini (and many more from/in various parts of the world) to have tirelessly counselled me to uplift my spirits whenever I felt down.

Lastly, it's a pleasure to thank my family in India for having stood by me through thick and thin. They have always encouraged me to pursue my dreams no matter how weird it seemed.

Contents

List of Figures

List of Tables

Introduction

MOBILE wireless connectivity is a key feature of a growing number of devices, which will count soon in tens of billions, from laptops, tablets, cell phones, cameras and other portable devices. The variety of applications and traffic types will be significantly larger than today and will result in more diverse requirements. These applications are driving the creation of new transmission techniques and design architectures that push the boundaries to achieve high throughput, low latency, area and power efficient implementations.

Channel coding is one of the key techniques that enable reliable high throughput data transfer through unreliable wireless channels. However, as a large variety of channel coding options and flavors are specified in existing and emerging digital communication standards, there is an increasing need for flexible implementations. In fact, several powerful error correction techniques exist today, each suitable for specific application parameters (frame size, transmission channel, signal-to-noise ratio, bandwidth, etc). Considering the emerging multi-mode and multi-standard applications, as well as the increasing interest for Software Defined Radio (SDR) and Cognitive Radio (CR) applications, combination of multiple error correction techniques becomes mandatory. Table 1 shows a representative set of mobile wireless standards to highlight their differences in data rates and channel encoding schemes. The most commonly used error correcting codes in these standards are Convolutional Codes (CC), Turbo codes (SBTC: Single Binary Turbo Codes and DBTC: Double Binary Turbo Codes), and Low-Density Parity-Check (LDPC) codes.

Standard	Codes	Rates	States	info. bits	Channel Throughput
EDGE	CC	1/4..1/3	64	..870	384 kbps
UMTS	CC	1/4..1/2	256	.. 504	.. 32 kbps
	SBTC	1/3	8	.. 5114	.. 2 Mbps
HSDPA	SBTC	1/2 - 3/4	8	.. 5114	.. 14.4 Mbps
CDMA-2k	CC	1/6 .. 1/2	256	..744	.. 28 kbps
	SBTC	1/5 - 1/2	8	..20730	.. 2 Mbps
IEEE-802.11n (WiFi)	CC	1/2..3/4	64	.. 4095	.. 450Mbps
	LDPC	1/2 - 5/6	-	.. 1620	.. 450Mbps
IEEE802.16e (WiMAX)	CC	1/2 - 5/6	64	.. 864	.. 75 Mbps
	DBTC	1/2 - 3/4	8	.. 4800	.. 75 Mbps
	LDPC	1/2 - 5/6	-	.. 1920	.. 75 Mbps
DVB-S2	LDPC	1/4 - 9/10	-	.. 64800	.. 90 Mbps
DVB-RCS	DBTC	1/3 - 6/7	8	.. 1728	.. 2 Mbps
3GPP-LTE	SBTC	0.33-0.95	8	.. 6144	.. 150 Mbps

Table 1 — Representative set of mobile wireless standards and related channel codes and parameters

Problems

In this context, and at the receiver side, it is well known that channel decoding is one of the most computation, communication, and memory intensive, and thus, power-consuming component. Channel decoder design has 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 through 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 time which is no more compatible with the severe time-to-market constraints and the continuous development of new standards and applications.

More recently, several contributions 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 proposed to increase the target flexibility to the support of different channel coding techniques. As a matter of fact, a knowledge gap is growing quickly in the last few years between the need for flexibility in the digital base-band processing segment of modern communication systems, and the actual availability of flexible while efficient hardware support to the quest for reconfigurability. The main reason that determines this growing gap is related to the poor area and energy efficiency of flexible solutions proposed till now and the huge increase of non-recurrent engineering (NRE) costs in the production of dedicated integrated circuits for specific applications (ASIC) with new semiconductor technologies.

Objectives and scope of the thesis

Towards the target of filling the above mentioned gap, this thesis work aims at defining and developing an efficient and high performance flexible channel decoder architecture model for emerging and future digital communication systems. The need of optimal solutions in terms of performance, area, and power consumption is increasing and cannot be neglected against flexibility. In common understanding, a ”blind” approach towards flexibility results in some loss in optimality. The objective of this work is related to unifying flexibility-oriented and optimization-oriented approaches. The main goal is to deliver enablers and building block solutions in order to derive, for a specific application need, the best balance between a highly flexible solution and a specifically optimized one.

Towards this objective, the thesis work investigates multiprocessing and Application-Specific Instruction-set Processor models (ASIP) which enable the designer to scale and freely tune the flexibility/performance trade-off as required by the considered application requirements. Related contributions are emerging rapidly seeking to improve the resulting architecture efficiency in terms of performance/area and in addition to increase the flexibility support.

By considering mainly the challenging Turbo and LDPC decoding applications, multi-ASIP channel decoder architectures are proposed targeting high flexibility combined with high Architecture Efficiency (AE) in terms of bits/cycle/iteration/mm^2. Different architecture alternatives and design approaches are explored. Furthermore, in order to be relevant to existing and emerging standards we limit the supported flexibility targeting LDPC and Turbo codes specified in

WiFi, WiMAX, and LTE. This also enables to compare with existing state-of-the-art implementations.

Thesis contributions

Towards the above mentioned objectives, the results of this thesis work can be summarized in the following 4 main contributions (which are also illustrated in Figure 1 corresponding to chapters 3, 4, 5, and 6):

- *Design of a scalable and flexible high throughput multi-ASIP LDPC/Turbo decoder*:
    - Proposal and design of a flexible and optimized ASIP architecture, namely **DecASIP**, supporting Turbo and LDPC decoding.
    - Proposal of an efficient resource sharing between LDPC and Turbo decoding modes.
    - Proposal of a scalable high throughput multi-ASIP channel decoder with efficient mapping of the target standards.
    - Proposal of a new LDPC decoding schedule adapted to the target multi-ASIP channel decoder.
    - Exploring possible parallelism techniques for efficient decoding of SBTC, DBTC, and LDPC codes.
    - Optimization of the dynamic configuration speed between the different supported decoding modes.
- *FPGA and ASIC prototyping of the proposed multi-ASIP LDPC/Turbo decoder*:
    - Proposal and design of a complete FPGA-based prototype of the proposed multi-ASIP LDPC/Turbo decoder.
    - ASIC integration of a 4-DecASIP channel decoder in the latest Telecom chip (namely MAG3D) designed by the CEA-LETI targeting 4G communication applications
- *Design of a parameterized ASIP for Turbo decoding, **TDecASIP***:
    - Increasing the architecture efficiency in terms of bit/cycle/iteration/mm^2.
    - Enabling the design of application-specific parameterized cores using an ASIP design flow.
- *Design of an optimized ASIP for LDPC decoding, namely **LDecASIP*** :
    - Increasing the architecture efficiency in terms of bit/cycle/iteration/mm^2.
    - Enhancing the ASIP-based LDPC decoder with a design-time feature enabling incremental changes for future support of other QC-LDPC codes (e.g. DVB-S2 with high expansion factor Z = 360).

Thesis Outline

The thesis manuscript is composed of six chapters as illustrated in Figure 1 and described below:

Chapter 1 introduces the basic concepts related to convolutional Turbo codes and LDPC codes along with their decoding algorithms. First, an overview of the fundamental concepts of channel coding and the basics for error-correcting codes are introduced. The second section presents the Turbo codes and details their basic components. This is followed, in the third section, by the presentation of the corresponding decoding algorithms, namely Maximum Aposteriori Probability (MAP) and the low complexity Max-Log-MAP. This section also briefs a note

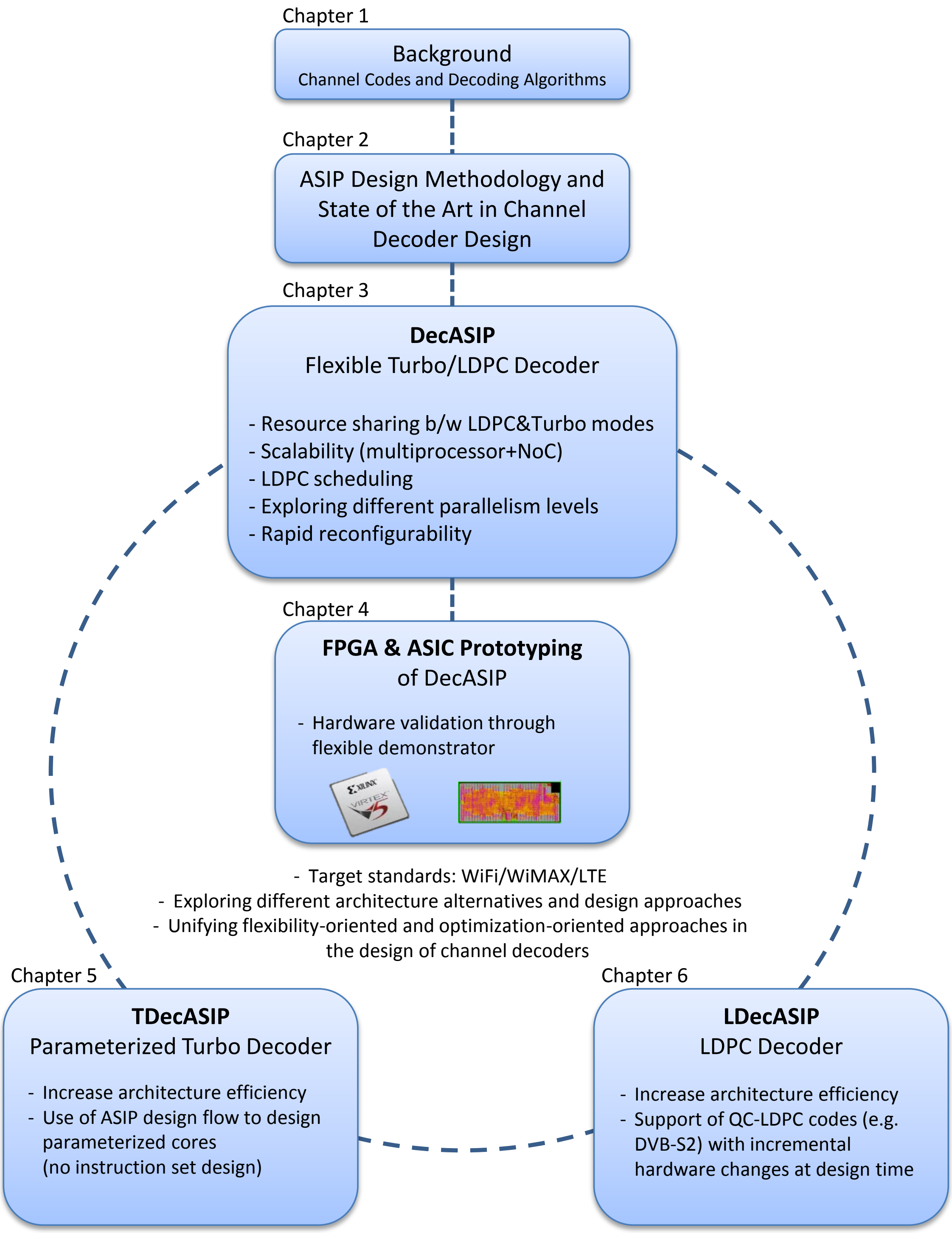

Chapter 1
Background
Channel Codes and Decoding Algorithms
Chapter 2
ASIP Design Methodology and State of the Art in Channel Decoder Design
Chapter 3
DecASIP
Flexible Turbo/LDPC Decoder
- Resource sharing b/w LDPC&Turbo modes
- Scalability (multiprocessor+NoC)
- LDPC scheduling
- Exploring different parallelism levels
- Rapid reconfigurability
Chapter 4
FPGA & ASIC Prototyping of DecASIP
- Hardware validation through flexible demonstrator
- Target standards: WiFi/WiMAX/LTE
- Exploring different architecture alternatives and design approaches
- Unifying flexibility-oriented and optimization-oriented approaches in the design of channel decoders
Chapter 5
TDecASIP
Parameterized Turbo Decoder
- Increase architecture efficiency
- Use of ASIP design flow to design parameterized cores (no instruction set design)
Chapter 6
LDecASIP
LDPC Decoder
- Increase architecture efficiency
- Support of QC-LDPC codes (e.g. DVB-S2) with incremental hardware changes at design time

Figure 1 **—** Overview of the thesis outline and contributions

on possible parallelism levels exploited for implementation. The last two sections focus on the presentation of the LDPC codes and their most commonly used algorithm, namely Normalized Min-Sum (NMS), in a reformulated manner as used in this thesis work.

Chapter 2 introduces the ASIP-based design approach and the considered Processor Designer tool from Synopsys (ex. CoWare tools). It further gives an overview on state-of-the-art efforts in channel decoder design in order to clarify the position of the proposed contributions in this thesis work. Based on the ASIP design approach, the chapter presents an initial ASIP architecture, namely TurbASIP, which constitutes the starting point of this thesis work.

Chapter 3 presents our first contributions in the design of flexible and optimized channel decoder supporting Turbo and LDPC codes. Starting with the initial TurbASIP architecture presented in the previous chapter, several objectives have been specified for this work, which can be summarized in the following points: (1) efficient resource sharing between the LDPC and Turbo decoding modes, (2) scalability to enable the accommodation of current and future high throughput requirements, (3) new LDPC decoding schedule adapted to the base TurbASIP architecture, (4) exploring possible parallelism techniques for efficient decoding of SBTC, DBTC, and LDPC codes, and (5) rapid configurability between the different supported decoding modes.

Towards fulfilling these objectives, an ASIP-based multiprocessor architecture is proposed and designed in two phases. The first phase achieves the targeted objectives through the design of a novel ASIP architecture, namely DecASIP$_{v1}$, and the efficient mapping of the target standards on an 8-DecASIP system decoder. The second phase (DecASIP$_{v2}$) mainly enhances the throughput in LDPC mode by increasing the supported parallelism degree. It also modifies the proposed LDPC scheduling to support 4-DecASIP or 2-DecASIP decoder architectures.

The first section of this chapter presents the design motivations and architectural choices made for the *DecASIP*$_{v1}$ along with the analysis on quantization and reference curves for the implemented modes, namely SBTC (LTE), DBTC (WiMAX) and LDPC (WiFi, WiMAX) modes. The second and third sections present the two design phases of the proposed DecASIP channel decoder.

Chapter 4 is dedicated to the presentation of the conducted efforts towards FPGA and ASIC prototyping of the proposed flexible channel decoder. A complete FPGA-based prototype of the proposed multi-standard Turbo/LDPC decoder is demonstrated. The functional prototype implements a full communication system including encoder, channel model, ASIP-based decoder and performance counters. All components are flexible and are dynamically configurable through a dedicated GUI (Graphical User Interface). The proposed prototype supports all communication modes defined in LTE, WiFi and WiMAX wireless communication standards. Furthermore, as a joint effort with another PhD student at the CEA-LETI (Pallavi Reddy), an ASIC integration of the proposed flexible channel decoder has been elaborated. A 4-DecASIP channel decoder is integrated in the latest Telecom chip (namely MAG3D) designed by the CEA-LETI targeting 4G communication applications. The chapter is organised as follows. The first section reiterates the communication system model presented in the first chapter drawing parallels to the implemented design units on the target FPGA prototype. Two blocks are illustrated in detail in the second section of this chapter: the flexible Turbo and LDPC encoders.

Chapter 5 presents a new optimized flexible standalone Turbo decoder, namely TDecASIP. The objective behind this new design is twofold: (1) investigate the maximum attainable architecture efficiency for ASIP-based Turbo decoding, and related to this first objective (2) investigate the possibility to design application-specific parameterized cores using the available ASIP design flow. The idea of this last objective is to evaluate the benefits from removing the need of a program memory and the related instruction decoder. Towards fulfilling these objectives, TDecASIP architecture is proposed and designed as a parameterized Turbo decoder. The proposed architecture exhibits a very high architecture efficiency and supports all SBTC and DBTC modes of 3GPP LTE and WiMAX standards respectively.

This chapter is organized as follows. The first section presents the proposed design flow using Synopsys Processor Designer tool to describe application-specific parameterized cores.

The second section illustrates the motivations behind the architectural choices and describes the proposed architecture of TDecASIP. Finally, the last two sections of the chapter presents the FPGA and ASIC synthesis results highlighting the attained architecture efficiency of this new flexible Turbo decoder design.

Chapter 6 presents a new optimized flexible standalone LDPC decoder, namely LDecASIP. The objective behind this new design is twofold: (1) investigate the maximum attainable architecture efficiency for ASIP-based LDPC decoding and (2) explore the possibility of realizing a flexible decoder allowing to implement and validate new/incremental algorithm changes with fast turnaround time in design. The idea of this last objective is to enhance the ASIP-based LDPC decoder with a design-time feature enabling incremental changes for future support of other QC-LDPC codes (e.g. DVB-S2 with high expansion factor $Z = 360$). Towards fulfilling these objectives, LDecASIP architecture is proposed and designed. The proposed architecture exhibits a very high architecture efficiency, supports all QC-LDPC codes and related parameters of WiFi and WiMAX standards (with expansion factors ranging from $Z = 24$ to $Z = 96$), and enables the support other QC-LDPC codes with structured incremental hardware changes at design time.

The chapter is organized as follows. The first section describes the design motivations along with the proposed LDPC decoder architecture. The second section presents the added design-time flexibility feature and illustrates the proposed way of upgrading the design to support other QC-LDPC codes through the example of DVB-S2 LDPC decoding. Finally, the third section presents the FPGA and ASIC synthesis results with architecture efficiency evaluation and related discussions.

CHAPTER

1 Background: Channel Codes and Decoding Algorithms

THIS chapter presents an overview of a typical communication system with special emphasis on channel coding and decoding algorithms. As this thesis work targets LDPC and Turbo codes, the chapter focuses mainly on detailing the decoding algorithms of these codes.

The first section introduces the need for channel coding and the notion of channel capacity. The second section presents the Turbo codes and details their basic components. This is followed, in the third section, by the presentation of the corresponding decoding algorithms, namely Maximum Aposteriori Probability (MAP) and the low complexity Max-Log-MAP. This section also briefs a note on possible parallelism levels that should be exploited for high throughput implementations. The last two sections focus on the presentation of the LDPC codes and their most commonly used decoding algorithm, namely Normalized Min-Sum (NMS), in a reformulated manner as used in this thesis work.

1.1 Communication system overview

A simplified block digram of a communication system which emphasis the role of the error-correction (control) coding (ECC) is shown in Figure 1.1. It consists of a *source*, a channel encoder, a transmission channel, a channel decoder and a *sink*. The *source* abstracts the information data to be transmitted, e.g. audio, video, text data, etc., while the *sink* block abstracts the actual recipient of this data, e.g. television, radio, cell phone, etc. The *source* output, if analog, has to be converted to digital form by a message encoder which may additionally remove redundancy in the information bits (source coding) before sending them to the channel encoder. The channel encoder introduces redundancy in a controlled manner so that at the receiving end, the redundant bits can be used for error detection or error correction. The modulator maps the encoded digital sequences into signal waveforms suitable for propagation. Modulation can be performed by varying the amplitude, the phase or the frequency of a sinusoidal waveform called the *carrier*. The channel block abstracts the communication medium over which the data is transmitted, e.g. air, wire-line, fiber optic channels, etc. Two major limitations of real channels are thermal noise and finite bandwidth. Hence in the receiver, the demodulator typically generates a digital sequence at its output as the best estimates of the transmitted codeword. In other words, the demodulator demaps the modulated signal received from the channel into soft values that represent the received signal. From these values, the channel decoder estimates the transmitted message based on the encoding rule and the characteristics of the channel with a goal to minimize the effect of the channel noise.

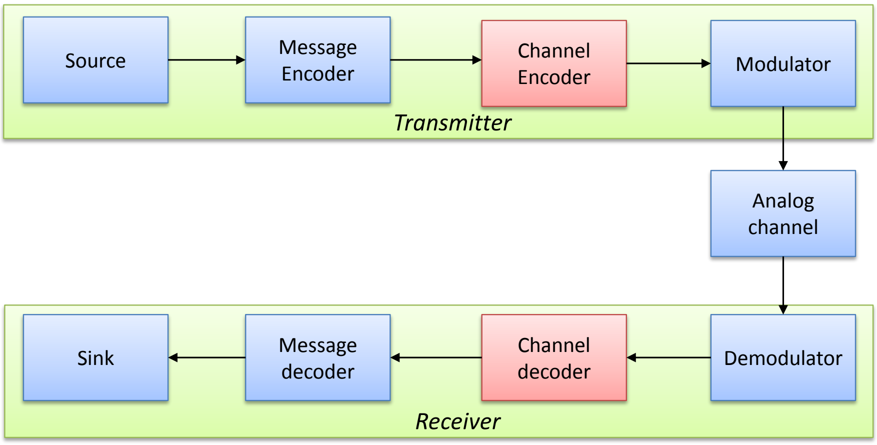


Figure 1.1 — Digital communication system model

The Shannon limit: The aim of every digital communication system is to transmit as much data as possible with little or no error utilizing minimum amount of power. Signal bandwidth is the measure of the speed of transmission. High speed transmission waveforms would need rapid changes in time and hence high transmission bandwidth. However, the capacitive and inductive properties of the channel prevents instantaneous change of signals, depending on the frequency range, environment and medium of the channel. Various probabilistic models have been proposed for simplicity of mathematical analysis like, Additive White Gaussian Noise (AWGN) channel, Rayleigh channel, etc. In this thesis, we consider the simple AWGN channel model. As the name suggests, the added noise to the transmitted signal has a Gaussian distribution.

For a given channel bandwidth B, there is an upper limit on the data rate related to the Energy per bit to noise ratio $\frac{E_b}{N0}$ of the channel [1]. This maximum limit is called *Channel capacity* and for an AWGN channel is given by (as shown by Claude Shannon [1]):

$$C = B \times log_2(1 + \frac{E_b}{N0})\ bits/sec \tag{1.1}$$

Shannon's channel coding theorem guarantees that data can be transmitted reliably at very low probability of error over a noisy communication channel if the transmission rate R is less than the threshold C. The probability of errors can be made to decrease exponentially as the frame length of the coding scheme goes to infinity. The gain or power saved w.r.t. to a uncoded system achieved due to the use of channel coding is called *Coding gain*.

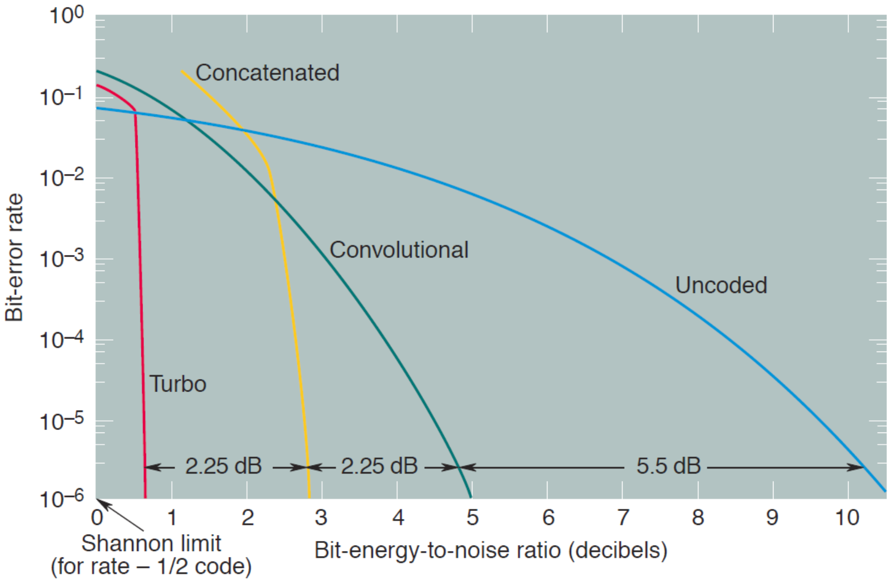


Figure 1.2 — Bit error rate performance of a Turbo code w.r.t. channel capacity and other conventional codes

This theorem only guarantees the existence of such codes but does not define these codes. Thus, ever since its first publication in 1948, the scientific community is trying to find error correction codes of finite length with reasonable complexity and approaching as close as possible to the channel capacity. In this context, different channel codes have been proposed aiming to achieve this channel capacity such as: convolutional codes, Hamming codes, Reed Solomon codes, convolutional Turbo codes, Block Turbo codes, LDPC codes, etc. Among these codes, Turbo and LDPC codes are shown to be capacity approaching. Figure 1.2 [2] illustrates how for a target bit-error rate of 10^{-6}, the use of convolutional and concatenated codes provides a 5.5-decibel and 7.75-decibel improvement respectively compared with the uncoded system. The use of a Turbo code imparts an additional 2.25-decibel improvement compared with the concatenated code, resulting in a total coding gain of 10 decibels compared with the uncoded system. With the use of a rate-1/2 Turbo code, system error performance can approach levels within about 1 decibel of the Shannon limit. Similar performance curves can be obtained for LDPC codes too. Due to such excellent error correction properties, LDPC and Turbo codes are widely used in recent and emerging wireless communication standards. The following sections introduce these two families of codes with their commonly used decoding algorithms.

1.2 Turbo codes

As mentioned earlier, the probability of errors can be made to decrease exponentially as the coded frame length (N) goes to infinity. However, for such long codes the optimum decoding scheme that simply computes the likelihood of every possible transmitted codeword presents a complexity which increases exponentially with N, hence practical implementation of such decoders becomes infeasible. As an alternative, David Forney proposed a divide and conquer approach to the problem by designing codes that are concatenation of simpler codes [3]. Two possible methods of concatenation exist, namely:

- Serial concatenation (Figure 1.3) where the output of outer code is the input of the inner code. Subsequently, it was observed that the addition of a function of interleaving between the two codes increases robustness of the concatenated codes significantly. Figure 1.3 shows an example of such serially concatenated codes with a global code rate of k/n with two component encoders (Code1, Code2). The encoders Code1 and Code2 encode the data at rates of k/p and p/n respectively.

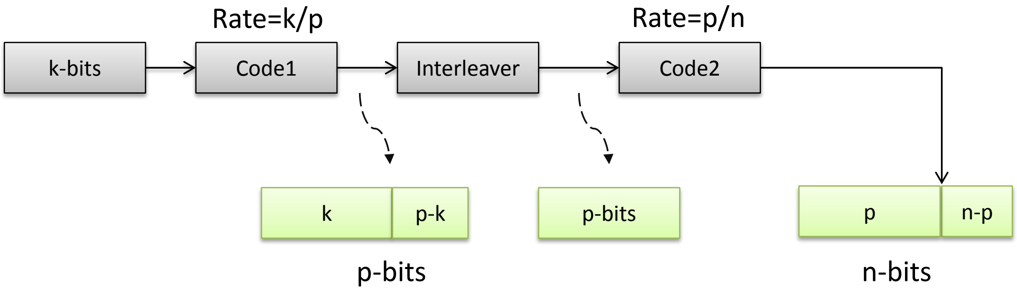


Figure 1.3 — Serial concatenated code structure

- Parallel concatenation (Figure 1.4) which consists of two systematic encoders, where the first encoder receives the source data X_i in natural order and at the same time the second encoder receives the same data but in interleaved order. The output is composed of the source data and the associated redundant bits produced by the natural and the interleaved domain encoders.

In fact, many existing and emerging wireless communication standards specify parallel concatenated convolutional Turbo codes. In this context, a typical Turbo encoder consists of parallel concatenation of two Recursive Systematic Convolutional (RSC) encoders separated by an interleaver as shown in Figure 1.4. The input bit stream X_i is encoded into 3 output streams namely the systematic S as well as two parities P_i and P_i' from the encoders that encode the natural and interleaved bit streams respectively. Generally, two types of RSC encoders are specified in

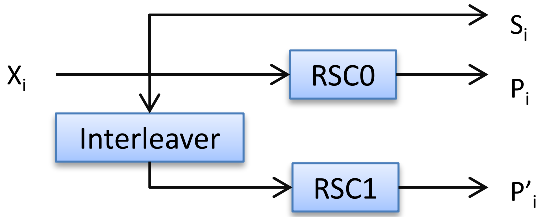


Figure 1.4 — Parallel RSC Turbo encoder structure

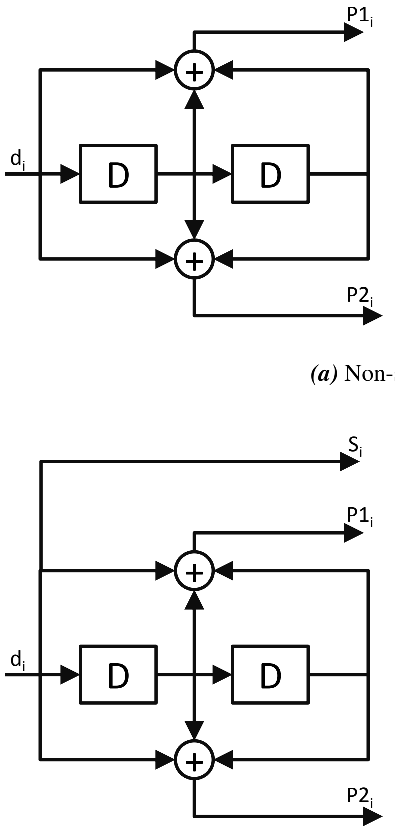
current standards:

- Double Binary Turbo Code (DBTC) encoder (specified in DVB-RCS and WiMAX standards) uses a RSC that encodes bit pairs (double binary symbols) of the incoming payload of size N bits to a stream of parity bit pairs.
- Single Binary Turbo Code (SBTC) encoder (specified in LTE standard) which uses a RSC that encodes bitwise the incoming payload of size N bits to a stream of parity bits.

The rest of this section presents the concepts of convolutional codes and interleaving rules as specified in the target wireless communication standards.

1.2.1 Recursive Systematic Convolutional codes

Convolutional codes have been widely used in applications such as space and satellite communications, cellular mobile, digital broadcasting, etc. Their popularity is due to their simple structure and easily implementable maximum likelihood soft decision decoding methods.
A convolutional code of rate $r = k/n$ is a linear function which, at every instant i, transforms an input symbol d_i of k bits into an output coded symbol c_i of n bits ($k > n$). 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 $(n - k)$ are made up of parity bits. The output bits are the linear combination of the current and the previous input bits. The linear function is usually given by generator polynomials. There



(a) Non-systematic convolutional code

(b) Systematic convolutional code

Figure 1.5 — Examples of Non systematic and Systematic convolutional codes

are three main types of convolutional codes namely Non-Systematic (NSC), Systematic (SC) and Recursive Systematic (RSC) convolutional codes. The only difference in the encoding of the SC and NSC is that, in NSC the output does not contain input systematic bits as shown in Figure 1.5a. This figure illustrates too how a convolutional encoder can be described using shift registers and modulo two adders. The generator polynomials of the convolutional codes examples of Figure 1.5 are given as:

$$\begin{aligned} S_i &= d_i \\ P1_i &= g1(x) = 1 + x^1 + x^2 \\ P2_i &= g2(x) = 1 + x^2 \\ C_i &\equiv (S_i, P1_i, P2_i) \end{aligned}$$

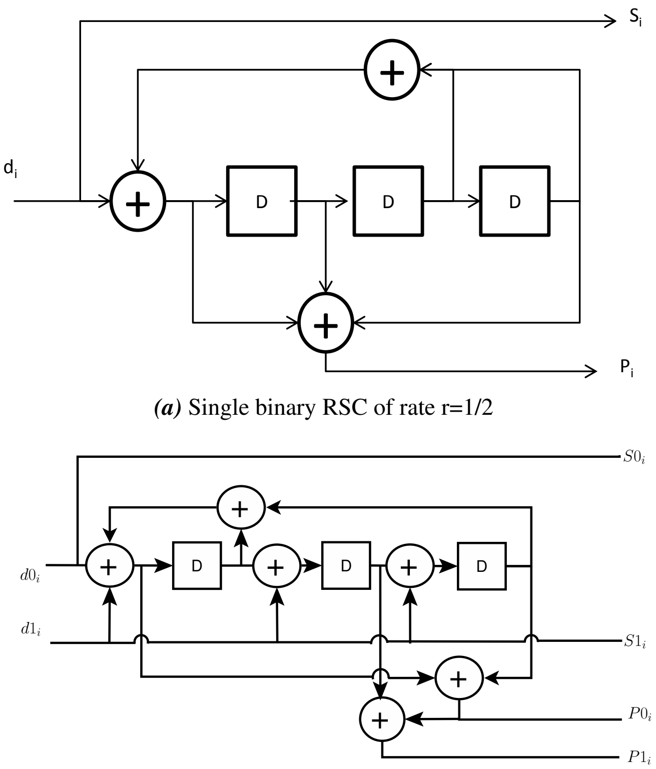
These generator polynomials represent the connections between the outputs of the shift register and the modulo two adders. They are generally represented by their coefficients in binary: (111) for $P1_i$ and (101) for $P2_i$. If the encoder shift register has feedback and includes input data as part of the output then it is called Recursive systematic encoder (RSC) (Figure 1.6a). The encoder shown Figure 1.6a is a single binary Recursive Systematic Convolutional (RSC) code (from LTE standard [4]) which encodes one bit at a time. The parity bits are generated by the polynomial given by:

$$\begin{aligned} P(D) &= \frac{1 + D + D^3}{1 + D^2 + D^3} \\ C_i &\equiv (S_i, P_i) \end{aligned}$$

Similar representation is used for a RSC which encodes 2 bits at a time as shown in Figure 1.6b. The encoder shown is called Double Binary Recursive Systematic Convolutional code (as used in WiMAX standard [5]) and the parity bits are generated by the polynomial given by:

$$\begin{aligned} S0_i &= d0_i \\ S1_i &= d1_i \\ P_1 &= \frac{1 + D^2 + D^3}{1 + D + D^3} \\ P_2 &= \frac{1 + D^3}{1 + D + D^3} \\ C_i &\equiv (S0_i, S1_i, P0_i, P1_i) \end{aligned}$$

Convolutional codes are characterized by a parameter called constraint length, given as K = M+1, where M is number of flip-flops of the encoder shift register. The number of flip-flops also indicated that the encoder has 2^M states. Another commonly used representation of convolutional encoding is the trellis diagram [6] which is made up of nodes and branches. A node represents the state S of the code. A branch represents a transition from one state $(S_{i-1}(m'))$ to another state $(S_i(m))$ due to an input bit pair (in case of a double binary convolutional code). Each transition is associated to input and output vector of the encoder. Figure 1.7 shows the trellis representation of the double binary encoder of Figure 1.6b that has K=4, i.e. with M=3 (8 states). It has to be noted that each state has 2^b=4 possible transitions, where $b = 2$ is the number of bits at the input of the encoder.



(a) Single binary RSC of rate r=1/2

(b) Double binary RSC of rate r=1/2

Figure 1.6 — Recursive systematic codes (RSC)

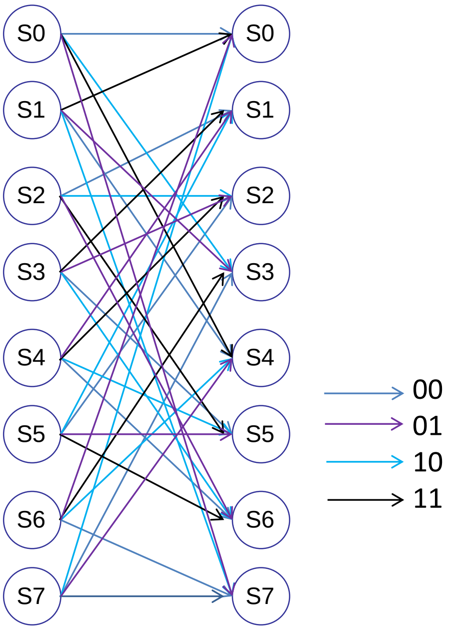


Figure 1.7 — Trellis diagram of encoder of Figure 1.6b

Trellis termination: During the encoding process the shift register of the encoder starts typically with the state zero. Towards the end of the encoding process M zeros are inserted in order to bring back the encoder state to zero. Thus, the transmitted codeword has extra parity bits that are the result of the zero insertion, known in the literature as *zero padding*. Such technique ensures that the encoder starts and ends in a known state, however it results in minor loss of transmission bandwidth.

As an alternative, *Tail biting* scheme is used in some standards like DVB-RCS and WiMAX which adopt circular recursive systematic convolutional (CRSC) codes. For this type of circular codes, the encoder is initialized for each frame of data in a certain state called the circulation state S_c that leads the encoder to return to the same state at the end of the encoding process of the frame. The existence of such a state is ensured when the size of the encoded data frame is not a multiple of the period of the encoding recursive generator [7]. The value of the circulation state S_c depends on the contents of the sequence to encode and determining S_c requires a pre-encoding operation. First, the encoder is initialized to the ”all zero” state and the data sequence is encoded once, leading to a final encoder state S_k^0. From this final state S_k^0, the circulation state S_c can be calculated using simple combinational operators or a corresponding lookup table as described in [7].

Viterbi algorithm is commonly used to decode convolutional codes [8]. Viterbi decoding is done via estimating the most likely sequence of states by observing the received bit sequence.

1.2.2 Turbo Code Interleaver

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:

$$S = \min_{i,j} \left(|i - j| + |\Pi(i) - \Pi(j)| \right) \tag{1.2}$$

The design of interleavers respecting a dispersion factor can be reasonably achieved through the S-random algorithm proposed in [9]. 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 WiMAX and LTE standards are described.

1.2.2.1 Almost regular permutation (ARP)

The ARP interleaver is used in double binary Turbo codes for both standards IEEE 802.16e WiMAX . 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.

$$\prod(j) = (P_0 \times j + P + 1) \; mod \; N \tag{1.3}$$

where

$$\begin{aligned} &P = 0 && \text{if } j \; mod \, 4 = 0 \\ &P = \frac{N}{2} + P_1 && \text{if } j \; mod \, 4 = 1 \\ &P = P_2 && \text{if } j \; mod \, 4 = 2 \\ &P = \frac{N}{2} + P_3 && \text{if } j \; mod \, 4 = 3 \end{aligned} \tag{1.4}$$

where the parameters P_0, P_1, P_2 and P_3 depend on the frame size and are specified in the corresponding standard [5].

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.

1.2.2.2 Quadratic polynomial permutation (QPP)

The interleaver specified in single binary Turbo codes for the LTE standard [4] is called quadratic polynomial permutation (QPP) interleaver. It is given by the following expression:

$$\prod(j) = (f_1 j + f_2 j^2) \; mod \, N \tag{1.5}$$

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 standard. In this standard, all the frame sizes are even numbers and are divisible by 4 and 8. Moreover, by definition, the 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 1.6 and 1.7:

$$\prod(2 \times k) \; mod \, 2 = 0 \tag{1.6}$$

$$\prod(2 \times k + 1) \; mod \, 2 = 1 \tag{1.7}$$

This property will be used later for the hardware implementation in order to design an extrinsic exchange module to avoid memory collisions when using particular parallelism techniques (butterfly schedule and trellis compression). More information on the other properties for the QPP interleavers are given in [10, 11]. Moreover, the frame size N is always divisible by 16, 32, and 64 when N$\geq$ 512, N$\geq$1024, and N$\geq$2048, respectively. The specific structure of the QPP interleaver equation enables sub-block parallelism degrees (see Sub-section 1.3.4.2) that range from $1, 2, ...$ to 64.

1.3 Turbo decoding

This section presents the principle of Turbo decoding with a brief overview of the Maximum Aposteriori Probability (MAP) algorithm. The Turbo decoding principle relies on the iterative exchange of probabilistic messages between two (or more) decoders dealing with the same received data [12]. In a typical Turbo decoding system (Figure 1.8), two decoders operate iteratively on the received frame (one in natural domain and the other in interleaved domain) and pass probabilistic messages, the so-called *extrinsic information*, to each other after each iteration. The decoders, namely Soft-Input Soft- Output (SISO) decoders, operate on soft information to improve the decoding performance. Thus, each SISO decoder takes into consideration, besides its own channel input data, the received extrinsic information from the other SISO decoder in order to improve its estimation over the iterations. Usually, but not necessarily, the computations are done in the logarithmic domain. Each decoder calculates the Log-Likelihood Ratio (LLR) for the ith data bit d_i, as

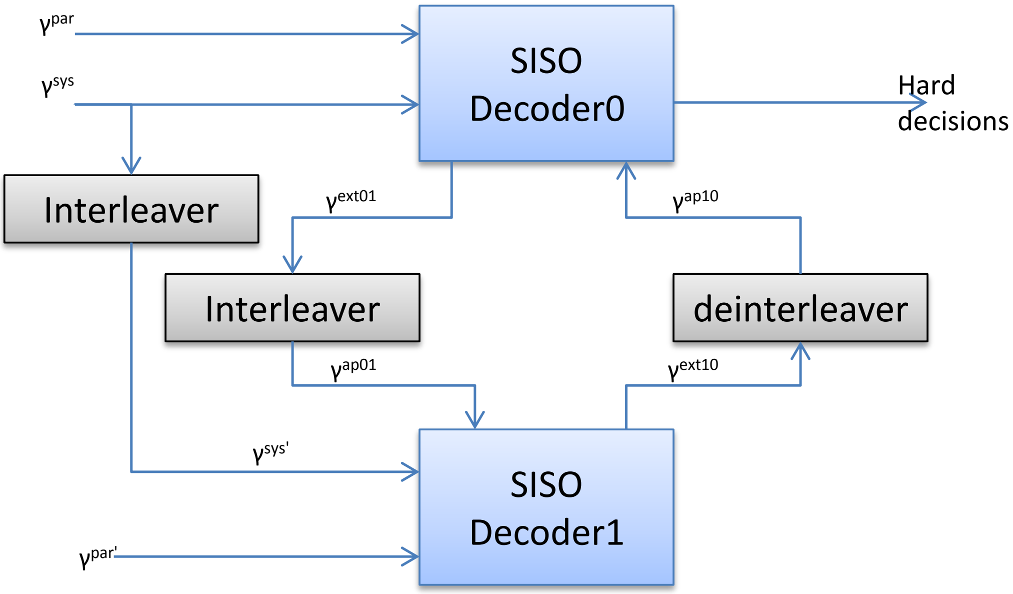


Figure 1.8 — Classical Turbo decoder structure

$$\gamma(d_i) = \ln \frac{Pr(d_i = +1|y)}{Pr(d_i = -1|y)} \tag{1.8}$$

Input LLRs causing trellis transition can be decomposed into 3 independent terms, as

$$\gamma(d_i) = \gamma^{ap}(d_i) + \gamma^{sys}(d_i) + \gamma^{par}(d_i) \tag{1.9}$$

where $\gamma^{ap}(d_i)$ is the a-priori information of d_i, $\gamma^{sys}(d_i)$ and $\gamma^{par}(d_i)$ are the channel measurement of the systematic and parity parts respectively. $\gamma^{ext}(d_i)$ is the extrinsic information that is sent by the SISO decoder to the other decoder (as shown in Figure 1.8). Extrinsic information γ^{ext} from one decoder becomes the a-priori information γ^{ap} for the other decoder at the next decoding stage. In Figure 1.8, γ^{ext01} represents the extrinsic information sent from SISO decoder0 to SISO decoder1, while γ^{ext10} denotes the extrinsic information sent from SISO decoder1 to SISO decoder0. Although different kinds of algorithms are proposed in the literature for this SISO decoding, Soft Output Viterbi Algorithm (SOVA) and Maximum Aposteriori Probability (MAP) algorithms are the most commonly used. The SOVA algorithm is a soft output variant of the classical Viterbi algorithm that aims to find the most likely sequence of the transmitted codeword [13]. This algorithm targets to minimize the frame error rate (FER). The MAP algorithm [14] on the other hand, which also referred to in the literature as Bahl-Cock-Jelinek-Raviv (BCJR) or forward-backward algorithm, targets to minimize the bit error rate (BER). This last algorithm, which is considered in this thesis work, 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.

1.3.1 Maximum Aposteriori Probability (MAP) algorithm

For each source symbol d_i^{sym} comprised of m bits, encoded in n output bits by an encoder having M memory elements (i.e 2^M states) at rate $r = k/n$, a MAP decoder provides 2^m *a posteriori* probabilities given the channel output y received by the decoder. The hard decision on the corresponding value j, i.e. $d_i^{sym} = j$, that maximizes the *a posteriori* probability is expressed in terms of joint probabilities as:

$$Pr(d_i^{sym} = j|y) = \frac{p(d_i^{sym} = j, y)}{\sum_{l=0}^{2^m-1} p(d_i^{sym} = l, y)} \tag{1.10}$$

The trellis structure of the code enables us to decompose the calculation of joint probabilities between past and future observations, given by:

$$Pr(d_i^{sym} = j|y) = \sum_{(s',s)/d_i^{sym}=j} \alpha_i(s')\gamma_i(s', s)\beta_{i+1}(s) \tag{1.11}$$

where:
Forward recursion metric ($\alpha_i(s)$), which gives the probability of the state s at instant i computed from the past values received from the channel, is given by

$$\alpha_{i+1}(s) = \sum_{s'=0}^{2^M-1} \alpha_i(s')\gamma_i(s', s), i \in 0, 1, ...N - 1 \tag{1.12}$$

Backward recursion metric ($\beta_i(s)$), which gives the probability of the state s at instant i computed from the future values received from the channel, is given by:

$$\beta_i(s) = \sum_{s'=0}^{2^M-1} \beta_{i+1}(s')\gamma_i(s', s), i \in N - 1, N - 2, ...0 \tag{1.13}$$

Branch metric ($\gamma_i(s', s)$), which gives the state transition probability from state s' to state s of the trellis at instant i, is given by:

$$\gamma_i(s', s) = P(s_i = s, y_i|s_{i-1} = s') = p(y_i|x_i).Pr^a(d_i^{sym} = d_i^{sym}(s', s)) \tag{1.14}$$

Assuming an equi-probable source, i.e a source that transmits all symbols with equal probability, implies that *apriori* probability ($Pr^a(d_i^{sym} = d_i^{sym}(s', s))$)= $\frac{1}{2^k}$. The channel transition probability for the considered AWGN channel model is given by $p(y_i|x_i)$ and can be expressed as:

$$\begin{aligned} p(y_i|x_i) &= \prod_{l=0}^{n-1} \frac{1}{\sigma\sqrt{2\pi}}.e^{-\frac{(y_{i,l}-x_{i,l})^2}{2\sigma^2}} \\ &= Constant.e^{\sum_{l=0}^{n-1} y_{i,l}x_{i,l}} \end{aligned} \tag{1.15}$$

where x_i and y_i are the i^{th} transmitted modulated symbol and received symbol respectively. The generated extrinsic information does not include the symbol channel input as this part of the information is shared by both SISO decoders, and does not have to be a matter of additional information transfer.

$$Pr^{ext}(d_i^{sym} = j|y) = \frac{\sum_{(s',s)/d_i^{sym}=j} \alpha_i(s')\gamma_i^{ext}(s',s)\beta_{i+1}(s)}{\sum_{(s',s)} \alpha_i(s')\gamma_i^{ext}(s',s)\beta_{i+1}(s)} \tag{1.16}$$

$$\gamma_i^{ext}(s',s) = Constant.e^{\frac{\sum_{l=k}^{n-1} y_{i,l}x_{i,l}}{\sigma^2}} \tag{1.17}$$

1.3.2 Max-Log-MAP approximation

The Log-MAP algorithm is a simplification of the MAP algorithm, introduced in the previous section, that transforms the semi ring sum-product $(R^+, +, \times, 0, 1)$ to semi ring $(R, max^*, +, -\infty, 0)$ where the max^* operator is defined as:

$$\begin{aligned} max^*(x,y) &= \sigma^2 \ln(e^{\frac{x}{\sigma^2}} + e^{\frac{y}{\sigma^2}}) \\ &= max(x,y) + \sigma^2 \ln(1 + e^{-\frac{|x-y|}{\sigma^2}}) \qquad (1.18) \\ &\approx max(x,y) \qquad (1.19) \end{aligned}$$

If equation (1.18) is used for metric computations, then the algorithm is called *Log-MAP* algorithm [15]. In practice a version of the algorithm, called *Max-Log MAP* algorithm that employs equation (1.19) is used [15]. This results in some negligible loss in decoding performance (0.1 db for DBTC) but simplifies the hardware implementation by eliminating the need of lookup tables required to implement the second term of equation (1.18). Using this approximation to compute the forward and backward recursion metrics, equations (1.12) and (1.13) become as follows:

$$\alpha_{i+1}(s) = \max_{s'=0}^{2^M-1}(\alpha_i(s') + \gamma_i(s',s)), i \in 0, 1, ..N-1 \tag{1.20}$$

$$\beta_i(s) = \max_{s'=0}^{2^M-1}(\beta_{i+1}(s') + \gamma_i(s',s)), i \in N-1, N-2, ..0 \tag{1.21}$$

$$\tag{1.22}$$

Similarly, the equations for computing the branch metric become as follows:

$$\gamma_i(s',s) = \sigma^2 \ln \gamma_i(s',s) = \gamma_i^{ext}(s',s) + L_i^a(j) + L_i^{sys}(j) \tag{1.23}$$

$$\gamma_i^{ext}(s',s) = \sigma^2\gamma_i^{ext}(s',s) = Constant + \sum_{l=i}^{n-1} y_{(j,i)}.x_{(j,i)} \tag{1.24}$$

where $\gamma_i^a(j)$ and $\gamma_i^{sys}(j)$ correspond to the apriori and systematic part of the information respectively.

Likewise, denoting $\hat{j}$ as the most probable symbol, the aposteriori (L_i^{apos}) and extrinsic informations (L_i^{ext}) are simplified to:

$$\gamma_i^{apos}(j) = \gamma_i^a(j) + L_i^{sys}(j) + \gamma_i^{ext}(j) - (\gamma_i^a(\hat{j}) + \gamma_i^{sys}(\hat{j})) \tag{1.25}$$

$$\begin{aligned} \gamma_i^{ext}(j) = &\max_{(s',s)/d_i^{sym}=j}(\alpha_i(s') + \gamma_i^{ext}(s',s) + \beta_{i+1}(s)) \\ &- \max_{(s',s)}(\alpha_i(s') + \gamma_i^{ext}(s',s) + \beta_{i+1}(s)) \qquad (1.26) \end{aligned}$$

1.3.3 Max-Log-MAP for Turbo decoding

Considering the above description of Max-Log-MAP algorithm and an 8-state double binary Turbo code, i.e. $m = 2$ bits per symbol, we can summarize the steps for Turbo decoding as follows: The SISO0 process bit LLRs in the natural order corresponding to $(S0, S1, P0, P1)$ while the SISO1 process in the interleaved sequence $(S0^{'}, S1^{'}, P0^{'}, P1^{'})$ corresponding to i^{th} symbol. Where $(S0, S1)$ and $(S0', S1')$ are the systematic bits in natural and interleaved order respectively. Similarly, $(P0, P1)$ and $(P0', P1')$ are the parity bits in natural and interleaved order respectively. Since the trellis is double binary, the input bit LLR's (Λ^j) to the decoder are converted to systematic (γ_i^{sys}) and parity symbol LLR's (γ_i^{par})given by the equations (1.27) to (1.30) below. The i^{th} systematic symbol $\gamma_i^{sys00}(s', s)$ can be calculated as the negative of $\gamma_i^{sys11}(s', s)$. Similarly, $\gamma_i^{sys01}(s', s)$, $\gamma_i^{par01}(s', s)$ and $\gamma_i^{par00}(s', s)$ can be calculated as the negative of $\gamma_i^{sys10}(s', s)$, $\gamma_i^{par10}(s', s)$ and $\gamma_i^{par11}(s', s)$ respectively.

$$\gamma_i^{sys11}(s', s) = \Lambda_{S0}^{2*j} + \Lambda_{S1}^{2*j+1} \tag{1.27}$$

$$\gamma_i^{sys10}(s', s) = \Lambda_{S0}^{2*j} - \Lambda_{S1}^{2*j+1} \tag{1.28}$$

$$\gamma_i^{par11}(s', s) = \Lambda_{P0}^{2*j} + \Lambda_{P1}^{2*j+1} \tag{1.29}$$

$$\gamma_i^{par10}(s', s) = \Lambda_{P0}^{2*j} - \Lambda_{P1}^{2*j+1} \tag{1.30}$$

The decoding consists of calculating the 8 current state metrics for the i^{th} symbol computed from future $(i + 1)$ and previous $(i - 1)$ state metrics, as given by the equations (1.31) and (1.32). $\beta_i(s)$ is the backward state metric of state s for the i^{th} input symbol and computed from the state metrics of the $(i + 1)^{th}$ symbol, i.e. when traversing the trellis in the reverse direction (Backward recursion). Similarly, $\alpha_i(s)$ is the forward state metric and is computed when traversing the trellis in the forward direction (Forward recursion).

$$\begin{gathered}\beta_i(s) = \max_{s'}(\beta_{i+1}(s') + \gamma_i(s', s)) \\ \forall(s', s \in 0, 1, ..7)\end{gathered} \tag{1.31}$$

$$\begin{gathered}\alpha_i(s) = \max_{s'}(\alpha_{i-1}(s') + \gamma_i(s', s)), \\ \forall(s', s \in 0, 1, ..7)\end{gathered} \tag{1.32}$$

$\gamma_i(s', s)$ is the state transition probability from the previous state s' to the current state s as given by equation (1.33). It consists of intrinsic (equation (1.34)) and parity information components.

$$\begin{gathered}\gamma_i(s', s) = \gamma_i^{intr_u}(s', s) + \gamma_i^{par_v}(s', s), \\ \forall(u, v \in 00, 01, 10, 11)\end{gathered} \tag{1.33}$$

$$\begin{gathered}\gamma_i^{intr_u}(s', s) = \gamma_i^{sys_u}(s', s) + \gamma_i^{n.ap_u}(s', s), \\ \forall(u \in 00, 01, 10, 11)\end{gathered} \tag{1.34}$$

The parameter $\gamma_i^{n.ap_u}(s', s)$ in equation (1.34) is the normalized apriori information of the i^{th} symbol or the normalized extrinsic information $(\gamma_i^{n.ext})$ sent by the other decoder component given by the equations (1.35) and (1.36).

$$\begin{aligned}\gamma_i^{ext}(d(s', s) = u) = \gamma_i^{apos}&(d(s', s) = u) \\ &- \gamma_i^{intr_u}(s', s), \\ &\forall(u \in 00, 01, 10, 11)\end{aligned} \tag{1.35}$$

$$\begin{aligned}\gamma_i^{n.ext}(d(s',s)=u) = Z_i^{ext}(d(s',s)=u)\\ -\, min(Z_i^{ext}(d(s',s)=u)),\\ \forall(u \in 00,01,10,11)\end{aligned} \tag{1.36}$$

where $d(s',s)=u$ indicates the decision that the processed symbol being u for a transition from state s' to s. γ_i^{apos} is the aposteriori information of the symbol given by the equation (1.37).

$$\begin{aligned}\gamma_i^{apos}(d(s',s)=u) = \max_{(s',s)/d(s',s)=u}(\alpha_{i-1}(s')\\ +\, \gamma_i(s',s)+\beta_i(s)),\\ \forall(u \in 00,01,10,11)\end{aligned} \tag{1.37}$$

A *Half iteration* consists of one forward-backward recursion processing of all symbols of the received frame either in natural/interleaved order. In this thesis, we consider processing of symbols in natural order is called as the first half iteration and in interleaved order as the second half iteration. Once all the iterations are completed (usually 6-7 iterations) the decoder produces hard decisions ($\gamma_i^{hard\ dec.}$), given by the equations (1.38) and (1.39).

$$\begin{aligned}\gamma_i^{hard\ dec.}(S0) = sign(\max(\gamma_i^{apos}(d(s',s)=01),\\ \gamma_i^{apos}(d(s',s)=00))\\ -\max(\gamma_i^{apos}(d(s',s)=11),\\ \gamma_i^{apos}(d(s',s)=10)))\end{aligned} \tag{1.38}$$

$$\begin{aligned}\gamma_i^{hard\ dec.}(S1) = sign(\max(\gamma_i^{apos}(d(s',s)=00),\\ \gamma_i^{apos}(d(s',s)=10))\\ -\max(\gamma_i^{apos}(d(s',s)=01),\\ \gamma_i^{pos}(d(s',s)=11)))\end{aligned} \tag{1.39}$$

Trellis initialization

In case the trellis is terminated circularly (as in WiMAX), the starting and the ending trellis state metrics of the received frame are initialized as equi-probable, i.e.:

$$\alpha_0(s)=0, s \in (0,1,...2^{M-1}) \tag{1.40}$$

$$\beta_{N-1}(s)=0, s \in (0,1,...2^{M-1}) \tag{1.41}$$

On the other hand, if the trellis is terminated using zero-padding technique (as in LTE), the state with known probability, namely state $s=0$, is initialized with a state metric equals to 0, while the metrics of the other states $s \neq 0$ are initialized to $-\infty$ (in practice, the most negative value of the considered numerical representation).

$$\alpha_0(s=0)=0,\ \alpha_0(s\neq 0)=-\infty, s \in (1,2...2^{M-1}) \tag{1.42}$$

$$\beta_{N-1}(s=0)=0,\ \beta_{N-1}(s\neq 0)=-\infty, s \in (1,2...2^{M-1}) \tag{1.43}$$

1.3.4 Parallelism in Turbo decoding

From the equations (1.27) to (1.37), we can infer three levels of parallelism that can be exploited in implementation of the Turbo decoder in order to achieve high throughput [16], namely:

- Metric level

- SISO decoder level
- Turbo decoder level

The following sub-sections summarize briefly the different parallelism techniques available at each of these levels.

1.3.4.1 Metric level parallelism

The metric level parallelism concerns the processing of all metrics involved in the decoding of each received symbol inside a MAP SISO decoder. It exploits the inherent parallelism of the trellis structure and the parallelism of the MAP computations [16, 17].

Parallelism of trellis transitions: Trellis-transitions parallelism refers to the trellis structure as the same operations related to the computation of γ, α, β and the extrinsic information (γ^{ext}) should be repeated for all the trellis transitions. Thus, the first metric (γ) calculations can be done in parallel to the maximum degree of parallelism bounded by the number of transitions in the trellis. The degree of parallelism associated with the computation of the branch metric is bounded by the number of possible binary combinations of input and parity bits. For example, in WiMAX DBTC case, which has 2 bits for systematic and parity, there can be $2^2 \times 2^2 = 16$ different branch metric combinations. The other metrics α, β and extrinsic computation can be parallelized with a bound of total number of transitions ($2^M \times 2^{(bits\ per\ symbol)}$) in a trellis. This parallelism implies low area overhead as only the computational units have to be duplicated. 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 MAP computations: The structure of the MAP algorithm permits parallel execution of the three MAP computations (α, β and extrinsic computation γ^{ext}) [18]. Two scheduling strategies are followed, namely:

1. **Forward-Backward schedule**: Parallel execution of backward recursion and extrinsic computations was proposed with the original forward-backward schedule as depicted in Figure 1.9. First the Forward recursion (α) is calculated (parallelism degree =1), followed by backward recursion (β) and extrinsic computation computed in parallel (parallelism degree =2).

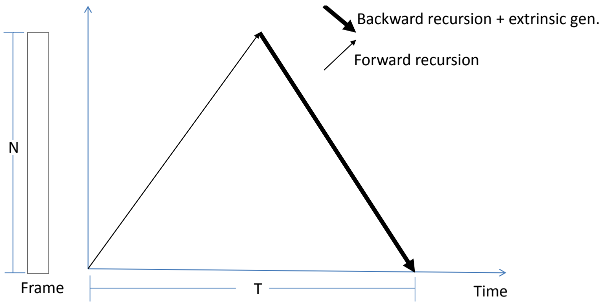


Figure 1.9 — Turbo decoding: Forward-Backward schedule

2. **Butterfly schedule**: This schedule doubles the parallelism degree of the Forward-Backward schedule by calculating the forward and backward recursions in parallel as

shown in Figure 1.10. Thus the parallelism degree is double that of Forward-Backward schedule. This results in factor 2 reduction in computation time with doubling of MAP computation resources but with no increase in memory. Metric computation parallelism is area efficient.

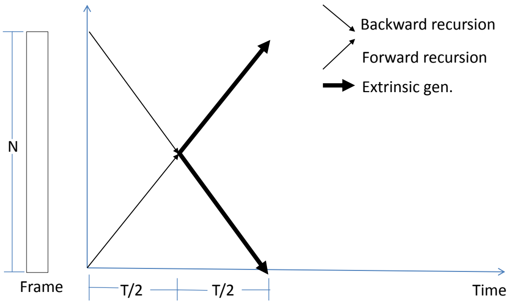


Figure 1.10 — Turbo decoding: Butterfly schedule

High radix trellis compression: The throughput can further be enhanced by adopting radix-2^n computation as used in [19], where n is the number of trellis sections that compressed in a single trellis. Figure 1.11 shows two trellis sections of a 4-state single binary trellis compressed to form a 4-state double binary trellis. This gives rise to decoding of two source data bits at the same time, thus increasing the throughput by 2 in the SBTC case (provided the duplication of the related hardware resources). In this case, the generation of individual aposteriori LLR's are done by the following equations:

$$\begin{aligned}\gamma_i^{ext}(d(s',s)=1) &= max\{\gamma_i^{ext}(d(s's)=01),\gamma_i^{ext}(d(s',s)=00)\}\\ &\quad - max\{\gamma_i^{ext}(d(s',s)=11),\gamma_i^{ext}(d(s',s)=10)\} \qquad (1.44)\\ \gamma_{i+1}^{ext}(d(s',s)=1) &= max\{\gamma_i^{ext}(d(s's)=01),\gamma_i^{ext}(d(s',s)=00)\}\\ &\quad - max\{\gamma_i^{ext}(d(s',s)=11),\gamma_i^{ext}(d(s',s)=10)\} \qquad (1.45)\end{aligned}$$

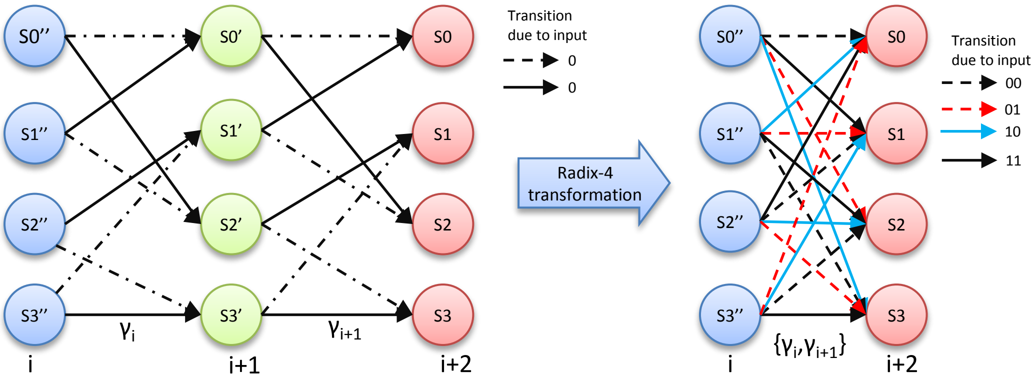


Figure 1.11 — Radix-4 trellis compression

Similarly, the definition of the branch metric γ and state metric calculations (α, β) are modified as follows:

$$\alpha_i(s) = \max_{s''}\{\alpha_{i-2}(s^{''}) + \gamma_i(s^{''},s)\} \qquad (1.46)$$

$$\beta_i(s) = \max_{s''}\{\beta_{i+2}(s^{''}) + \gamma_i(s^{''},s)\} \qquad (1.47)$$

$$\gamma_i(s^{''},s) = \gamma_{i-1}(s^{''},s^{'}) + \gamma_i(s^{'},s) \qquad (1.48)$$

1.3.4.2 SISO decoder level parallelism

This level of parallelism exploits the use of multiple SISO decoders, each executing the MAP algorithm and processing a sub-block of the same frame in natural or interleaved 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 $N_{sub-blocks}$ sub-blocks and then each sub-block is processed by a MAP-SISO decoder using adequate initializations. Besides duplication of the SISO decoders, this parallelism imposes two other constraints:

- The interleaving has to be parallelized in order to scale proportionally to the number of SISO decoders added. Because of the scramble property of interleaving, this parallelism can induce communication conflicts in accessing memories that store channel/extrinsic values. The interleavers of emerging standards are conflict-free for certain parallelism degrees (as explained in Section 1.2.2). In case of conflicts, an appropriate communication structure, e.g. Network On Chip (NoC), should be implemented for conflict management [20].
- The MAP-SISO decoders have to be initialized adequately; this is done either by acquisition or by message passing.

  1. Acquisition method: The acquisition method involves estimating sub-block boundary recursion metrics using an overlapping region, called acquisition window or prologue, as illustrated in Figure 1.12. The state metrics of the SISO decoders are initialized to zero and backward/forward recursion is carried out in order to find the more reliable state metric at the sub-block boundary. This has two implications on the implementation. First of all, extra memory is required to store the overlapping windows. Secondly, extra time will be required for performing the acquisition related computations, which impacts the throughput performance of the decoder.

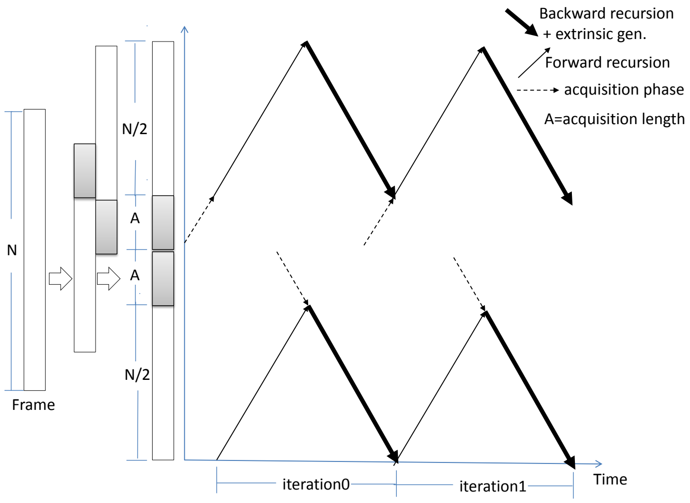


Figure 1.12 — Sub-blocking with initialization through acquistion

  2. Message passing: In this method, sub-block boundaries are initialized with recursion metrics computed during the previous iteration in the neighboring sub-blocks (see Figure 1.13). Thus, there is no need for overlapping windows and the related extra memory, in addition the time overhead is negligible. In [16] a detailed analysis of the

parallelism efficiency of these two methods is presented which clearly favors the use of message passing technique.

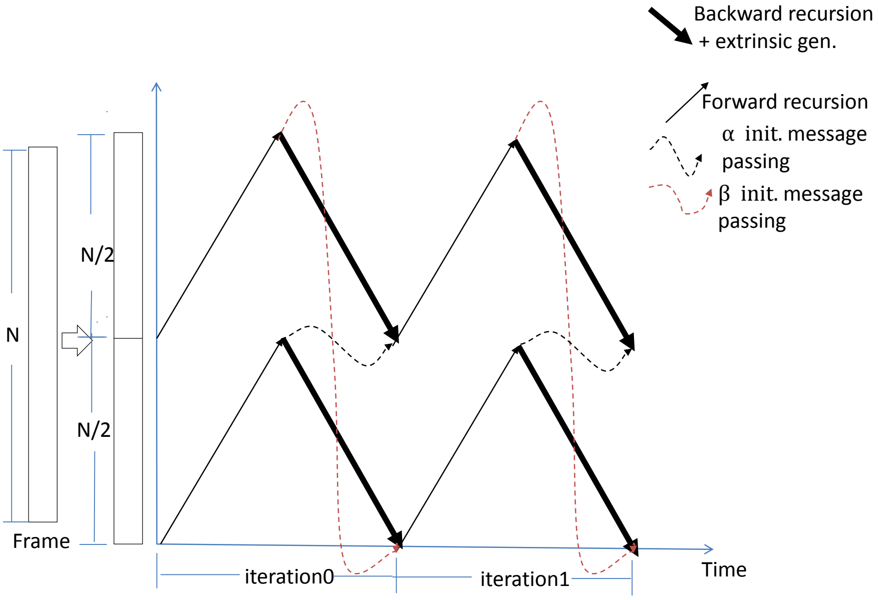


Figure 1.13 — Sub-blocking with initialization through message passing

Windowing: The LTE standard specifies a target throughput of 150 Mbps and has a maximum frame length of 6144 bits. The previous paragraph discussed achieving high throughputs via sub-block parallelism, wherein the incoming frame is divided into sub-blocks and each SISO decoder operates on a sub-block with message passing across sub-block boundaries. Assuming Backward-forward scheduling (which is similar to forward-backward schedule except here backward recursion is scheduled first) is used, using only sub-block parallelism would require storing the intermediate β state metrics calculated in the backward recursion in the memory. Thus, the state metric memory (henceforth called the cross metric memory) has to have a memory depth equals to the sub-block length. In order to increase the area efficiency of the design, sub-blocks are further divided into L windows usually of size 64-128 symbols. Thus, the cross metric memory depth is now reduced to the size of the window (with an additional requirement of storing state metric boundaries values of all windows to be used in the next iteration). Each SISO decoder processes the sub-block, window by window using message passing for state metric initializations across window boundaries as shown in Figure 1.14.

Shuffled Turbo decoding: The basic idea of shuffled decoding technique [21] 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 as soon as available. The shuffled decoding technique performs decoding (impacts computation time) and interleaving (impacts communication time) fully concurrently while serial decoding implies waiting for the update of all extrinsic information before starting the next half iteration (Figure 1.15). Thus, by doubling the number of MAP SISO decoders, component-decoder parallelism halves the iteration period in comparison with originally proposed serial Turbo decoding. Nevertheless, in return to the high parallelism degree an overhead of iteration between 5 and 50 percent is incurred depending on the MAP computation schedules, on the degree of sub-block parallelism,

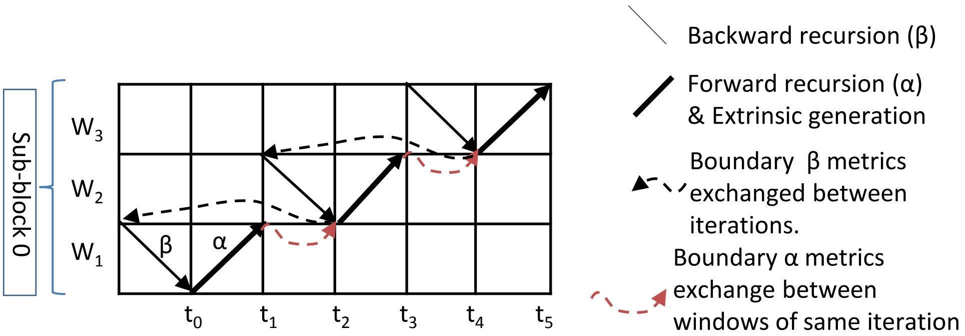

Sub-block 0
W3
W2
W1
β
α
t0
t1
t2
t3
t4
t5
Backward recursion (β)
Forward recursion (α) & Extrinsic generation
Boundary β metrics exchanged between iterations.
Boundary α metrics exchange between windows of same iteration

Figure 1.14 — Sub-blocking and windowing with initialization through message passing

on propagation time, and on interleaving rules, to attain the same BER performance of serial decoding [16].

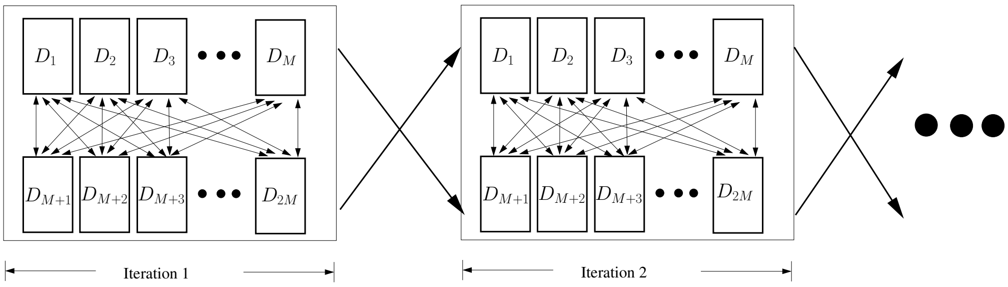


Figure 1.15 — Turbo decoding: Shuffled decoding strategy, where D_x= SISO decoders x=1,2,..

1.3.4.3 Turbo decoder level parallelism

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 Low Density Parity Check codes

Low-density parity-check (LDPC) codes constitute the other class of error correction codes that have received popularity over the past decade in the coding community because of their excellent error correction capability and near-capacity performance. They were first introduced by Gallager in his PhD thesis in 1960 [22], but they were ignored until 1997. They were rediscovered independently by MacKay (1997) [23] and Richardson/Urbanke (1998) [24] as an alternative to the capacity achieving codes, namely Turbo Codes. Authors in [25] report to have constructed LDPC codes and measured bit error rate (BER) performance that comes very close to the Shannon limit for the AWGN channel (within 0.04 dB at BER=10^{-6}) with iterative decoding and very long frame size of 10^7. The LDPC codes have been recently adopted in many standards, such as IEEE 802.11n (WiFi), IEEE 802.16e (WiMAX), DVB-S2, etc. The following sections give a brief background of linear block codes, to which LDPC codes belong, followed by a description of QC-LDPC codes adopted in recent wireless communication standards and a presentation of the most commonly used decoding algorithm, namely Normalized Min-Sum (NMS).

1.4.1 Linear block codes

Unlike convolution codes, where the encoding process depends on the past inputs and current state of the encoder, linear block codes operate on blocks of data of length k. The output encoded data is of length n and consists of the actual input data and redundancy bits. The redundancy bits use a linear combination of input bits or group of bits over Galois field 2^m. If $m = 2$, the arithmetic operations can be mapped over logical XOR (modulo-2 adder) and AND (modulo-2 multiplier) operations.

The encoding process consists of forming a n bit codeword from k information bits by adding $n - k$ parity bits. In other words, encoding can be simply done by choosing 2^k vectors out of 2^n as valid codewords for transmitting, as shown in Figure 1.16 for (n=3,k=2) linear block code. As we can see, there are many possibilities of mapping. One such particular case of mapping is called systematic encoding, where the input message bits are found in the beginning (or end) of the codeword while the rest of the bits are parity bits. This method simplifies the encoding process at the encoder and also the realization of the decoder.

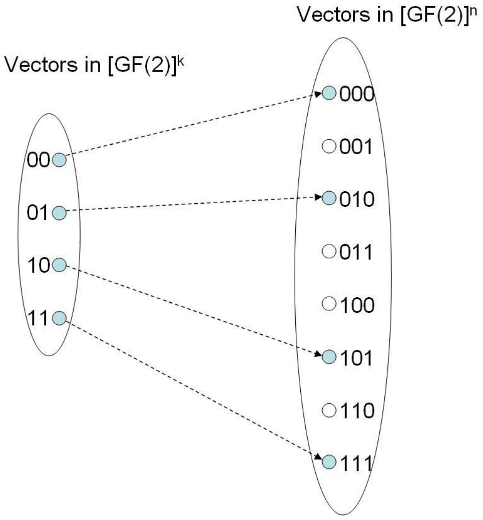


Figure 1.16 — Linear block code mapping example for (n=3, k=2)

The encoding process consists of a simple multiplication (over GF(2)) of the input vector X with the generator matrix G to obtain the codeword C as illustrated in the example of Figure 1.17 and equation (1.49). In this example the generator matrix G is composed of an identity matrix $I_{(k,k)}$ of size k and a parity matrix $P_{(n-k,n)}$.

$$C = X.G, \; e.g. \; X = [0100], \; C = [0100\,000] \tag{1.49}$$

Decoding on the other hand is done by matrix multiplications over GF(2) of the received codeword R, with the parity check matrix H as illustrated in the example of Figure 1.17 and

$$G=\begin{array}{cccc|ccc} & & I_{(k,k)} & & & P_{(k,\,n\text{-}k)} & \\ 1&0&0&0&1&1&1\\ 0&1&0&0&0&1&1\\ 0&0&1&0&1&0&1\\ 0&0&0&1&1&1&0 \end{array}\qquad H=\begin{array}{cccc|ccc} & & P^{T}_{(n\text{-}k,k)} & & & I_{(n\text{-}k,n\text{-}k)} & \\ 1&0&1&1&1&0&0\\ 1&1&0&1&0&1&0\\ 1&1&1&0&0&0&1 \end{array}$$

Figure 1.17 — Linear block code generator matrix G and parity check matrix H for (n=7,k=4)

equation (1.50).

$$S = R.H^T,\ e.g.\ R = [0100\,001],\ S = [011] \tag{1.50}$$

If the resulting vector S (called the *syndrome*) is zero vector, it indicates no errors have been introduced. The decoding process can also be represented using a *Tanner graph* (Figure 1.18), where the ones in the H matrix are shown as the lines connecting the variable and check nodes [26]. The variable nodes (n equals to the number of columns) are initialized with the input message and the check nodes (m equals to number of rows) implement XOR operations and calculate if the resulting syndrome S evaluates to zero. If the syndrome is non-zero, then a lookup table based decoding is used to compare the received codeword and the syndrome obtained to detect and correct the error.

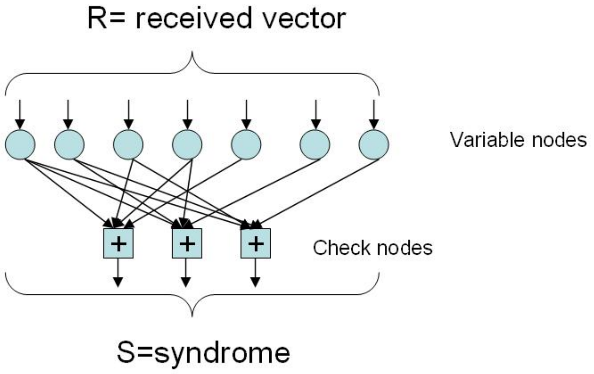


Figure 1.18 — Tanner graph of H matrix in 1.17

Different types of linear block codes are proposed in the literature and used in existing communication systems, such as LDPC codes, Reed-Solomon codes (based over GF(2^m)), BCH codes, Goley codes, etc.

1.4.2 QC-LDPC codes

LDPC codes are a class of linear block codes specified by a very sparse binary parity check matrix H:

$$H.x^T = 0 \tag{1.51}$$

where x is a codeword and H is the check matrix that has M rows and N columns. The parity-check matrix H of an LDPC code can be seen as a concatenation of parity-check matrices $H^0, H^1, ..., H^{M-1}$, corresponding to the super-codes $C^0, C^1, ...C^{N-1}$, hence can be regarded as a class of concatenated codes [27].

In the Gallager's original LDPC code design, there were a fixed number of ones in both the rows (M) and the columns (N) of the parity check matrix. Furthermore, each bit is used in say L parity check constraints and each parity check constraint is the XOR of L bits. This class of codes is referred to as regular LDPC codes. Current standards specify irregular LDPC codes which contain unequal number of ones along rows and columns in the check matrix. Irregular LDPC codes provide generally better BER performance characteristics when compared to regular LDPC codes. The number of ones along the row of a check matrix is called Variable node degree, in the Tanner graph representation, is the number of edges that connect to the variable node, while check node degree is the number of edges that connect to the check node. As the focus of the thesis is the irregular LDPC codes adopted in the standards, namely WiFi and WiMAX, this section discusses only the related family of LDPC codes.

I_{1}	I_{25}	I_{55}		I_{47}	I_{4}		I_{91}	I_{84}	I_{8}	I_{86}	I_{52}	I_{82}	I_{33}	I_{5}	I_{0}	I_{36}	I_{20}	I_{4}	I_{77}	I_{80}	I_{0}		
	I_{6}		I_{36}	I_{40}	I_{47}	I_{12}	I_{79}	I_{47}		I_{41}	I_{21}	I_{12}	I_{71}	I_{14}	I_{72}	I_{0}	I_{44}	I_{49}	I_{0}	I_{0}	I_{0}	I_{0}	
I_{51}	I_{81}	I_{83}	I_{4}	I_{67}		I_{21}		I_{31}	I_{24}	I_{91}	I_{61}	I_{81}	I_{9}	I_{86}	I_{78}	I_{60}	I_{88}	I_{67}	I_{15}			I_{0}	I_{0}
I_{50}		I_{50}	I_{15}		I_{36}	I_{13}	I_{10}	I_{11}	I_{20}	I_{53}	I_{90}	I_{29}	I_{92}	I_{57}	I_{30}	I_{84}	I_{92}	I_{11}	I_{66}	I_{80}			I_{0}

Figure 1.19 — LDPC check matrix representations: H_{base}

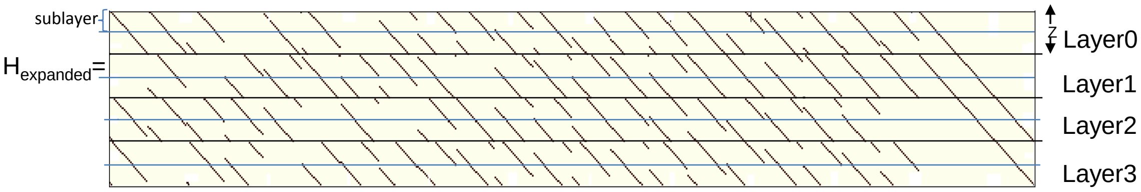


Figure 1.20 — LDPC check matrix representations: $H_{expanded}$ form of 1.19

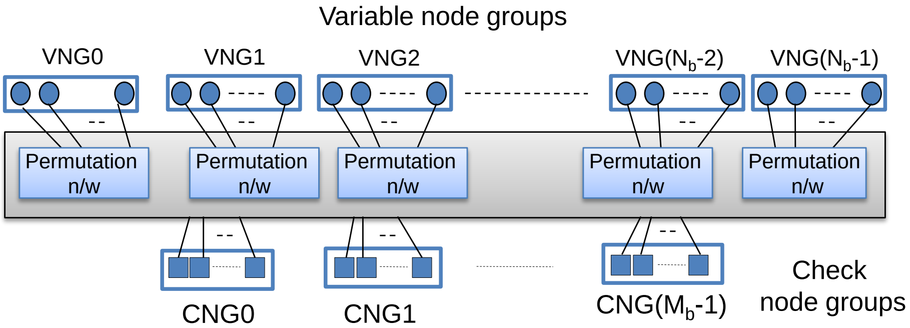


Figure 1.21 — LDPC check matrix representations: Generalised Tanner graph representation of LDPC H-matrix

Figure 1.19 shows the typical format of the H_{base} specified to define LDPC check matrix in the considered wireless communication standards (WiFi and WiMAX). It consists of N_b block columns and M_b block rows where each of the non-negative values is replaced by a permutation matrix of size $Z \times Z$. Z is the so-called *expansion factor* specified by the standard. The negative values are replaced by zero square matrix of size Z. LDPC codes whose check matrices can be expressed in this form are called Quasi-Cyclic LDPC (QC-LDPC) codes.

The LDPC check matrix can also be represented in a graphical form with a bipartite Tanner graph (Figure 1.21) as illustrated in the previous section on linear block codes. In the LDPC case, each column and row in $H_{expanded}$ represents a variable node and a check node respectively. The particular structure of QC-LDPC codes allows to define Check Node Groups $CNG_x \; \forall \; x = 0, 1..(M-1)$ of size Z connected to Variable Node Groups $VNG_y \; \forall \; y = 0, 1..(N-1)$. The lines connecting the variable and check node groups are representative of the ones in the check matrix H (refer Figure 1.20).

Although, LDPC encoding through the method of generator matrix as in the case of linear block codes is possible, it is seldom done in implementations due to the large frame sizes. Alternatively, [5] and [28] specifies different ways of encoding of LDPC codes directly from the check matrix definitions for WiFi or WiMAX LDPC codes respectively.

1.4.3 LDPC in WiFi and WiMAX standard

Low-density parity-check (LDPC) codes have been adapted in new standards such as in the WiFi [28] and WiMAX [5] 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. InWiMAX 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 i.e. 27,54 and 81. Table 1.1 summarizes the LDPC code parameters for these two standards. The maximum required channel throughput is up to 75 Mbps for the WiMAX, and 450 Mbps for WiFi.

Standard	WiMAX			WiFi		
Parameters	#	min	max	#	min	max
Frame lengths	19	576	2304	3	648	1944
Sub-matrix 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 1.1 — Characteristics of WiFi and WiMAX LDPC check matrices

1.5 Low Density Parity Check decoding

Several algorithms are proposed for LDPC decoding and most of them are derived from the well-known belief propagation (BP) algorithm. The principle consists of exchanging iteratively messages (probabilities, or beliefs) along the edges of the Tanner graph. The message $L(n,m)$ 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 $(L(n))$ 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 $L(m,n)$ depends on all messages received from connected VNs except the one being updated (as shown in Figure 1.22). 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 1.51) or another stopping criteria are satisfied or the maximum number of iterations is reached.

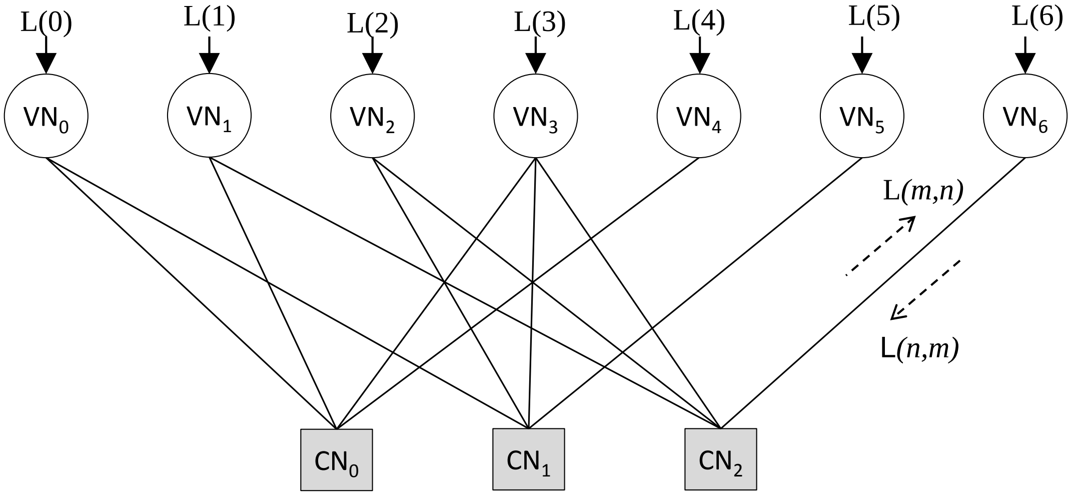


Figure 1.22 **—** Tanner graph example showing the two-phase message passing decoding

1.5.1 LDPC decoding algorithm: Normalized Min-Sum (NMS)

The Min-sum decoding algorithm [29] for LDPC codes are usually described in the Log-domain using LLR to reduce the computational implementation complexity. The Min-Sum algorithm is the hardware efficient implementation of the Sum-product/Belief propagation algorithm [22]. It proposes an approximation to simplify the calculations of updated messages. The message passing is summarized as follows:

Let, the a posteriori probability (APP) Log-Likelihood Ratio (LLR) of each bit n is defined as:

$$L_{ch}(n) = \log \frac{Pr(n=0)}{Pr(n=1)} \tag{1.52}$$

The check node message from check node m to variable node n is denoted as $L(m,n)$. Similarly, the message from variable node n to check node m is denoted as $L(n,m)$.

1. Initialization: The variable message $L(n)$ is initialized to the channel LLR input and $L(m,n)$ are initialized to zero.

2. Variable Node Update: For each check node m, the new check node messages $L(n,m)$, corresponding to all variable nodes j that participate in this parity check equation, are computed using the belief propagation algorithm:

$$L(m,n) = \prod_{j \in N(m)\backslash n} sign(L(j,m))\Psi(\sum_{j \in N(m)\backslash n} \Psi(L(j,m))) \tag{1.53}$$

where $N(m)$ is the set of variable nodes that are connected to check node m, and $N(m)\backslash n$ is the set $N(m)$ with variable node n excluded. The non-linear function $\Psi(x)$ is defined as

$$\Psi(x) = -\log(\tan(\frac{|x|)}{2}) \tag{1.54}$$

To reduce the implementation complexity, the sub-optimal min-sum algorithm [30] can be used to approximate the non-linear function $\Psi(x)$. The normalized min-sum and the offset min-sum algorithms are the two most often used algorithms. For offset min-sum algorithm the equation (1.53) is changed with a offset factor β, as shown below:

$$L(m,n) = \prod_{j \in N(m)\backslash n} sign(L(j,m)). \min_{j \in N(m)\backslash n} (|L(j,m)|) - \beta \tag{1.55}$$

For neglible performance loss, normalized min-sum algorithm is used neglecting the offset factor β and using a scaling factor α. Thus, the above equation of check node update is written as:

$$L(m,n) = \alpha. \prod_{j \in N(m)\backslash n} sign(L(j,m)). \min_{j \in N(m)\backslash n} (|L(j,m)|) \tag{1.56}$$

3. Check Node Update: The a posteriori LLR messages L_n are computed as:

$$L(n) = \sum_{j \in M(n)} L(j,n) \tag{1.57}$$

   where $M(n)$ is the set of check nodes that are connected to variable node n. The variable to check node update message $L(n,m)$ is computed as :

$$L(n,m) = L(n) - L(m,n) \tag{1.58}$$

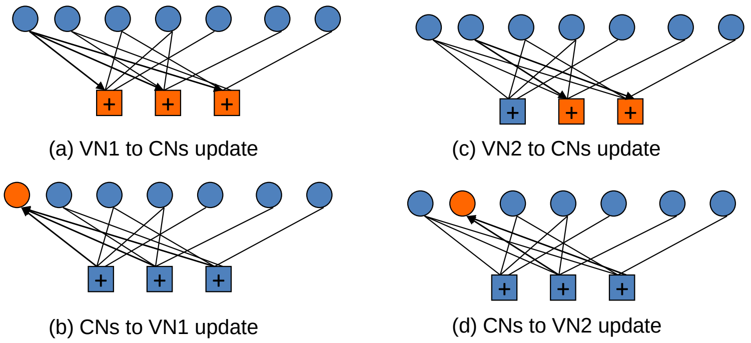
4. Parity check: If all the parity check equations are satisfied, the decoding process is completed, else the above two phases are computed to start a new iteration.

1.5.2 Scheduling

The previous section described flooding schedule, which is the classical way of scheduling the BP algorithm. Several other scheduling techniques have been investigated in the literature and a summary is given below:

1. **Vertical shuffle**: In [21] authors proposed a shuffled BP algorithm which converges faster than the BP algorithm. In this schedule, the idea is to update the information as soon as it has been computed and so that the next check node /variable node uses most updated information. The decoding process is as follows:
   At each step all the CN that are connected to a VN are updated according to equation (1.63). This is followed by the update of related VN as given by the equation (1.59). Thus all the variable node are processed one after the other. This schedule is also called as vertical shuffle. The following figure explains the decoding schedule in the context of linear block code presented in Section 1.4.1.

2. **Horizontal shuffled**: The logical data flow of the decoding process is shown in Figure 1.24, for the check matrix of the linear block code presented in Section 1.4.1. The figure shows the an alternative to the previous scheduling technique. Here, a check node receives updates from several variable nodes connected to it followed by the check node updating those variable nodes.



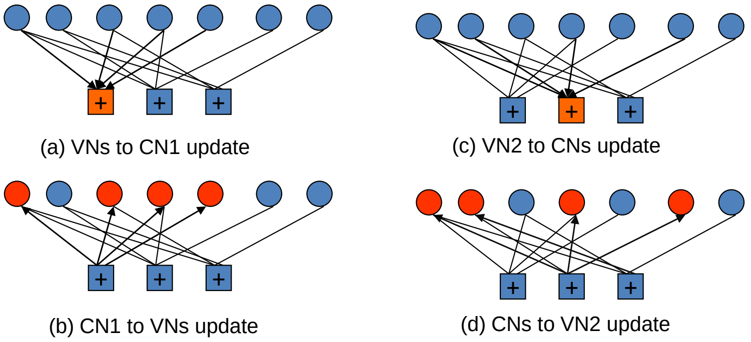
(a) VN1 to CNs update

(c) VN2 to CNs update

(b) CNs to VN1 update

(d) CNs to VN2 update

Figure 1.23 — LDPC decoding using vertical schedule



(a) VNs to CN1 update

(c) VN2 to CNs update

(b) CN1 to VNs update

(d) CNs to VN2 update

Figure 1.24 — LDPC decoding using horizontal schedule

In the context of QC-LDPC codes which is the type of LDPC codes specified in WiMAX and WiFi standard the authors in [27] introduced the concept of Turbo Decoding Message Passing (TDMP, also referred as layered decoding) where block rows (check nodes) are seen as super-codes. Each super-code of the H matrix is a parity-check code corresponding to the row (layer) of the H_{base} matrix. The layered decoding algorithm decodes a codeword iteratively in a M sub-iterations which are equal to the number of super-codes, with one sub-iteration per constituent super-code. This can be considered as a variant of the horizontal schedule presented here except that group of check nodes that are not connected the same variable nodes are schedule for decoding at a time. Once both the updates are complete the next check node is scheduled for processing. Furthermore, this scheduling improves the decoding convergence speed by a factor of 2.

1.5.3 Modified NMS formulation for implementation

In this section, we present the computations implied by NMS algorithm with a slightly modified formulation adapted to the hardware architecture presented in this thesis.

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 (1.58) 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.

$$L^i(n,m) = L^i(n) - L^{i-1}(m,n),\ L^0(m,n) = 0 \tag{1.59}$$

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 all the 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 (1.60). The sign of the message is the product of the signs of the messages received from all VNs, as given in equation (1.61).

$$min^i_{n'm} = \min_{n' \in N(m)\backslash n} (|L^i(n', m)|) \tag{1.60}$$

$$sgn^i_m = \prod_{n \in N(m)} sgn(L^i(n,m)) \tag{1.61}$$

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

$$RV^i(m) = [min0, min1, ind, sgn]^i_m \tag{1.62}$$

The extrinsic information ($L^i(m,n)$) is derived at the variable node as

$$L^i(m,n) = sgn(L^i(n,m)) \times sgn^i_m \times (\alpha \times (min0^i\ or\ min1^i)) \tag{1.63}$$

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

$$L^i(n) = L^i(n,m) + L^i(m,n) \tag{1.64}$$

The sign of $L^i(n)$ indicates the hard decision on the decoded bit.

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

1.6 Summary

This chapter provided the basic background on Turbo and LDPC codes along with their construction and decoding algorithms. Max-Log MAP and Normalized Min-Sum algorithms were illustrated to be the hardware efficient versions of the MAP and Sum-Product algorithms respectively. The different parallelism levels which can be exploited in the implementation of a Turbo decoder were also explained. For LDPC decoding, a brief presentation on existing computation scheduling techniques was given. Finally, the modified NMS formulation adopted in this thesis work was presented.

CHAPTER

2 ASIP Design Methodology and State of the Art in Channel Decoder Design

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 applications, and particularly in the considered digital communication domain. The availability of well established design methodology and tools further promotes this trend.

The first section of this chapter introduces the evolution of embedded processor architectures towards customizable instruction-set ones illustrating our main motivation behind the selection of Application-Specific Instruction-set Processor (ASIP) design approach. It also presents an overview on existing ASIP design flows and presents the considered Processor Designer tool.

The second section gives an overview on state-of-the-art efforts in channel decoder design in order to clarify the position of the proposed contributions in this thesis work. A considerable amount of contributions have been proposed in this challenging domain over the past years targeting many different variants in terms of classes of error correction codes, algorithmic and architecture optimization techniques, design objectives, and design approaches. The proposed overview is far from being exhaustive, however a selection of recent works related to the thesis scope in terms of flexibility support of Turbo and LDPC decoding is presented.

One of these related recent contributions has been carried out at the Electronics department of Telecom Bretagne using the ASIP design approach and targeting flexible Turbo decoding. Thus, the third section of this chapter presents this initial ASIP architecture, namely TurbASIP, which constitutes the starting point of this thesis work.

2.1 Customizable embedded processors

A large share of the integrated circuits manufactured today feature an impressive complexity of hundreds of millions of transistors [31]. The non-recurrent engineering costs of such high-end application-specific integrated circuits are becoming hardly bearable by many products individually. In addition, product life cycles are becoming in general more and more short due to the accelerated emerging of new applications and services (often approaching one year in the consumer electronics market). This new context drives several trends in design approaches and architecture models. Regarding design approaches, system level design and high level synthesis methodologies to accelerate the design cycle and to reduce the non-recurrent engineering costs are being widely investigated. Concerning the architecture model, opportunities for modular reuse are pushing industry to use more and more flexible (or software-programmable) solutions for practically every class of devices and applications.

In traditional design of flexible hardware architectures, the flexibility is incorporated by the expert designer through the use of initialization parameters loaded from a configuration memory or input ports of the design. The architecture description involves manual design of a Finite State Machine (FSM) that controls the different design units of the pipeline taking into account the various supported parameters. But when the number of flexibility parameters increases, the design and validation of such parametrized control logic become more and more complicated.

On the other hand, instruction-set based processors provide inherently high flexibility in terms of control logic design. Their architectures have evolved dramatically in the last couple of decades [31]: from microprogrammed finite state machines, processors have transformed into single rigid pipelines; then, they became parallel pipelines so that various instructions could be scheduled for execution 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, the great majority of the high-performance processors produced today address relatively narrow classes of applications. This is to design customized processors to the very needs of the application rather than having them as rigid fixed entities. The emergence of this trend has been made successful given the development of new design methodologies and tools. Such tools enable the designers to specify a customizable processor in weeks rather than months. Leading companies in providing such methodologies and tools include Processor Designer tool [32] and ARC cores by Synopsys [33] and Tensilica [34]. The shape and boundaries of the architectural space covered by the tool chain differentiate the design approach attempted. These approaches can be classified roughly into three categories [31]:

Parameterizable processors are families of processors belonging to a single type 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, etc.) and others can be scaled (main data-path width, number and type of execution pipelines, number of registers, etc.).

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 [35, 36].

Custom processor development tools are frameworks to support architects in the effort to design from scratch a completely custom processor with its complete tool chain (compiler, simulator, etc.). Ideally, it uses an Architectural Description Language (ADL) to describe the architecture of the design from which all tools and the synthesizable description of the core can be generated [32,37].

It is worth noting that these approaches are not mutually exclusive, as some characteristics can overlap, for example a parameterizable processor may also be extensible, or a template processor in a processor development framework can be easily parameterized and is naturally extensible. All these approaches fall under the name of customizable processors and often are referred as ASIP for Application-Specific Instruction-set Processors (ASIPs).

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 Architectures). Although several specific rASIP architectures have been proposed in the literature (e.g. [38,39]) and several design methodologies are emerging recently (e.g. [40,41]), 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.

2.1.1 Application-Specific Instruction-set Processors

ASIPs are tailored to particular applications, thereby combining performance and energy efficiency of dedicated hardware solutions with the flexibility of a programmable solution. The main idea is to design a programmable architecture tailored to a specific application, thus preserving a much higher degree of flexibility than a dedicated ASIC solution. Several alternative solutions to ASIPs are available in order to implement the desired task, depending on the requirements: for functions which need lower processing capacity but should be kept flexible a software implementation running on a General-Purpose Processor (GPP) or a microcontroller may be the best solution; if some additional processing capacity 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 dedicated hardware blocks (Application Specific Integrated Circuits, ASICs, or even physically optimized ICs), with no flexibility feature. If some flexibility is required, FPGAs which allow for reconfiguration after fabrication, may be the right choice if some price in terms of performance, area and power can be acceptable. 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. Comparison of these implementation methods in regard to performance, flexibility and power consumption is best described in Figure 2.1 [42].

2.1.2 ADL-based design tool: Processor designer

Processor Designer is an ASIP design environment belonging to the category of *custom processor* that uses LISA ADL [43]. The LISA language (Language for Instruction Set Architecture) and the tool were the results of the research work conducted at Aachen University of Technology, Germany. It was first commercialized by the startup LISATek which was then acquired by CoWare in 2003 and later by Synopsys in 2010. The tool was developed with a simulator-centric view [31] and uses a C-like language (LISA) for design description

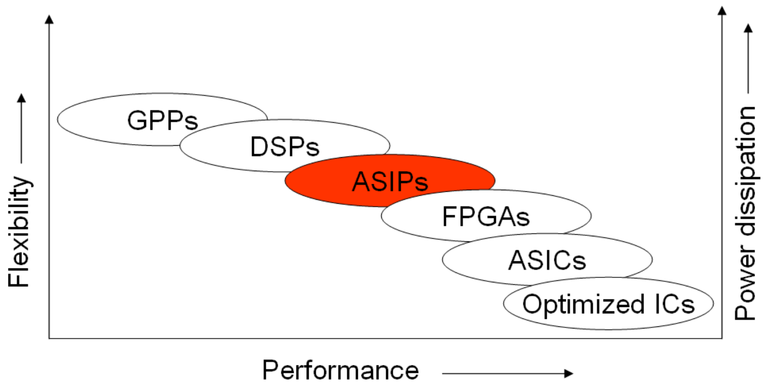


Figure 2.1 — Comparison of performance, flexibility and power dissipation trade-off of implementation methods

of programmable architectures, 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 tool'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 [42]:

- 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 are 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.

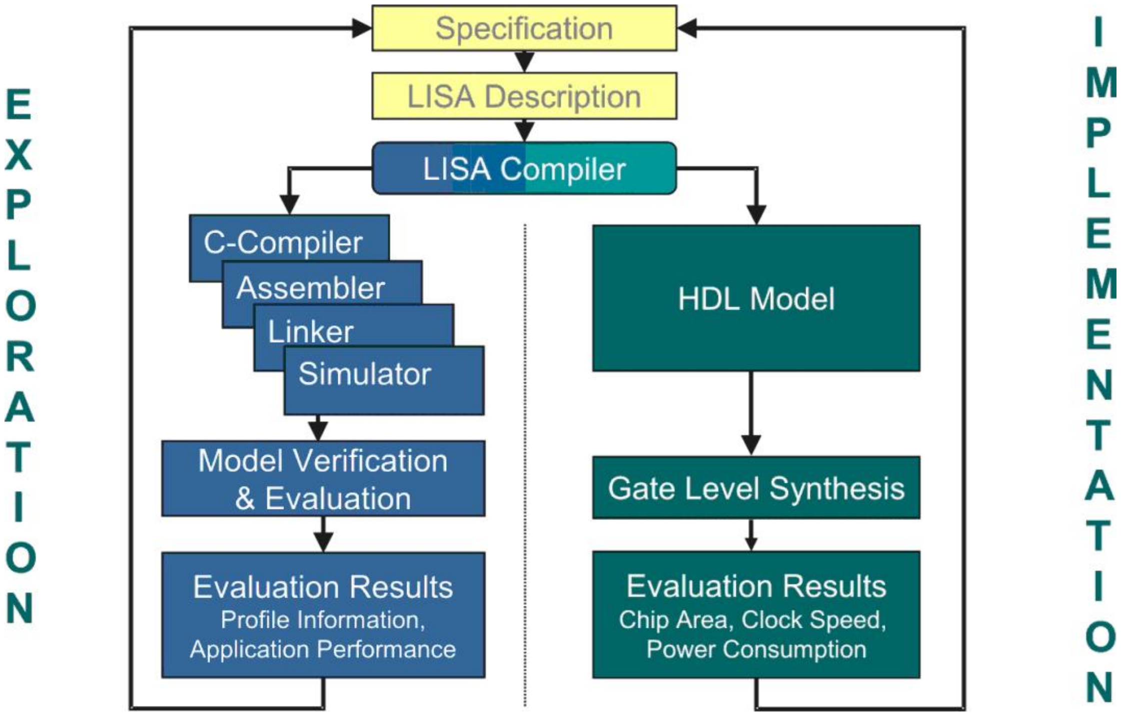


Figure 2.2 — LISA-based ASIP architecture exploration flow

2.1.3 Classical ASIP design flow

Figure 2.3 shows the classical ASIP design flow adopted with the Processor Designer tool. The architecture design description is refined over 3 abstraction levels.

ASIP design description in the LISA ADL and the assembly code (ASM) define the functionality of the architecture through the instructions designed for the specific application. These ASMs are interpreted and tested for functional correctness through the use of macro assembler and linker. The generated executable *EXE* can be later used for simulation and debugging.

Once the correctness of the design description is ensured the design description in LISA is automatically translated to VHDL equivalent models by the Processor Designer along with memory layout and HDL simulation memory files. The Processor Designer tool also provides exe2txt script that automatically translates memory initialization files described in ASM to equivalent text (*.mmap) files that can be used by the VHDL files for VHDL memory initialization. This is the second level of design description which can be used along which HDL simulation tools like Modelsim for hardware simulation.

The third level of description is at the circuit level through the use of ASIC or FPGA tools for hardware realization. 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.

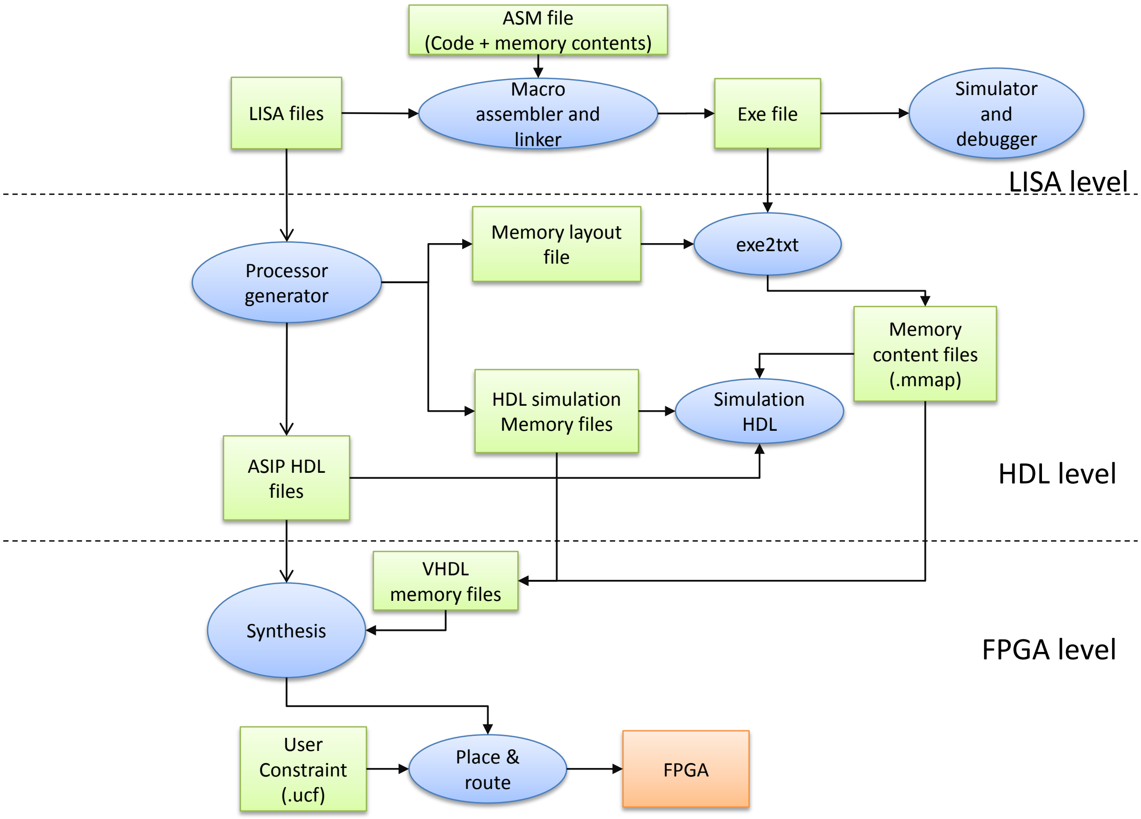


Figure 2.3 — LISA-based ASIP architecture design flow

2.2 State of the art in channel decoder design

This second gives an overview on state-of-the-art efforts in channel decoder design in order to clarify the position of the proposed contributions in this thesis work. A considerable amount of contributions have been proposed in this challenging domain over the past years targeting many different variants in terms of classes of error correction codes, algorithmic and architecture optimization techniques, design objectives, and design approaches. The proposed overview is far from being exhaustive, however a selection of recent works related to the thesis scope in terms of flexibility support of Turbo and LDPC decoding is presented. The section is organized in three parts summarizing recent related works in Turbo decoding architecture design, LDPC decoding architectures, and multi-code channel decoding architectures.

2.2.1 Turbo decoding architectures

Ever since the discovery of Turbo codes in 1993 [12], considerable amount of research works has been targeting practical VLSI implementations of Turbo decoders. Using mainly the low complexity sub-optimal Max-Log-MAP algorithm, many contributions have been proposed targeting diverse design objectives in terms of area efficiency, energy efficiency, flexibility, scalability and high throughput. Among the initial efforts in this context we can cite the examples of [44–47]. The work in [44] has investigated sub-block parallelism in order to increase the throughput. At the lower parallelism level of MAP computations, the work in [18] has explored several computational scheduling schemes. The butterfly schedule using two recursion units is shown to achieve the best results in terms of computational logic and decoding latency. Further, forward backward schedule using two recursion units each working on adjacent windows is shown to achieve the same decoding latency with slight addition of computational logic. In both cases, acquisition for backward recursion and message passing for forward recursion are used. However, the presented analysis does not consider the extrinsic memory access conflicts that might arise due to the interleaving rules imposed by the standard. Quantization issue have been studied in [45] which reported the use of 6 bit and 8 bits for input and extrinsic LLRs respectively with acceptable degradation of 0.1 dB w.r.t. the floating point reference model. They further report the use of 10 bit quantization for state metric LLRs. Authors of [48] present an overview of the implementation aspects related to Turbo decoding architectures discussing the issues of LLR quantization and iteration stopping criteria. Several joint algorithmic/architecture optimization techniques have been investigated and proposed in [49]. One of the first ASIC implementation was presented in [50] achieving a throughput of 50 Mbps with 10 decoding iterations and an operating frequency of 1 GHz. On the other hand, few works have investigated new Turbo decoding architectures based on other decoding algorithms like SOVA [46, 51].

Turbo codes have been since widely adopted in wireless communication standards like CDMA2000, 3GPP, LTE, WiMAX, DVB-RCS, etc. Many implementations have succeeded to meet the low throughput requirements of the early standards (e.g. CDMA2000 and 3GPP) using advanced DSP architectures [52–54] or customizable processors [55]. However, the scalability of such implementations are limited by the block interleavers specified in these standards which cause memory access contentions when targeting higher sub-block parallelism degree. The introduction of contention-free interleavers, like ARP in WiMAX and QPP in LTE, alleviated this limitation enabling high throughput implementations [56–61]. The work presented in [56], targeting LTE, allows multiple SISO decoders (1, 2, 4, or 8) to concurrently process frame sub-blocks and integrates a three stage network to connect the multiple memory and SISO decoder modules. Implemented in 90nm CMOS technology, the design achieves a throughput of 129 Mbps with 8 iterations and occupies an area of 2.1 mm^2 while exhibiting a power consumption of 219 mW and supporting the maximum specified frame size of 6144 bits. The dedicated architecture proposed in [61] for SBTC specified in LTE achieves high throughput of 150 Mbps while using the acquisition method for boundary state metric initialization along with 11 bits state metric quantization. Targeting Gbps throughputs, a recent work [62] has proposed an LTE compliant Turbo decoder architecture with 32 parallel SISO decoders using Radix-4 trellis compression and butterfly schedule of forward backward calculations. A throughput of 2.15 Gbps is achieved with an on chip area of 7.1 mm^2 using 65nm CMOS technology.

Other works have proposed the additional support of DBTC specified in WiMAX standard. As an example, the work in [60] presents an architecture which supports all DBTC parameters specified in WiMAX and 18 frame sizes of the 188 specified in LTE. A high area efficiency is achieved by supporting only those frame sizes with interleaving properties that can be easily mapped to the extrinsic exchange paths of DBTC. Another example is the parameterized architecture of [59] which supports both Turbo modes (DBTC and SBTC) and achieves a high throughput of 187 Mbps with 8 parallel MAP decoders. It exploits sub-block parallelism along with radix-4 trellis compression scheme for efficient support of SBTC.

Furthermore, ASIP design approaches have been explored in this application context. In [63] a flexible and high performance ASIP model for Turbo decoding was proposed which can be configured to support all simple and double binary Turbo codes up to eight states. The architecture uses shuffled decoding with frame sub-blocking. The extrinsic information are iteratively and concurrently exchanged between multiple component decoders via an on-chip communication network presented in [64]. Since Turbo code interleaving rule varies from one standard to another and/or one mode to another, NoC based solutions are investigated [64] for parallel Turbo decoding. The flexibility of these on-chip communication networks enables their use for both DBTC and SBTC standards. To that purpose, several application specific on-chip communication networks were recently introduced based on various topologies like de-Bruijn, Butterfly, 2D Mesh, chordal ring [65], and Benes [66],.

2.2.2 LDPC decoding architectures

Since the renewed interest in LDPC codes by 1997, considerable number of LDPC decoder architectures have been proposed targeting different design objectives and/or using different design approaches [27, 67, 68]. The first efforts in this context were mainly aiming to reduce the implementation complexity by defining new classes of LDPC codes jointly with their efficient hardware decoding architectures. Examples of such efforts can be found in [69–72] which were mainly implemented on FPGA.

Some early results have succeeded to demonstrate high throughput ASIC implementation for specific code structure and parameters. The work in [67] achieves 1 Gbps throughput for 1024-bit frame size and 1/2 code rate by wiring the whole Tanner graph into hardware. In [73], a code-programmable and code-rate tunable LDPC decoder is proposed, but the code length is still fixed to 2048 bits for simple VLSI implementation. On the other hand, some works [70, 72, 74, 75] have considered partly parallel decoder architecture to achieve high throughput targeting special class of LDPC codes for implementation realized on FPGA. [69] one of the first FPGA implementation of LDPC and further shows its implementation complexity to be comparable to that of Turbo decoding.

Later designs included flexibility as a design objective in order to support different code rates, frame sizes, check matrices, and parallelism degrees for a class of LDPC codes. The work in [70] proposes and new class of LDPC codes (namely Hardware-Constrained LDPC) and demonstrates a low complexity flexible FPGA-based implementation. In [76], a LDPC decoder that supports three block sizes and four code rates is designed by storing 12 different parity check matrices on-chip. The flexible implementation solution given in [77] exploits commonalities between 2 specific types of LDPC codes (one derived from [23, 78] and the second from [79]) and derives a decoding architecture capable of efficiently handling these codes.

These early efforts eventually resulted in the emergence of the class of QC-LDPC codes in many applications and communication standards such as: WiMAX, WiFi, DVB-S2, DVB-T2, etc. These codes allow for high parallelism degrees and an efficient implementation of the connectivity between variable and check nodes.

In this context, many channel decoder architectures have been proposed following the introduction of QC-LDPC codes in WiMAX by 2005 (IEEE 802.16e-2005) and in WiFi by 2009 (IEEE 802.11n). Both standards specify various code rates (ranging from 1/2 to 5/6) and frame sizes (ranging from 576 to 2304 bits) with a target throughput of 70 Mbps for WiMAX and 300 Mbps for WiFi.

Hence, some of the proposed architectures have targeted WiMAX standard, like the ones found in [80–83]. Other works have proposed dedicated architectures targeting the WiFi standard, like the ones found in [81, 84, 85]. [84] shows a scalable LDPC decoder for 802.11n WiFi standard suitable for portable devices shows that the proposed design is suitable for portable devices, with throughput ranging from 180 to 410 Mbps, and the power consumption being below 235 mW when implemented on 65 nm CMOS technology. The implementation in [85] decodes LDPC frames consuming low power (less than 435 mw) when implemented in 180 nm CMOS technology. The design utilize the column overlapping of the LDPC parity check matrix through which the amount of access for the memory storing the posterior values is minimized. In addition, a thresholding decoding scheme is proposed which reduces the memory access by trading off the error correcting performance. As both WiMAX and WiFi standards specify similar struction of structure of QC-LDPC codes, few other initiatives have proposed multi-standard LDPC decoder architectures supporting WiMAX and WiFi standardized LDPC codes and associated parameters, such as [86–89]. In [88] a reconfigurable message-passing network is proposed to facilitate message transportation in decoding multimode QC-LDPC codes. WiFi and WiMAX LDPC codes are supported by exploiting the shift-routing network features, to route the decoding messages in parallel to fully support those specific to 19 and 3 sub-matrix sizes defined in IEEE 802.16e and IEEE 802.11n applications with less hardware complexity. While in [89] a multi-mode message passing through an enhanced self-routing and the two-way duplicated switch network are presented.

Most of these architectures utilize a parametrized design approach and efficiently exploit the parallelism level allowed by the standard [81, 84, 88–90]. Each proposes new algorithmic and/or architecture techniques to meet specific design objectives in terms of throughput and multi-mode support with low area and/or low power consumption as mentioned before. For example: [81] utilizes the value-reuse property of offset min-sum, block-serial scheduling of computations and turbo decoding message passing algorithm to achieve low memory usage, reduction of routers, and increased throughput. [90] used two stopping criteria along with usage of belief propagation algorithm (through the use of lookup tables) in the design of low power high throughput architecture. The decoding will stop if the two conditions are satisfied: (1) the hard decisions for the information bits based on their LLR values do not change over two successive iterations, and (2) the minimum of the absolute values of the information bit LLRs is larger than a pre-defined threshold.

However, some recent works have also proposed the use of ASIP-based approach [91, 92]. [91] presents a decoder with a five-stage pipeline, 32-bit RISC processor and it can supports three different code rates (0.4, 0.6 and 0.8) by only modifying the program. [92] proposed a four pipeline stage decoder for a special class of LDPC codes (based on extended irregular repeat-accumulate codes) using Synopsys processor designer tool.

Finally, the QC-LDPC code structure and the allowed efficient partial parallelism lead generally for efficient memory organization and communication interconnect. Most of the proposed decoder designs use barrel shifters [80, 87, 90] to implement the underlined shift operations in routing the exchanged information between processing elements or to/from memories. In this context, targeting flexible multiprocessor LDPC decoder implementations, several works have investigated the use of application-specific Networks-on-Chips (NoC) [81,93–96]. For example, in [96], a de-Bruijn topology NoC is proposed to handle the communication between variable and check nodes for any LDPC code. The router embeds a modified shortest path routing algorithm that can be executed in 1 clock cycle, together with deadlock-free and buffer-reducing arbitration policies. In [81] on the other hand, uses benes-network for inter-processing element communication. In [95] analyzes further the possible architectural possibilities and choices in this context considering 2D, de-Bruijn, ZONOC and MDN networks.

Regarding wireless digital video broadcasting standards (DVB-S2/T2), the specified LDPC codes are characterized by two main differences with respect to those specified in WiMAX and WiFi: (1) very large frame sizes of 16400 bits and 64800 bits and (2) existence of double diagonal permutation matrices in the definition of check matrix. These parameters impact considerably the decoder architecture, memory requirements, and error rate performances. Several dedicated architectures have been proposed [97–99]. As an example, the architecture proposed in [97] processes 360 check nodes in parallel achieving a throughput of 91 Mbps (at 300 MHz for rate 1/2) and occupying an area of 11 mm^2 in 90nm CMOS technology. [99] solves the well known problem of superpositions of permutation matrices. The enhanced convergence speed of Gauss-Seidel decoding is used to reduce area and power consumption. Furthermore, propose a modified version of the lambda-Min algorithm which allows to further decrease the memory requirements of the decoder by compressing the extrinsic information.

2.2.3 Multi-code channel decoding architectures

Flexibility requirement of channel decoding architectures becomes more and more crucial when considering the emerging multi-mode and multi-standard applications, as well as the increasing interest for Software Defined Radio and Cognitive Radio concepts [100, 101]. Even for a single standard like WiMAX, several error correction codes (convolutional, Turbo, LDPC, and block Turbo) are specified as mandatory or optional. Hence, in the last few years, several multi-code architectures have been explored and proposed to support the decoding of two or more different classes of error correction codes (e.g. LDPC and Turbo decoding, Turbo and convolutional (Viterbi decoding), etc.). The main aim is to share the memory, logic and/or communication interconnects in order to achieve better efficiency in terms of area when compared to the direct assembly of dedicated individual decoders.

In this context, few initiatives have investigated the combination of Turbo and Viterbi decoding. Authors of [102] and [103] have proposed a unified architecture designed for UMTS base stations. A dual mode Viterbi/Turbo decoder, sharing path metric calculation and extrinsic information memories, is proposed. A trellis processor used to update path metrics in both supported decoding algorithms. A 2 Mbps throughput at 88 MHz of clock frequency is demonstrated when performing 10 Turbo decoding iterations. In [104], another combined architecture is suggested for wireless terminals. In this architecture the data-path and the memories are shared. A Max-Log-MAP algorithm is used for decoding both convolutional and Turbo codes. However, this is only possible when the throughput requirement for convolutional codes (e.g. 12.2 kbps) is much lower than that of Turbo codes (e.g. 384 kbps). In another effort to combine the two types of decoders, soft Viterbi decoding is used for Turbo decoding and hard output Viterbi decoding is used for convolutional codes [105].

Similarly, unified decoder architectures for LDPC and Turbo codes has been presented in [106–110]. Multi-code decoding is achieved in [106] by employing flexible add-compare-select (FACS) units. By representing LDPC codes as parallel concatenated Single Parity Check (SPC) codes, the authors have efficiently reused the Turbo decoding hardware resources for LDPC decoding functions. The architecture supports decoding of SBTC codes of LTE and LDPC codes of WiFi and WiMAX. When implemented in 90nm CMOS technology, the work reports a maximum throughput of 450 Mbps for SBTC decoding and 600 Mbps for LDPC decoding while occupying a total area of 3.2 mm^2. Similar architecture is presented in [110] to share logic and memory resources with additional decoding support of Turbo codes specified in 3GPP, DVB-SH, and WiMAX standards. The entire design is implemented in 45nm CMOS technology occupying an area of 0.9 mm^2 and clocked at 150 MHz to achieve low power and yet meeting the target throughput. However, studies presented in [111] conclude that such data-path sharing for LDPC and Turbo decoding has little benefits and only for special configurations which have similar memory requirements between the decoding modes (LDPC / Turbo). It further mentions that even in such cases, sharing memory is much more attractive than sharing computational hardware. In fact, the best match for a combined LDPC/Turbo data path can be achieved when both have the same granularity, e.g. at the check-node and log-butterfly operator level [111]. However the size of these data-paths is so small that the configuration logic will have comparable area and thus lead to ineffective reuse of data-path logic.

Besides the above mentioned multi-code decoder architectures which can be considered as parameterized cores, several initiatives have explored ASIP based design in this application context. As an example, the FlexiTreP ASIP presented in [112] supports trellis based channel codes (i.e. convolutional, SBTC and DBTC) for various standards. Decoding of QC-LDPC codes was later added to this architecture and presented in [113] as FlexiChap where memory sharing across Turbo and LDPC modes was explored. It was shown that only a slight increase in area occurs (from 0.31 mm^2 for FlexiTreP to 0.39mm^2 for FlexiChap). In [108, 109], the authors propose an ASIP architecture addressing in a unified way the Turbo and LDPC coding requirements of LTE, WiFi, WiMAX and DVB-S2/T2 with data-path and memory reuse across the different FEC families. Results illustrate how the obtained area was lower than the cumulated area of dedicated Turbo and LDPC solutions.

Finally, several NoC-based scalable multiprocessor architectures for combined LDPC/Turbo decoding were investigated. The authors in [114] propose an efficient algorithm which can compute a collision-free memory mapping of interleaving laws with no constraint imposed on the code itself and the target parallelism degree. Thus, a versatile implementation supporting LDPC/Turbo decoding requires for each supported code a pre-processing step to recompute the corresponding memory mapping. A flexible on-chip interconnection network is designed with the aim of fully exploiting the parallelism of the LDPC/Turbo decoder architecture by reducing the message latency, alleviating the memory conflicts and efficiently routing any permutation from the network input ports to its output ports. Another example is the work presented in [115] which proposes a NoC based multiprocessor architecture, based on generalized Kautz topology [116], supporting LDPC and Turbo decoding and achieving throughputs of 93 Mbps and 72 Mbps respectively.

Overall, many contributions are emerging rapidly in this domain, seeking to increase the flexibility support and to improve the resulting architecture efficiency in terms of performance/area [117]. This thesis work belongs to these last efforts targeting multi-code channel decoding architectures. The main objective is to investigate the maximum achievable architecture efficiency when adopting the rapid design methodology and well established tools related to ASIP design concept associated with the high scalability provided by multiprocessor architecture models. By considering mainly the challenging Turbo and LDPC decoding applications, new multi-ASIP channel decoder architectures are proposed targeting high flexibility combined with high Architecture Efficiency. Different architecture alternatives and design approaches are explored in this context.

2.3 Initial ASIP architecture for flexible Turbo decoding

One of the recent related state-of-the-art efforts has been initiated few years ago at the Electronic department of Telecom Bretagne towards the design of flexible Turbo decoding architecture in the context of a previous thesis study [16]. In this initial effort, the main target was to explore the effectiveness of the newly proposed ASIP-design tools in terms of quality of the generated HDL code and flexibility limitations when targeting this class of applications. To that end, the target flexibility was set very high to investigate the support of any convolutional code trellis of Turbo codes. Furthermore, besides the non support of LDPC decoding, this initial ASIP architecture were missing several features which will be addressed in the next chapter.

In this section we give a brief summary of this initial reference ASIP architecture (namely TurbASIP) for flexible Turbo decoding. One of the main feature of this architecture is the investigation and the exploitation of the various parallelism techniques available for Turbo decoding, particularly for DBTC.

2.3.1 Overview of the TurbASIP architecture

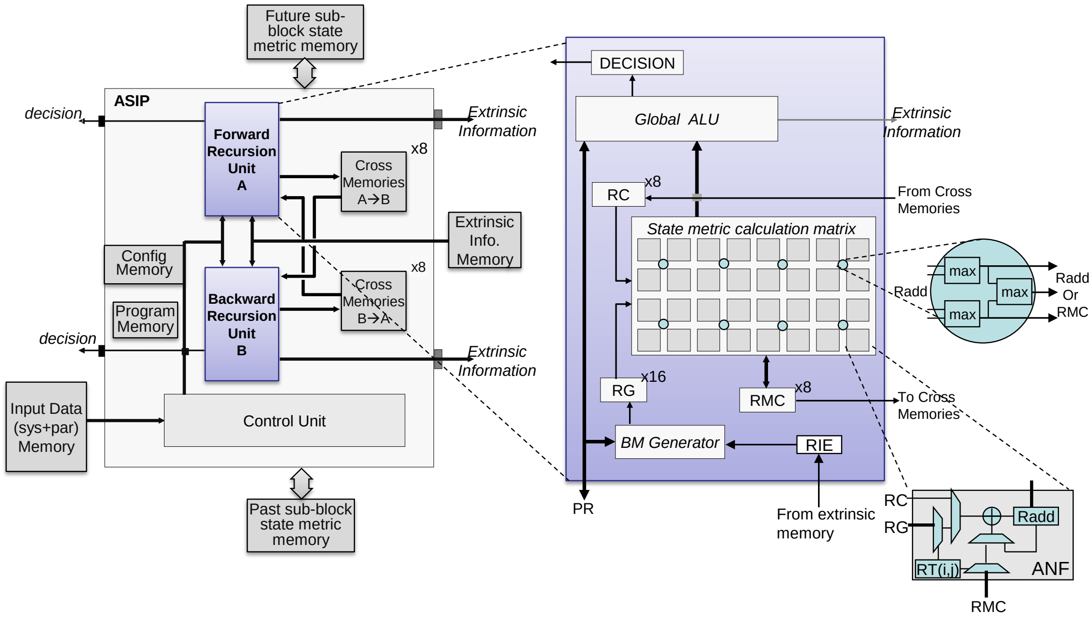


Figure 2.4 — TurbASIP architecture

Figure 2.4 presents the block diagram of the initial TurbASIP architecture. Most of the parallelism techniques presented in Sub-section 1.3.4 (page 20) are exploited (except trellis compression). At the branch metric level, the ASIP computes in parallel all possible combinations of branch metric LLRs (γ) and store them to 16 *RG* registers. The branch metric LLRs are computed from the channel LLR values and extrinsic values stored in the input memories and extrinsic memory banks. Adding to it, butterfly schedule (described in 1.3.4.1, page 21) is adopted by the use of two recursion units to process the window in parallel in the forward and backward directions. As the flexibility target was limited to the support of a maximum of 8 states double binary Turbo codes, each of the 8 states in the trellis has 4 branches. Therefore, and in order to fully exploit the efficient parallelism of trellis transition, each recursion unit integrates 32 state metric calculation units that compute the possible 32 next state metric LLRs and store them in the *RADD* registers, using the trellis description information which are written in the *RT* registers (refer to the architecture of the ANF unit of Figure 2.4). The trellis description bits are read from the external trellis configuration registers of depth 4 and width 32 bits. Each register contains the trellis branch description for sets the 8 *RT* registers. The muxes connected to the *RADD* registers enable the circuit to calculate state metrics (α or β) by *RG+RMC* operation. Alternatively, it also enables to calculate $\alpha + \beta + \gamma$ required for aposteriori LLR calculation by adding *RC* fetched from the cross memories and newly calculated *RADD*. The structure of the *max* units permits it to compute symbol aposteriori values (γ^{apos}) or state LLRs ($\alpha(s', s)$ or $\beta(s', s)$) by taking maximum row-wise or column wise respectively. The 8 state LLRs calculated by the *max* units are stored in *RMC* registers. The butterfly scheme needs storage of intermediate state metric LLRs until the half window boundary is reached. *Cross metric memories* A and B store the intermediate state metric generated from the recursion units. During the processing of the second half of the window, Cross metric memories are read to *RC* registers. *RMC* registers store the previous $\alpha(s', s)$ (for recursion unit A) or $\beta(s', s)$ (for recursion unit B) state values fetched from the cross memories. The calculated aposteriori LLR values γ^{apos} or the extrinsic information γ^{ext} are exchanged via *Extrinsic information* ports. The hard decisions produced by the recursion units are provided on the 2 *Decisions* ports of width $WindowLength/2$.

Furthermore, at the SISO decoder level, multiple SISO decoders are supported by the use of sub-blocking with message passing for boundary state initialization as shown in Figure 2.5. To increase the use of parallelism degree, shuffled decoding is used by employing multiple SISO decoders grouped into natural and interleaved domain processing the frame in natural order (i.e. first half iteration done by component decoder 0) and interleaved order (i.e. second half iteration done by component decoder 1) respectively. Finally, the extrinsic information generated are sent to the other domain through the butterfly NoC interconnection.

2.3.2 TurbASIP pipeline

TurbASIP is modelled as a processor with 8 pipeline stages as shown in Figure 2.6. The first 3 stages of the pipeline correspond to instruction address generation, instruction fetch and instruction decode. The branch metrics are calculated in two stages: *BM1* calculates the symbol LLRs $\gamma^{sys}, \gamma^{par}$ along with the scaled γ^{ext} with the scaling factor $S_c = 0.875$. *BM2* calculates the intrinsic LLRs γ^{intr} and branch metric LLR $\gamma(s', s)$. The *EX* stage contains the adders required for the calculation of the 32 state metric LLRs as defined by the equations (1.31) and (1.32) (in page 19). The *MAX1* stage contains the reconfigurable max operators and calculates the α and β state metric LLRs by taking the maximum of the *RADD* registers column wise (Figure 2.7a). During extrinsic generation phase, the max units of *MAX1* are reconfigured to calculate the maximum of the four *RADD* registers along the horizontal direction (Figure 2.7b). Thus, 2 partial aposteriori LLR values are generated per row. The final aposteriori LLRs are generated by the max units of the *MAX2* pipeline stage that calculate the maximum of the eight partial aposteriori LLRs, to produce the four aposteriori symbol LLRs γ^{apos}. The aposteriori LLRs thus generated

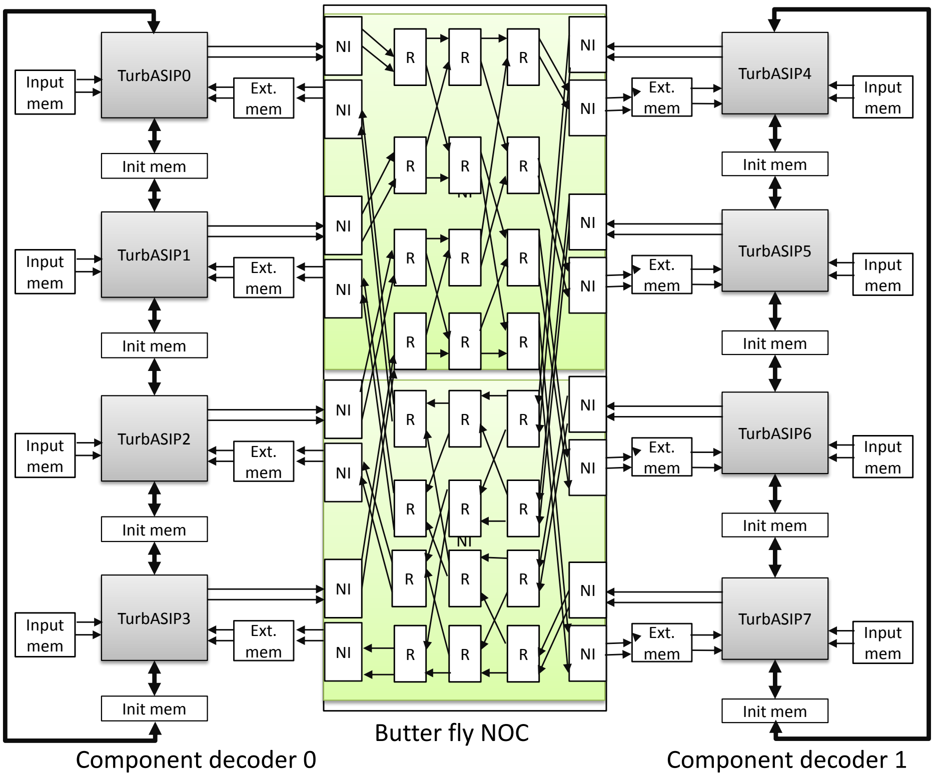


Figure 2.5 — Multi-TurbASIP architecture

can contain overflow errors, which are detected and corrected in the *MAX2* pipeline stage (as it is explained in the next sub-section). The last stage of the pipeline (*ST*) generates the extrinsic information from aposteriori LLRs and the intrinsic LLRs γ^{intr} as given by the equation (1.36) (in page 20).

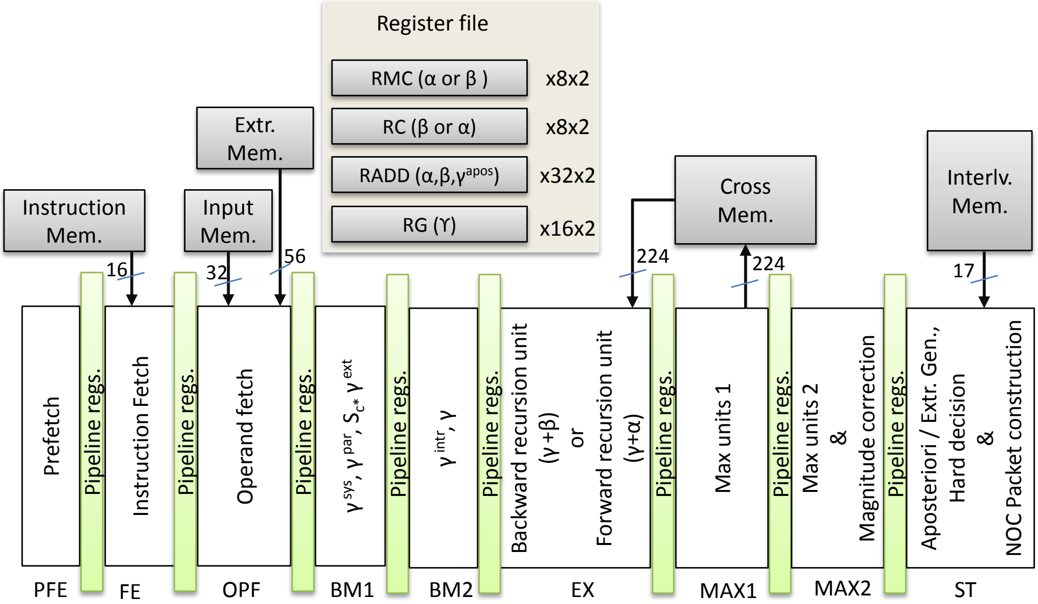
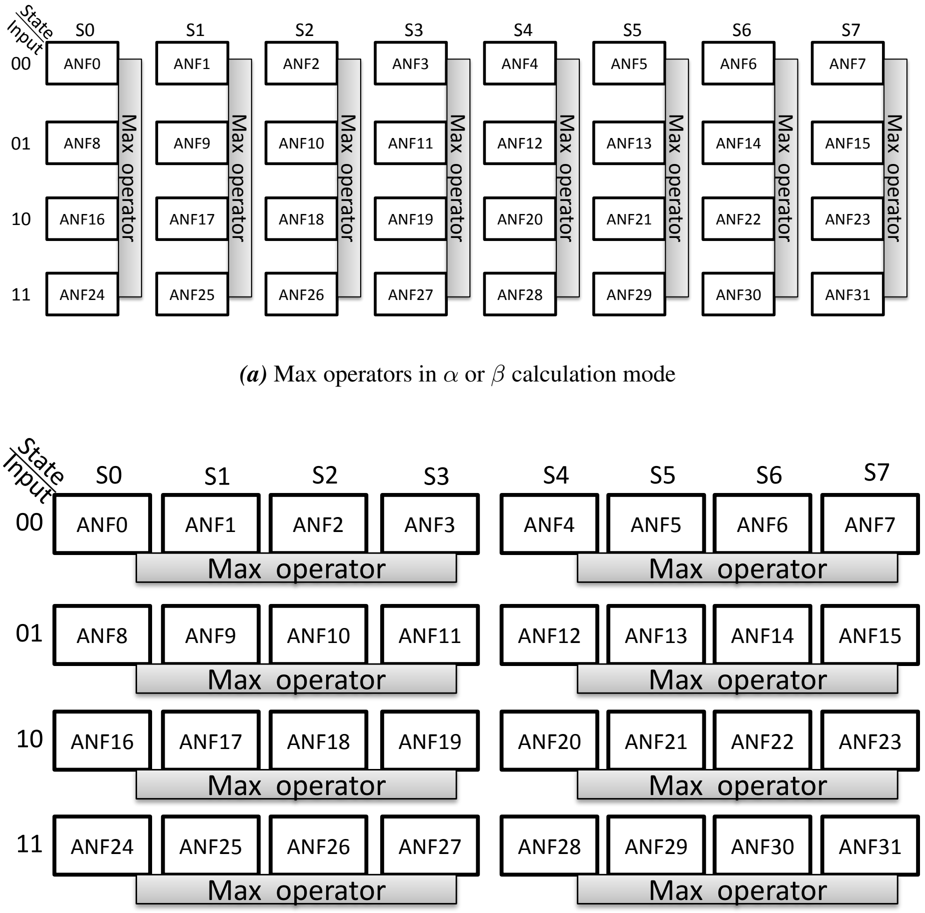


Figure 2.6 — Overview of the TurbASIP pipeline stages along with its register file and memory banks


State
Input
S0 S1 S2 S3 S4 S5 S6 S7
00 ANF0 ANF1 ANF2 ANF3 ANF4 ANF5 ANF6 ANF7
01 ANF8 ANF9 ANF10 ANF11 ANF12 ANF13 ANF14 ANF15
10 ANF16 ANF17 ANF18 ANF19 ANF20 ANF21 ANF22 ANF23
11 ANF24 ANF25 ANF26 ANF27 ANF28 ANF29 ANF30 ANF31
Max operator

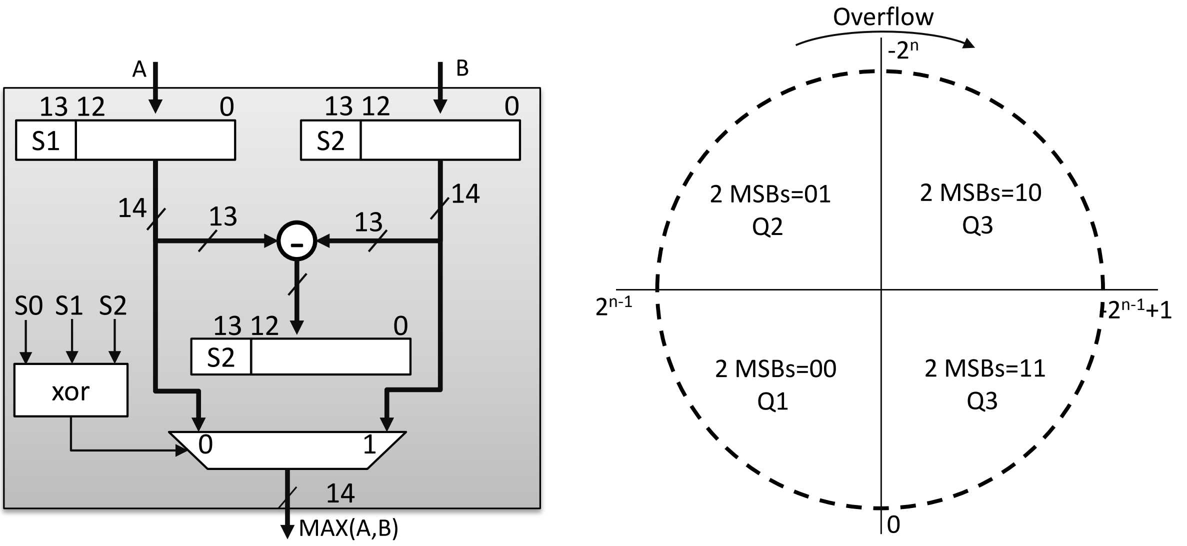
(a) Max operators in α or β calculation mode

(b) Max operators in aposteriori calculation mode

Figure 2.7 — Four input max operator modes

2.3.3 Max and modulo operators

While performing the addition operation involved in extrinsic information computation, the already stored sum (in RADD REG) of state metric and branch metric ($\alpha + \gamma$ or $\beta + \gamma$) is added with the other state metric (β or α respectively). Thus, the accumulated metrics during the backward and forward recursions pose an issue when using a fixed quantization in a hardware implementation. To avoid this issue, most of existing implementations adopt a rescaling scheme which often consists of subtracting the minimum metric from all metrics. A more efficient technique to avoid this issue, which has been adopted in the proposed ASIP architecture, is to apply the modulo operator [118] which corresponds to the overflow mechanism in two's complement arithmetic and therefore has no hardware cost. When overflow occurs, causing metric values to enter from positive region to negative region, the largest value becomes the smallest. In order to handle this case, specialized MAX operators are designed (Figure 2.8a) to detect this condition and produce the correct MAX value. Another issue is when hard decisions are taken on aposteriori LLR informations or extrinsic LLRs are sent to the other component decoder, as that component decoder will not be able to differentiate the correct maximum extrinsic. This issue is resolved by correcting the overflow errors as shown in Figure 2.8b, in which n bits represent the quantization of state metrics or extrinsic information. If an extrinsic information lies in Q-2, its two MSB's will be "01" whereas in Q-3 they will be "10". Hence, if some of the extrinsic



(a) The Max Operator

(b) Overflow condition during aposteriori/extrinsic calculation

Figure 2.8 — Max operator unit and over flow condition

informations related to different combinations of a symbol lay in Q-2 and 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 incrementing the two MSB bits of all the extrinsic informations of the symbol.

2.3.4 TurbASIP: sample assembly code

TurbASIP is programmed through an assembly code. The instructions of the assembly code are designed to specifically do specialized operations required by the Turbo decoding algorithm. The program memory can be split into 3 code sections. The first section (Listing 2.1) corresponds to the assembly code of the first iteration of the turbo decoding which starts by initializing the ASIP configuration registers setting the mode by reading the trellis configuration registers (instruction *SET_CONF*). Next, the size in symbols of half of the window (SET_SIZE), the scaling factor (SET_SF), number of windows (SET_WINDOW_N) and the initial window counter (SET_WINDOW_ID). The value 6 used by the *SET_SF* command is to scale the input extrinsic LLRs by 0.875 before computing the branch metrics. Since, WiMAX Turbo code is a circular code, the initial window boundary state registers of the first window (RMC) are initialized to be uniform i.e. equi-probable. *ZOLB* i.e. zero overhead loop instruction, uses branch prediction in looping. With this single instruction, lines from (@26) and (@27) of code are executed 32 times i.e. half of window size (as set by the SET_SIZE instruction). The instruction at line (@26), *DATA LEFT WITHOUT_EXT ADD M COLUMN* implement the left side of butterfly decoding scheme. During the first iteration of the shuffled decoding schedule the extrinsic memories are uninitialized, hence the *WITHOUT_EXT* field of the instruction specifies not to use extrinsic information in the branch metric calculations. The *ADD M* field of the instruction force the *ANF* unit to do 32 state metric calculations ($\alpha + \gamma$ or $\beta + \gamma$) in the *RADD* registers in the *EX* stage of the pipeline. The *COLUMN* field enforces MAX units to do max operation column wise, storing the resulting state metric LLRs (α or β) in the *RMC* registers in the *MAX1* stage of the pipeline. As *RADD* and *RMC* are updated in adjacent pipeline stages, this creates data dependency in the calculation of *RADD* register values. Hence, an idle cycle is introduced via *NOP* instruction at line (@27).

Once the left side of butterfly decoding schedule is complete for a window, the right side of the butterfly schedule is processed by executing the instructions at lines (@30) and (@33) 32 times. *DATA RIGHT WITHOUT_EXT ADD M COLUMN* calculates the state metric LLRs similar to the previous *DATA* instruction. While the subsequent instruction, *EXT ADD i LINE* calculates the extrinsic information, wherein the *ADD i* configures the multiplexers in the *ANF* unit of the *EX* pipeline stage to calculate the sum of *RC* (α or β) and *RADD* (which contains $\beta + \gamma$ or $\alpha + \gamma$). The field *LINE* enforces the max operators in the *MAX1* pipeline stage to calculate the maximum row-wise. Additionally, this instruction activates *MAX2* and *ST* pipeline stages to calculate the γ^{apos} and γ^{ext} informations. Furthermore, the ST stage is activated to fetch the corresponding interleaved/de-interleaved address for NoC packetization.

If the current processed window is not the last window of the sub-block, the *EXC_WINDOW* instruction handles the boundary state metrics initialization of the *RMC* registers (inside the ASIP). If not, this instruction initializes the next logical window processed by neighboring ASIPs (neighboring sub-blocks). In both cases, the window counter *WINDOW_ID* is incremented. The instruction *REPEAT UNTIL ptr y times* at line (@18) executes *y+1* times the instructions from (@current line+2, i.e. line @22) to the flag pointer *ptr* at line (@38).

The second section of the assembly code (shown in Listing 2.2) corresponds to the other iterations of the Turbo decoding process, except the last iteration (shown in Listing 2.3). Thus, the second section is similar to the one presented above, except that the *DATA* instructions utilize (and scale) extrinsic LLRs during branch metric calculation. *REPEAT UNTIL ..* instruction is reused here in nested way to realize the execution of all the windows for num_iter times by the use of *PUSH* and *POP* instructions at lines (@46) and (@60) respectively. These instructions store and retrieve the current loop counter value (of the outer *REPEAT UNTIL...* instruction) in an stack of depth 1.

The final section of the assembly code is presented in Listing 2.3 and corresponds to the execution of the last iteration. *HARD WITH_EXT ADD i LINE* is similar to the *EXT WITH_EXT ADD i LINE* instruction, however instead of calculating γ^{ext} it calculates the hard decision bits from the aposteriori LLRs. The hard decision bits are accumulated in an internal register while the instruction *ST_DEC* outputs the final hard decision.

Listing 2.1 — TurbASIP assembly code for the first iteration

```
1  .text
2  ;set configuration wimax
3     SET_CONF 0
4     SET_CONF 1
5     SET_CONF 2
6     SET_CONF 3
7  ;set half window size
8     SET_SIZE 32
9  ; set the scale factor 6=0.875
10    SET_SF 6
11 ;set number of windows and initial
12 ;window id counter to zero
13    SET_WINDOW_N 2
14    SET_WINDOW_ID 0
15 ;set boundary initialization of
16 ;RMC registers as uniform
```

```
  SET_RMC UNIFORM, UNIFORM
  REPEAT UNTIL _loop0 1 times
  NOP
;repeat instructions between _RW1 to_CW1
;and between _CW1 to_LW1 SET_SIZE times
  ZOLB _RW1,_CW1,_LW1
  W_LD_BETA 0
;configure max units to take max column wise.
;store results in RMC
_RW1: DATA LEFT WITHOUT_EXT ADD M COLUMN
_CW1: NOP
;configure max units to take max column wise.
;store results in RMC
  DATA RIGHT WITHOUT_EXT ADD M COLUMN
;configure max units to take max rowwise.
;to calculate extrinsic
_LW1: EXT WITHOUT_EXT ADD i LINE
;increment the window number
  EXC_WINDOW
  NOP
  NOP
_loop0: NOP
```

Listing 2.2 — TurbASIP assembly code for the middle iterations (e.g. iterations 2 to 6)

```
;start the second part of the turbo decoding
;iterations 2 to 6
  NOP
  set_window_id 0
  NOP
  REPEAT UNTIL _loop1 4 TIMES
  NOP
  PUSH
  NOP
  REPEAT UNTIL _loop2 1 TIMES
  NOP
  ZOLB _RW0,_CW0,_LW0
  W_LD_BETA 1
_RW0: DATA LEFT READ_EXT ADD m COLUMN
_CW0: NOP
  DATA RIGHT READ_EXT ADD m COLUMN
_LW0: EXT WITH_EXT ADD i LINE
  EXC_WINDOW
  NOP
  NOP
_loop2: NOP
  POP
  NOP
  SET_WINDOW_ID 0
```

```
_loop1: NOP
```

***Listing 2.3* —** TurbASIP assembly code of the last iteration

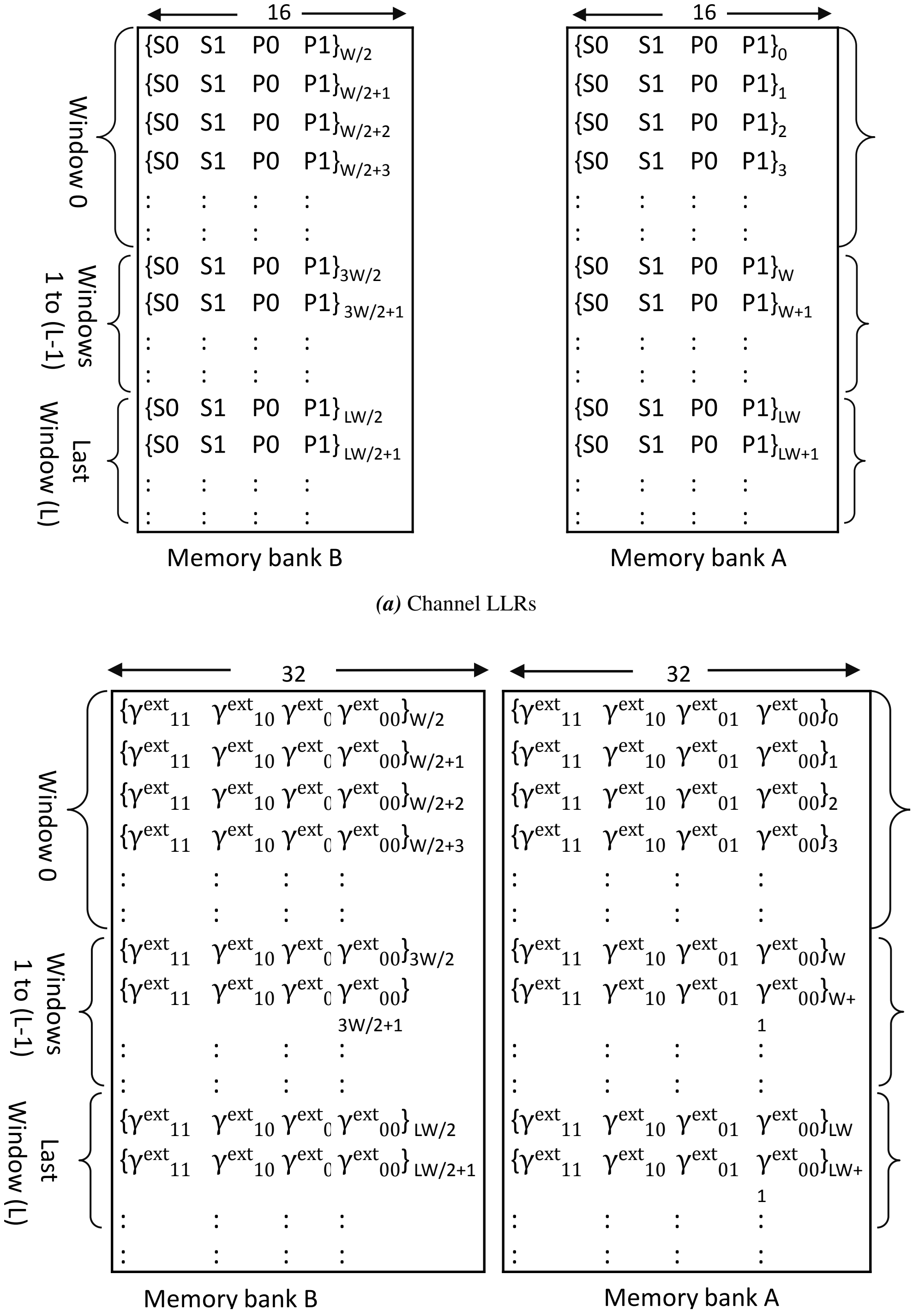
```
;start the last iteration giving
;out hard decisions
  REPEAT UNTIL _loop3 1 TIMES
  NOP
  ZOLB _RW,_CW,_LW
  W_LD_BETA 1
_RW: DATA LEFT READ_EXT ADD m COLUMN
_CW: NOP
  DATA RIGHT READ_EXT ADD m COLUMN
_LW: HARD WITH_EXT ADD i LINE
  EXC_WINDOW
  NOP
  NOP
  ST_DEC
_loop3: NOP
  NOP
  NOP
  NOP
```

2.3.5 Memory partitions

The input memories are partitioned into two banks each of width 16 bits. Each location of the memory stores systematic and parity bits adjacent to each other. Considering W is the number of symbols in a window (i.e. window length) and L the number of windows in the sub-block, the LLRs corresponding to the first half of the window is stored in the memory bank A and the second half is stored in the memory bank B (Figure 2.9a). Similarly, the four symbol extrinsic LLRs arriving through the NoC are also stored in similar order as in Figure 2.9b. The widths and the depths of the other memory banks in the architecture for TurbASIP in 4x4 configuration is given in Table 2.1. The instructions are of 16 bits width. The input and extrinsic LLRs are quantized to 4 bits and 8 bits respectively. Since 8 RMC (each quantized to 8 bits) values are stored during the processing of the left butterfly. The width of the cross memories is 80 bits. The interleave / de-interleave memories require 13 bits wide memories as the maximum frame size is 6144 bits.

2.3.6 ASIC synthesis results

The proposed ASIP was described in LISA and was translated into VHDL using the Processor Designer tool. ASIC synthesis targeting 65nm general purpose CMOS technology has resulted in a total TurbASIP area of 0.19 mm^2 (logic and memories) with a clock frequency of 500 MHz. The area utilization of each pipeline stage is as shown in Table 2.2. All the registers used in the design are jointly represented as Register file. The TurbASIP was initially designed to be a highly flexible architecture supporting run-time or design time reconfigurability. While trellis descriptions can be changed at run-time, the bit width or quantization of LLR represented needs


16
16
Window 0
Windows 1 to (L-1)
Last Window (L)
{S0 S1 P0 P1}W/2
{S0 S1 P0 P1}W/2+1
{S0 S1 P0 P1}W/2+2
{S0 S1 P0 P1}W/2+3
{S0 S1 P0 P1}3W/2
{S0 S1 P0 P1}3W/2+1
{S0 S1 P0 P1}LW/2
{S0 S1 P0 P1}LW/2+1
{S0 S1 P0 P1}0
{S0 S1 P0 P1}1
{S0 S1 P0 P1}2
{S0 S1 P0 P1}3
{S0 S1 P0 P1}W
{S0 S1 P0 P1}W+1
{S0 S1 P0 P1}LW
{S0 S1 P0 P1}LW+1
Memory bank B
Memory bank A
32
32
{γext11 γext10 γext0 γext00}W/2
{γext11 γext10 γext0 γext00}W/2+1
{γext11 γext10 γext0 γext00}W/2+2
{γext11 γext10 γext0 γext00}W/2+3
{γext11 γext10 γext0 γext00}3W/2
{γext11 γext10 γext0 γext00}3W/2+1
{γext11 γext10 γext0 γext00}LW/2
{γext11 γext10 γext0 γext00}LW/2+1
{γext11 γext10 γext01 γext00}0
{γext11 γext10 γext01 γext00}1
{γext11 γext10 γext01 γext00}2
{γext11 γext10 γext01 γext00}3
{γext11 γext10 γext01 γext00}W
{γext11 γext10 γext01 γext00}W+1
{γext11 γext10 γext01 γext00}LW
{γext11 γext10 γext01 γext00}LW+1
Memory bank B
Memory bank A

(a) Channel LLRs

(b) Extrinsic LLRs

Figure 2.9 — TurbASIP: LLRs storage in memory banks

to be fixed at design time. As TurbASIP serves as the base design of the thesis, we present the architecture synthesis results with its original quantization of 4 and 8 bits input and extrinsic LLR representation respectively. The total area of the 4x4 TurbASIP system decoder is 1.53 mm^2 (Table 2.2).

Memory	Number of banks	Depth	Width
Input	2	768	16
Extrinsic	2	768	32
Cross	2	32	80
Interleave/deinterleave	1	768	13
Trellis description Regs.	1	4	32
State boundary	1	12	80
Instruction memory	1	16	64

Table 2.1 — Memory bank partitions for a single TurbASIP in 4x4 mode

Design unit	Area in um^2
PF	613
FE	261
DC	1481
OPF	804
BM1	1741
BM2	2272
EX	9974
MAX1	7200
MAX2	842
ST	2205
Register File	34380
Memory Interface	4586
Total TurbASIP logic	70426
Total TurbASIP memories	120308
Butterfly NOC	18980
Total 4x4 TurbASIP system decoder	1535368

Table 2.2 — ASIC synthesis results for complete 4x4 TurbASIP system using 65nm general purpose CMOS technology (worst case 0.9v, 125C)

On average, in order to process two double binary symbols, TurbASIP needs 2 instructions for left butterfly (lines @52 and @53 of Listing 2.2) and 2 instructions for right butterfly (lines @54 and @55 of Listing 2.2) per iteration. This means that TurbASIP needs $N_{instr} = 4$ instructions (thus 4 clock cycles) to process 4 bits (2 symbols each composed of $Bits_{sym} = 2$ bits), per iteration. The throughput achieved is given by the expression:

$$\text{Throughput} = \frac{2 \times Bits_{sym} \times f_{clk} \times N_A/2}{N_{instr} \times N_{iter}} \tag{2.1}$$

where N_A= Number of ASIPs.

Considering a 4x4 TurbASIP system decoder, and using the above expression where N_A=8, f_{clk}=500MHz, N_{iter}=6, the achieved throughput is around 333 Mbps.

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. Furthermore, an overview on state-of-the-art efforts in channel decoder design was addressed. The proposed overview presents a selection of recent works related to the thesis scope in terms of flexibility support of Turbo and LDPC decoding in order to clarify the position of the proposed contributions in this thesis.

The chapter has also presented the architecture of an initial ASIP for flexible Turbo decoding. This ASIP has been developed in a previous thesis study at the Electronic department of Telecom Bretagne. In this initial architecture, the main target was to explore the effectiveness of the newly proposed ASIP-design tools in terms of quality of the generated HDL code and flexibility limitations when targeting this class of applications. To that end, the target flexibility was set very high to investigate the support of any convolutional code trellis of Turbo codes. Although not supporting LDPC decoding, this architecture has investigated the exploitation of the various parallelism techniques available for Turbo decoding, particularly for DBTC. This initial effort constitutes the starting point of this thesis work.

CHAPTER

3 DecASIP: Flexible Turbo/LDPC Decoder

THIS chapter presents our contributions in the design of flexible and optimized channel decoder supporting Turbo and LDPC codes. Starting with the initial TurbASIP architecture presented in the previous chapter, several design goals were specified for this work, which were: (1) efficient resource sharing between the LDPC and Turbo decoding modes, (2) scalability to support current and future high throughput requirements, (3) new LDPC decoding schedule adapted to the base TurbASIP architecture, (4) exploring possible parallelism techniques for efficient decoding of SBTC, DBTC, and LDPC codes, and (5) quick reconfigurability between the different supported decoding modes. In order to be relevant to the industrial needs we limited the design flexibility supporting only LDPC and Turbo codes specified in WiFi, WiMAX, and LTE. Furthermore, this also enables to compare with existing state-of-the-art implementations.

Towards fulfilling these objectives, an ASIP-based multiprocessor architecture is proposed and designed in two steps. In the first step we designed a novel ASIP architecture (DecASIP$_{v1}$) and developed an 8 DecASIP system decoder, efficiently mapping the target standards. In the second step (DecASIP$_{v2}$) mainly enhanced the throughput in LDPC mode by increasing the supported parallism degree. Additionally, a modified LDPC scheduling was proposed to support 4-DecASIP or 2-DecASIP decoder architectures.

The first section of this chapter presents the design motivations and architectural choices made for the *DecASIP*$_{v1}$. A detailed analysis of the quantization impacts alongs with the performance evaluation vis a vis reference models for different implemented modes (SBTC (LTE), DBTC (WiMAX) and LDPC (WiFi, WiMAX) modes) are also presented. The second and third sections present the two design phases of the proposed DecASIP channel decoder.

3.1 Design motivations

Towards the design goals stated above, and considering the state-of-the-art analysis presented in the previous chapter together with the available TurbASIP architecture, few general design decisions have been made. Regarding the target architecture model, we adopt multi-ASIP NoC based approach for rapid design cycle and design scalability. Regarding resource sharing for the target Turbo/LDPC channel decoder, and taking into consideration the dominant memory requirement along with the availability of low computational complexity decoding algorithms, we choose to investigate efficient memory and communication interconnect sharing while using Max-Log-MAP for Turbo decoding and NMS for LDPC decoding.

As the target flexibility is chosen to be limited to the support of LDPC and Turbo codes specified in WiFi, WiMAX, and LTE standards, the corresponding communication modes and throughput requirements (Table 1 in page 1) are considered.

In addition to these general design choices, below a summary of other specific design choices related to LDPC and Turbo modes for the target proposal of flexible ASIP channel decoder, namely DecASIP.

LDPC mode:

- With the general design choice of investigating multi-ASIP architecture model and scalability, and the fact that all LDPC codes specified in the target WiFi and WiMAX standards have 24 variable node groups (columns of the H_{base}), we chose to target an ASIP design processing 3 variable node groups and 3 check nodes. The aim is to design a reasonable scalable decoder that avoids small memory fragments and yet be capable of processing the entire input frame with reasonable parallelism degree. In this regard, the parallelism degree can be scaled using a multi-ASIP decoding architecture with a maximum of 8 ASIPs to process the whole 24 variable node groups in parallel.
- With the target scalable multi-ASIP decoder, we choose to investigate new computational scheduling which allows to keep channel LLRs localized to the ASIP and to move only the required data as messages across the multiple ASIPs. Furthermore, in this context, NoC based communication interconnect can be adopted.

Turbo mode:

- We chose to use the available TurbASIP architecture as a starting point for the target DecASIP in Turbo mode. Given the target maximum scalability of 8 ASIPs in LDPC mode, we chose to reduce the internal parallelism degree of TurbASIP by removing one recursion unit. This choice halves the throughput of TurbASIP, yet achieves the target throughput of 150Mbps in LTE mode with 8 TurbASIP (equation (2.1)).
- Low complexity *ARP* and *QPP* hardware interleaving generators are adopted in DecASIP rather than the memory-based interleavers used in TurbASIP.
- Radix-4 trellis compression technique is adopted in SBTC mode (LTE) for efficient hardware resource sharing with DBTC mode (WiMAX).
- As seen in the previous chapter, the program memory of the TurbASIP is composed of 3 sections: one for the first iteration, followed by an assembly code section for regular iterations and at the end a section for the last iteration and hard decision. This results in a large program memory which we target to avoid in DecASIP by making the loop branch decisions automatically based on internal counters.

3.1.1 Architecture Efficiency

The aim of this thesis work is to propose channel decoder architectures targeting high flexibility, but with a main focus on the architecture efficiency in terms of performance/area. In order to be able to evaluate the different architecture alternatives and design approaches which are explored and to be able to compare with state-of-the-art implementations, we define the *Architecture efficiency* (AE) metric as follows:

$$AE = \frac{Throughput \times N_{iter}}{Area_{Norm} \times f_{clk}} \tag{3.1}$$

Its unit of measure is $bits/cycle/iteration/mm^2$ and it represents the number of decoded bits per clock cycle per iteration per mm^2 that the proposed iterative channel decoder implementation is able to deliver. A high architecture efficiency indicates an optimized design which exploits efficiently its hardware resources during its execution time. It is worth to note here that even this metric does not exhibits directly the energy consumption measure it still has an indirect relation with it: improving the use of the hardware resources at each clock cycle will typically lead to improved energy efficiency for a fixed target throughput requirement.

An interesting point in the above expression of the AE concerns the normalization of the throughput achieved with respect to the considered clock frequency (f_{clk}) which increases the fairness when comparisons are done between different decoding architectures running at different clock frequencies. Published results in this context consider either the maximum achievable clock frequency by the proposed architecture or a lower operational clock frequency which is sufficient to achieve the target throughput. Thus, normalizing the presented throughput by the considered clock frequency enables to better exhibit the efficiency of the proposed architectural choices.

Towards the same objective, the above expression of the AE normalizes the throughput by the considered number of decoding iterations (N_{iter}) as the published results can use slightly different values which impact the overall throughput. In most of these works, the same low complexity decoding algorithms, with identical convergence speed, are used.

Similarly, the AE expression uses a normalized area measure ($Area_{Norm}$) as the published decoders are often based on different technology nodes (e.g. 180nm, 130nm, 65nm, etc.). To than end, the following scaling formula is used [119]:

$$Area_{Norm} = Area_{Given}(\frac{Tech_{Norm}}{Tech_{Given}})^2 \tag{3.2}$$

where:
$Tech_{Norm}$ = Feature size of the target technology for normalization; in our case it is 65nm,
$Tech_{Given}$ = Feature size of the technology used,
$Area_{Given}$ = Occupied Area,
$Area_{Norm}$ = Normalized Area.

In addition, when the published design area is given post-place and route a downscaling factor of 2 is applied to obtain a reasonable estimate of the post-synthesis area. This factor is not very accurate as it depends to many parameters (technology node, CAD tools, operating conditions, etc.), but it gives a reasonable idea as it corresponds to the usually observed ratio [117]. Similarly, the achievable maximum clock frequency can vary in this context. It is worth to note here that the technology nodes and foundries provide many different libraries associated with different characteristics and conditions (high speed, high density, low power, general purpose with worst, nominal, or best case operating conditions in terms of temperature and supply voltage) which can impact considerably the resulting area and maximum clock frequency. This issue increases the difficulty to make a fully fair comparison with state-of-the-art implementations which use different target technology and/or operating conditions.

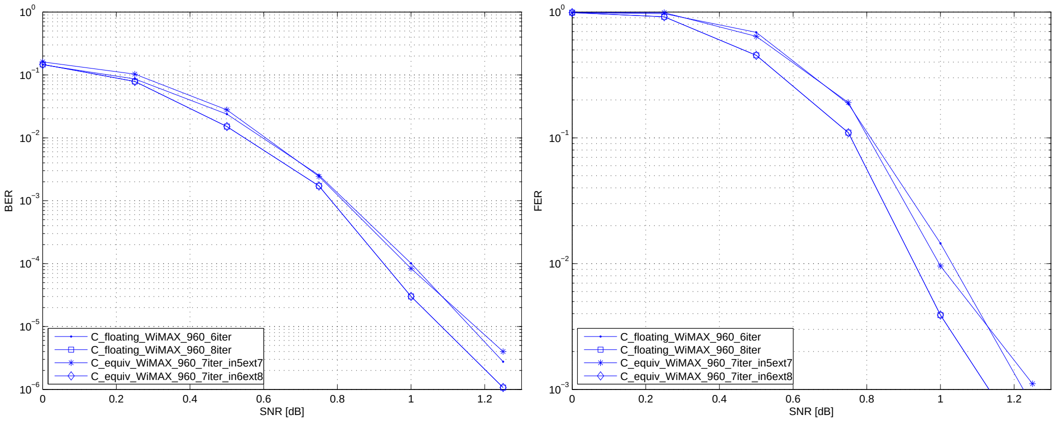
Finally, we consider in this evaluation that the compared architectures provide identical communication performances in terms of error rates, close to the reference optimal decoding of the considered codes and communication parameters. In this context, it is worth to note that an appropriate quantization should be used as besides its impact on the communication performance, it impacts significantly the channel decoder area in terms of memory, computation, and communication resources.

In this thesis work, we consider the technology feature size of the target technology for normalization to be $Tech_{Norm}$=65nm. Using the above expression, the architecture efficiency of the TurbASIP decoder (presented in the previous chapter) evaluates to 2.6 $bits/cycle/iteration/mm^2$.

3.1.2 Quantization analysis

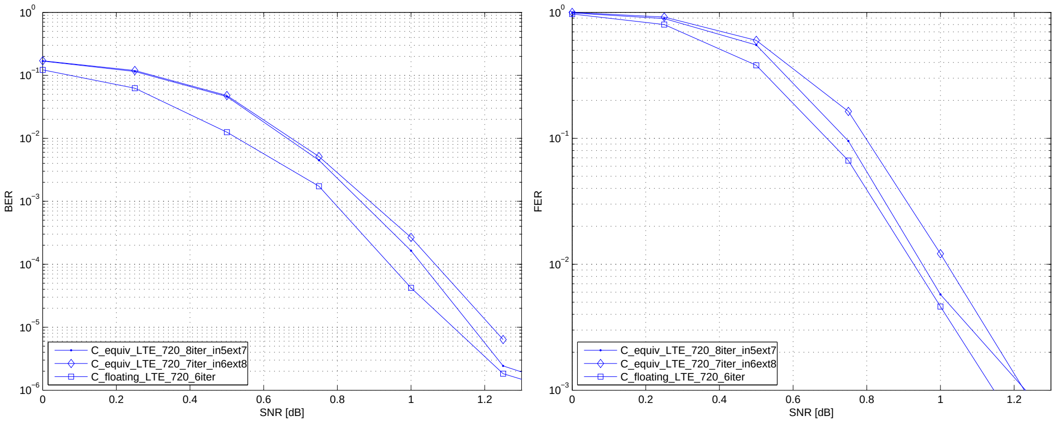
In the objective to achieve an area optimized channel decoder while preserving the communication performances in terms of error rates, accurate simulation analysis of the quantized C-models of Turbo and LDPC decoding algorithms are inevitable. The analysis presented in this section considers the quantization levels of input and extrinsic LLRs along with extrinsic LLR scaling factors S_c in order to achieve the required convergence with acceptable performance degradation in terms of bit error rates. For Turbo decoding, the algorithm considered is the Max-Log MAP algorithm presented in Chapter 1. For ease of presentation, we denote the quantization values of the input LLRs (in), the extrinsic LLRs (ex), and the extrinsic scaling factors (S_C) as (in, ex, S_C). Figure 3.1 presents the BER simulation results for 6 and 8 Turbo iterations for the WiMAX standard (DBTC) with a frame size of 1920 bits using floating point C-simulations. The figure also presents the BER simulation results for 7 iterations with two different quantizations: (6,8,0.875) and (5,7,0.875). Similar curves from floating point C-simulations (6 iterations) along with quantizations of (5,7,0.4375) and (6,8,0.4375) are drawn in Figure 3.2 for the LTE standard (SBTC) with a frame size of 1440 bits using radix-4 trellis compression with 8 and 7 iterations respectively. From both figures it can be observed that the quantizations (in,ext)=(6,8) and (5,7) perform close to the floating point simulation results with acceptable performance degradation of less than 0.2 dB. Additionally, scaling factors of 0.875 (for the DBTC of WiMAX) and 0.4375 (for the SBTC of LTE) were found suitable for both communication performance and hardware implementation requirements.

Regarding LDPC decoding, Figure 3.3 presents the reference curves obtained for the code rate 1/2 of WiMAX and WiFi LDPC codes for frame sizes of 1152 bits and 1296 bits. Floating point C-simulations of the NMS algorithm are found to have approximately 0.10 dB performance degradation w.r.t the ideal reference using Sum-Product Algorithm (SPA). The quantized model with (7,5,0.875) has less than 0.1dB w.r.t. the floating point C-simulations of the Sum-Product Algorithm. Very little BER performance improvements are obtained beyond 15 iterations. Authors of [107] present a look up table based update that further reduces the performance loss to less than 0.1 dB.



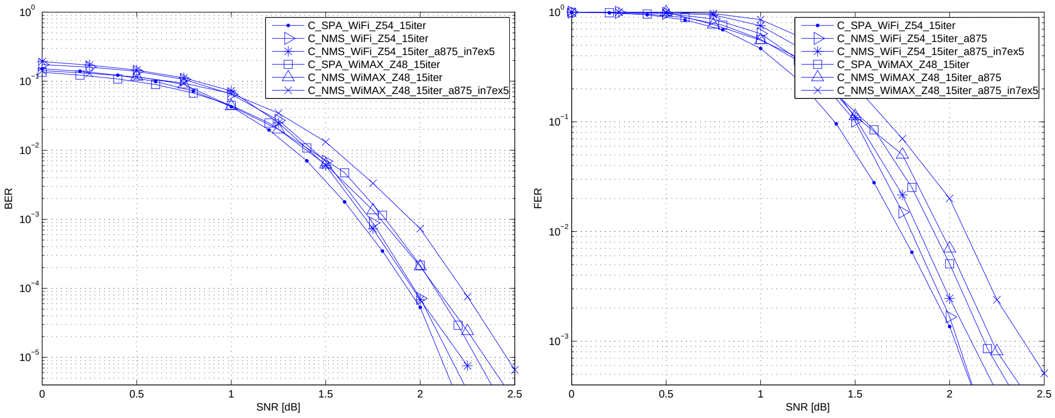
(a) BER results ***(b)*** FER results

Figure 3.1 **—** C-simulations BER and FER results for WiMAX frame size 1920 bits and code rate of 1/3



(a) BER results ***(b)*** FER results

Figure 3.2 **—** C-simulation BER and FER results for LTE frame size 1440 bits and code rate of 1/3



(a) BER results ***(b)*** FER results

Figure 3.3 **—** C-simulation BER and FER results for LDPC WiMAX Z=48 and WiFi Z=54 and code rate of 1/2

3.2 DecASIP$_{v1}$

In this section we present our first design approach towards the target flexible multi-ASIP architecture supporting the decoding of LDPC and Turbo codes. Regrading the support of Turbo decoding, the presented TurbASIP architecture in the previous chapter constitutes our starting point and this section details mainly the added or modified features. Additionally, this section presents the architectural enhancements to support the LDPC decoding with special emphasis on LDPC computational scheduling and hardware resource sharing (memories and communication structure).

3.2.1 System architecture

Based on the design motivations presented in the previous section, the proposed **"DecASIP$_{v1}$"** system architecture is shown in Figure 3.4. It consists of 8 DecASIPs interconnected via a de-Bruijn network. The topology of the de-Bruijn NoC is as shown in Figure 3.5. For the Turbo decoding mode, the DecASIPs of each of the two component decoders are also connected by two 10-bit buses (referenced here as $\alpha - \beta\ bus$) to allow the exchange of sub-block boundary state metrics. Each DecASIP has 3 memory banks of size 256×24 which are used to store the input channel LLR values (input CV memories). The extrinsic LLR values are stored in 3 banks of size 256×30. Each DecASIP is further equipped with two 32×80 memories (not shown in Figure 4.5) which implement buffers to store the state metrics β in Turbo mode and the variable to check node messages $L(n,m)$ in LDPC mode.

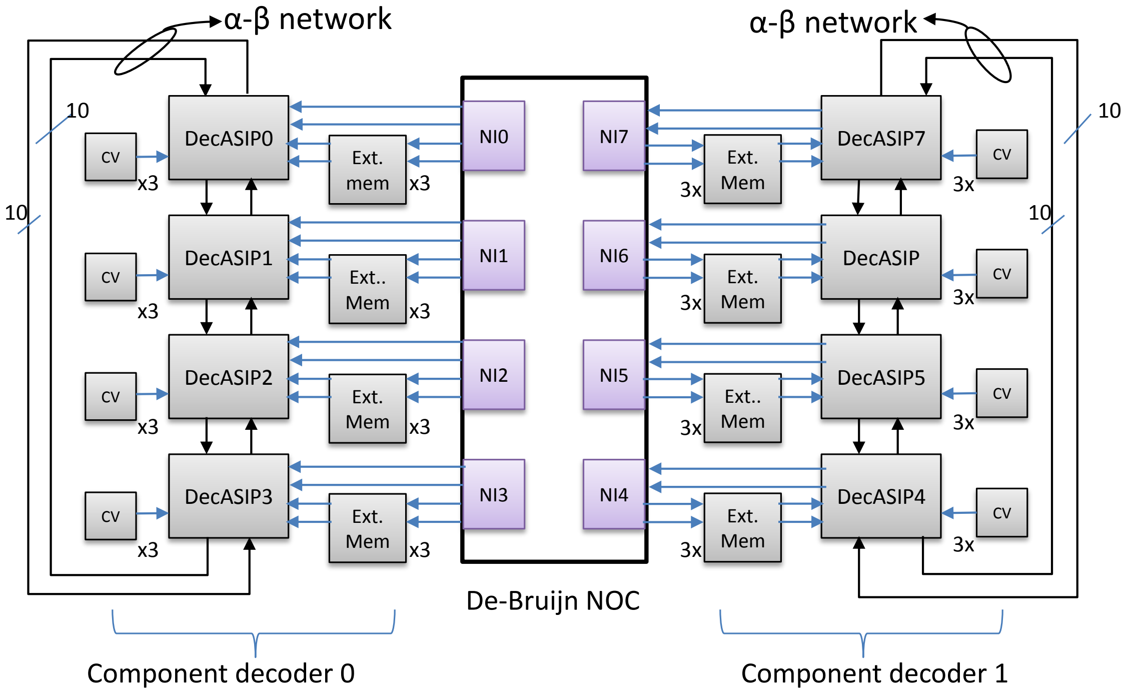


Figure 3.4 — DecASIP$_{v1}$ System Architecture

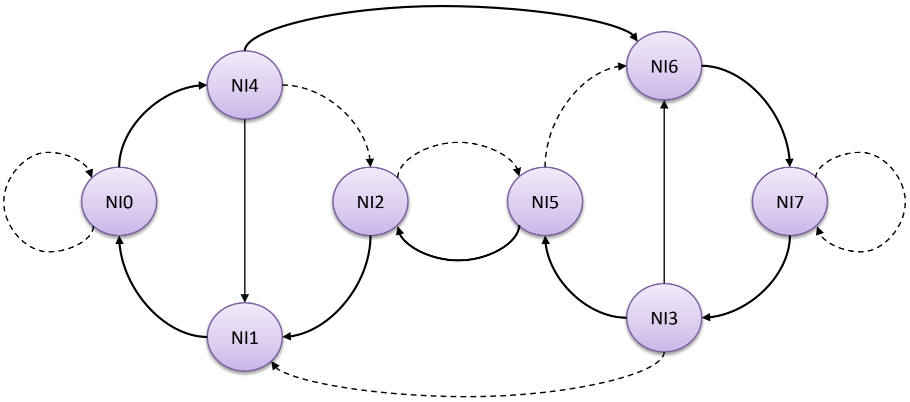


Figure 3.5 **—** Binary de-Bruijn NoC topology for 8 nodes

3.2.2 Turbo mode

The system architecture in Turbo mode is quite similar to the TurbASIP system architecture presented in Section 2.3 (page 46), with DecASIPs connected to the Network Interfaces (NI) (0,1,2,3) and (4,5,6,7) as component decoder0 and component decoder1 respectively. As a design choice, a maximum window size of 64 symbols is adopted. Window sizes below this value present boundary effects that result in performance degradation [16] for a given number of iterations. Each input memory bank can store a maximum of 4 windows and each DecASIP can process a maximum of 12 windows (3 banks with 4 windows each). Thus, the maximum frame size that can be supported in Turbo mode with the 4×4 DecASIP system architecture is 6144 bits (which is the maximum frame size specified in the LTE standard). The α and β state metric exchanges across sub-block boundaries are done via the 10 bit α-β buses.

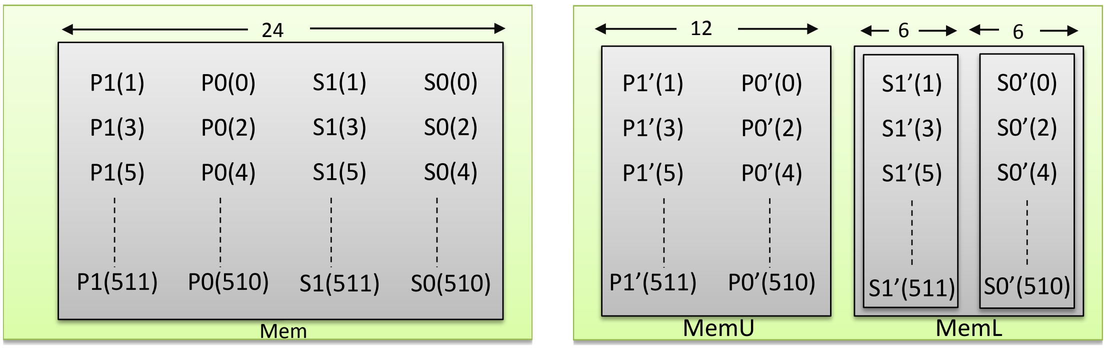
3.2.2.1 Memory architecture

As mentioned in the previous section, the input channel values (LLRs) are quantized to 6 bits in Turbo mode (SBTC, DBTC). Systematic LLRs (S1, S0) and parity LLRs (P1, P0) are stored in the input memory banks as shown in Figure 3.6. Each of the 3 input memory banks of the DecASIPs of component decoder0 are 24 bits wide. Note that for SBTC mode, systematic channel LLRs (S1', S0') of the component decoder1 arrive in interleaved order, while the parity channel LLRs (P1', P0') arrive in natural order. Hence, each input memory bank of the component decoder1 are split into 3 sub-banks of width 12, 6, and 6.

Each extrinsic memory bank is divided into 2 smaller banks (15 bits each). In DBTC mode, $\gamma^{n.ext01}$ and $\gamma^{n.ext10}$ are stored in the MemL, while $\gamma^{n.ext11}$ is stored in MemU (Figure 3.7a). In SBTC mode, the two extrinsic LLRs (γ^{ext0} and γ^{ext1}) corresponding to S0 and S1 are stored in the lower and upper memory respectively as shown in Figure 3.7b.

As Radix-4 trellis compression technique is used in SBTC mode, (i)th and (i+1)th bit extrinsic LLRs (γ^{ext_i}, $\gamma^{ext_{i+1}}$) corresponding to the systematic bits (S_i, S_{i+1}) are generated in pairs and are written in the extrinsic memory banks of the other component decoder in interleaved/deinterleaved order as required. This implies that the write addresses of these bit extrinsic LLRs can be different. In DBTC mode, the 3 normalized extrinsic LLRs ($\gamma^{n.ext11}$, $\gamma^{n.ext10}$, $\gamma^{n.ext01}$)$_i$ have the same write address corresponding to the same ith systematic bit pair (S0,S1)$_i$.

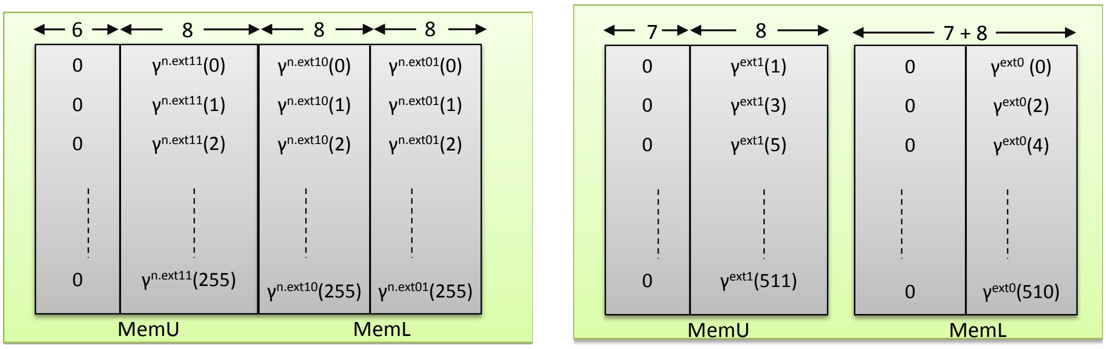
In order to enable simultaneous writing in both modes, the extrinsic memory banks are divided as shown in Figure 3.7.



(a) Component decoder 0

(b) Component decoder 1

Figure 3.6 — DecASIP$_{v1}$: Input memory (CV) bank0 organization in Turbo mode



(a) In DBTC mode

(b) In SBTC mode

Figure 3.7 — DecASIP$_{v1}$: Extrinsic memory bank0 organization in Turbo mode

3.2.2.2 Processing schedule

Each of the DecASIP recursion unit is optimized to process the SBTC trellis specified in the LTE standard with Radix-4 trellis compression along with the DBTC trellis specified in the WiMAX standard in forward and backward recursion modes. The forward-backward schedule (presented in Section 1.3.4.1, page 21) is slightly modified in this implementation by first executing the backward schedule followed by the forward schedule. Figure 3.8 illustrates the adopted processing schedule in Turbo mode. Notice that this scheme ensures that the extrinsic information generated are always in sequence within a sub-block requiring initialization at the beginning of the sub-block. As in the TurbASIP case, the sub-block boundary state metric initialization are done via exchange of α and β state metrics at the end of each iteration (Figure 4.5) through the α-β buses.

3.2.2.3 Pipeline architecture

Figure 3.9 presents the pipeline stages of the DecASIP$_{v1}$ using the same building blocks of the TurbASIP (referred in Section 2.3.2). The main difference here remains in the utilization of one

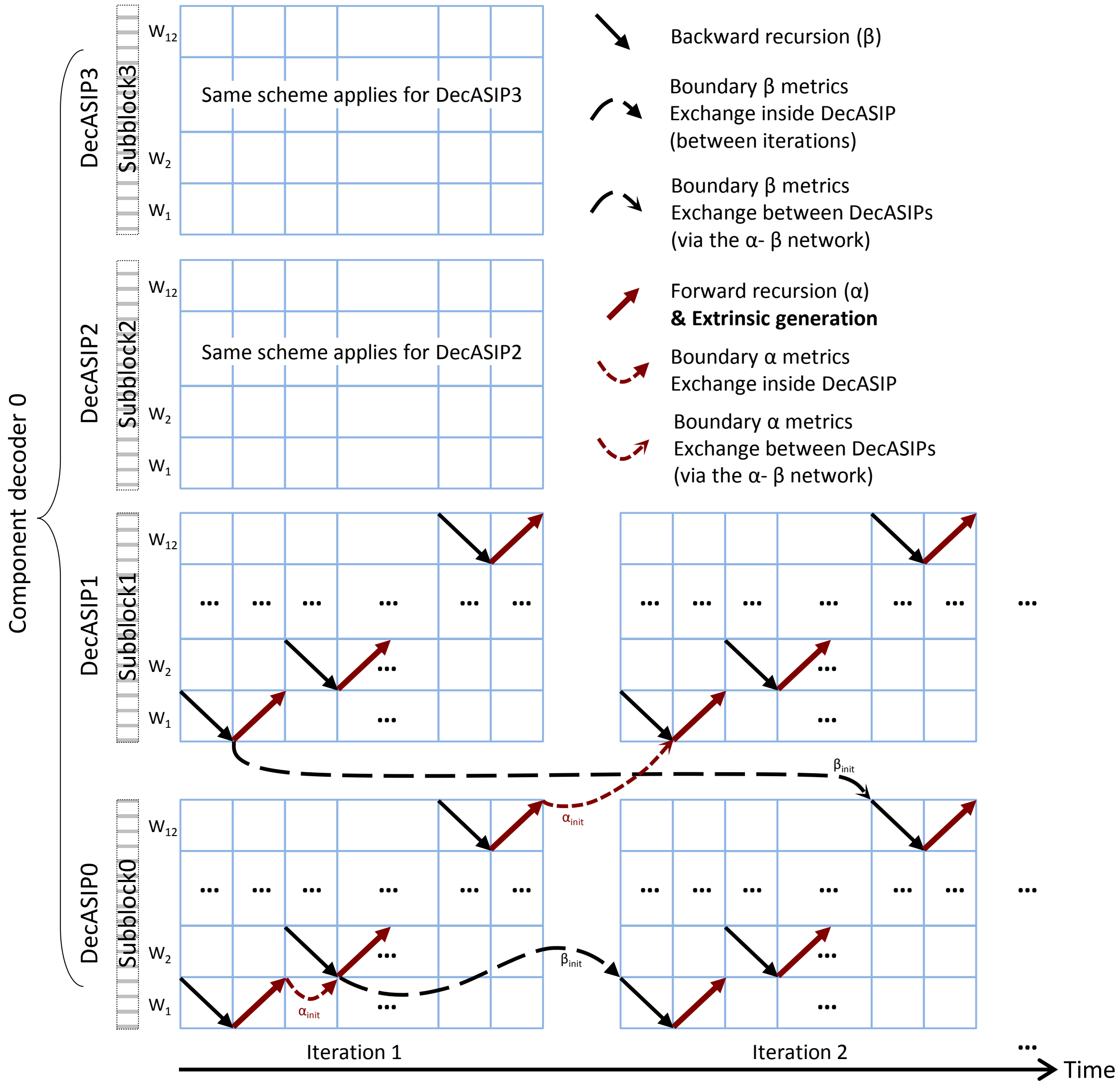


Figure 3.8 — DecASIP$_{v1}$: Backward-Forward schedule adopted in Turbo mode. The number of processed windows per DecASIP depends on the frame size, the maximum number of windows per DecASIP is 12.

recursion unit that processes the trellis steps in the backward and the forward directions in the *Ex* pipeline stage. Furthermore, the *ST* pipeline stage produces two extrinsic LLRs in SBTC mode and 3 normalized (w.r.t. to γ^{ext00}) symbol extrinsic LLRs in DBTC mode.

3.2.2.4 Interleave/deinterleave address generation

The generated extrinsic information packets also carry the address header which determines the destination DecASIP and the memory address at which the data is written. The interleaving/deinterleaving addresses required w.r.t. the LTE standard QPP interleaving rule is as described below.

Let N be the frame size in bits at the encoder input. For $j = 0...N-1$, $I(j) = (F_1 * j + F_2 * j^2) mod N$, where F_1 and F_2 are constants defined in the standard with j being the index of the natural order. These addresses can be recursively derived using the

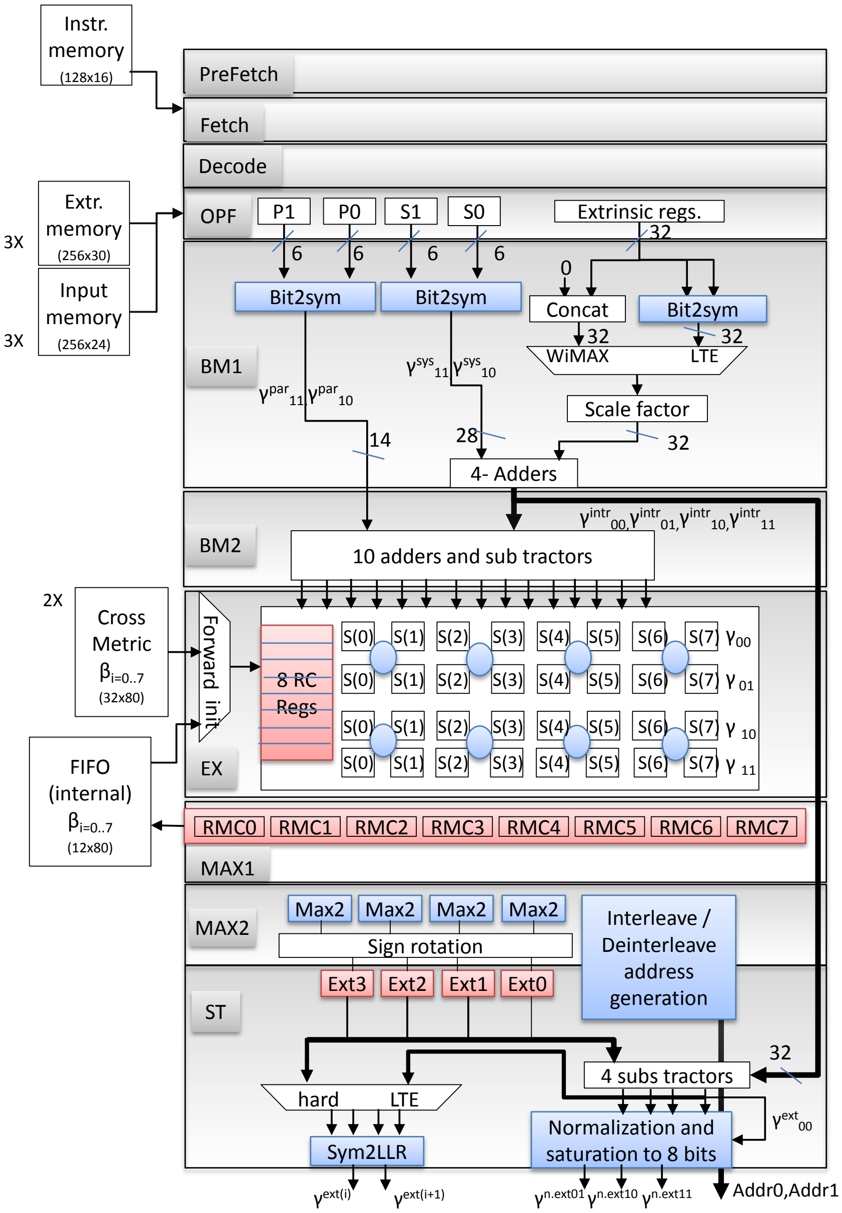


Figure 3.9 — DecASIP$_{v1}$: Pipeline architecture in Turbo mode

following expressions:

$$I(j+1) = (I(j) + G(j))modN \tag{3.3}$$

$$G(j) = (G(j-1) + 2F_1)modN \tag{3.4}$$

The deinterleaved address pattern required by component decoder1 can be generated recursively as described here. Let the deinterleaved address sequence be $D = [d_0, d_1, ..d_{N-1}]$. Taking a second order modulo-N linear circular difference of the sequence D gives step size values as given by the equation (3.5) and (3.6) below. The number of steps (N_s) depends on the frame size and can take at most 8 different values.

$$D' = [d_0 - d_{N-1}, d_1 - d_0, ..., d_{N-1} - d_{N-2}]modN \tag{3.5}$$

$$D'' = [D'_0 - D'_{N-1}, D'_1 - D'_0, ..., D'_{N-1} - D'_{N-2}]modN \tag{3.6}$$

value	LTE 1440 bits		WiMAX 1920 bits	
	Interleaved	deinterleaved	Interleaved	deinterleaved
Step 0	840	120	39	659
Step 1	840	120	35	587
Step 2	840	120	59	759
Step 3	840	120	135	87
Step 4	840	120	39	659
Step 5	840	120	35	587
Step 6	840	120	59	759
Step 7	840	120	135	87
Seed 0	929	1409	886	302
Seed 1	1111	1351	0	0

Table 3.1 — DecASIP$_{v1}$: Interleaved/deinterleaved address generation step and seed values in Turbo mode

The following pseudo code illustrates the deinterleave address generation process:

```
$for\ i = 1 : N$
    $d(i) = (d_{(i-1)} - D'_{(i-1)})modN;$
    $D'_i = (D'_{(i-1)} + D''((i-1)mod(N_s)))modN;$
$end$
```

Similar sequences is generated for the ARP interleaver specified in the WiMAX standard, where the number of steps obtained is maximum 4. Figure 3.10 presents the corresponding hardware generation architecture. Table 3.1 gives an example of the *step* and seed values for LTE and WiMAX frames of length 1440 bits and 1920 bits respectively. Similar values can be derived from the above expressions for all frame sizes specified in these standards.

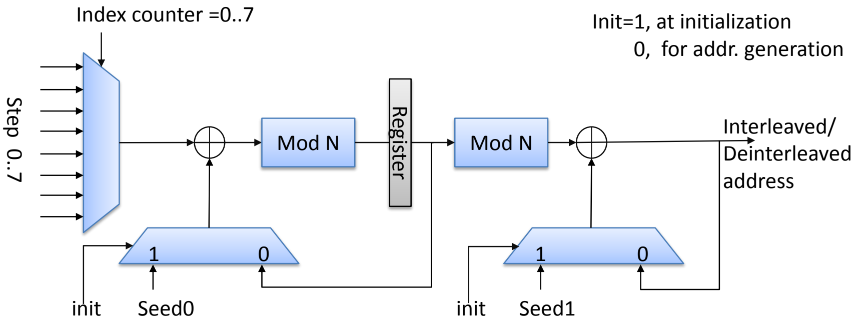


Figure 3.10 — DecASIP$_{v1}$: ARP and QPP interleaved/deinterleaved address generation in Turbo mode

3.2.2.5 NoC messages

As the maximum number of NoC messages (extrinsic information) generated in the forward recursion is 2 (SBTC mode), the NoC message format is as shown in Figure 3.11. In the DBTC mode, two packets are generated one carrying ($\gamma^{n.ext10}$ & $\gamma^{n.ext11}$) and the other carrying $\gamma^{n.ext01}$. Both packets are addressed to the same destination. The router adds 13 more bits for network congestion management, namely the time stamp and priority bits. Detailed description on this NoC routing can be found in [20].

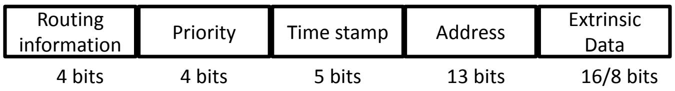


Figure 3.11 **—** DecASIP$_{v1}$: NoC packets format in Turbo mode

3.2.2.6 Assembly code

An assembly code example of the DecASIP in Turbo mode is as shown in Listing 3.1. First we initialize the DecASIP mode (SBTC, DBTC), the current window counter value ($n = 0$), the number of windows (L) per DecASIP, the length of windows (W) and the length of the last window (W_L). The *REPEAT* instruction controls the number of iterations (*ITER_MAX=6*). Notice the absence of the *PUSH* and *POP* instructions (used in TurbASIP), which are implied through the *REPEAT* instruction that recursively increments the window counter until *WINDOW_N* to complete one iteration and then initializing it back to zero at the beginning of a new iteration. Furthermore, the *REPEAT* instruction also enables the execution of the *EXE_REC* only if the last window is executed in the previous iteration. This ensures that the boundary state metric LLRs are exchanged only after the last window is processed. For the first iteration (iter=0) the DecASIPs start with zero as the initial state metric ($\alpha\_int(w_{n=0}^{iter=0})(i = 0) = \beta\_int(w_{n=0}^{iter=0})(i = W - 1) = 0$). During the first iteration, the extrinsic memory contents are not read as they are still uninitialized.
As in the TurbASIP design, the *ZOLB* instruction enables the execution of the instructions at lines @26-27 and @30-32 to execute W number of times. In case the current window being processed is the last window of the sub-block, instructions at lines @26-27 and @30-32 are executed W_L number of times.

Listing 3.1 **—** DecASIP$_{v1}$: Assembly code in DBTC mode. Example with a frame size of 1920 bits and 8-DecASIP system decoder

```
 1  ;initialize DecASIPs registers from configuration memory
 2            SET CONF double
 3  _INIT:    NOP
 4  ;set current window counter n=0
 5            SET_WINDOW_ID 0
 6  ;set max. number of windows
 7            SET_WINDOW_N 3
 8  ;set regular (W) and last window length W_L
 9            SET_SIZE 63,48
10  ;repeat @11−41 if last window executed else
11  ;repeat @28−41, for 6∗WINDOW_N times
12            REPEAT until _LOOP 6 times
13            NOP
14  ;exchange alpha0−beta0 of state0−7
15            EXE_REC ALPHA_BETA0
```

```
16            EXE_REC ALPHA_BETA1
17            EXE_REC ALPHA_BETA2
18            EXE_REC ALPHA_BETA3
19            EXE_REC ALPHA_BETA4
20            EXE_REC ALPHA_BETA5
21            EXE_REC ALPHA_BETA6
22            EXE_REC ALPHA_BETA7
23   ;repeat 30−31, and 35−36 for Current ”WindowLen” times
24            ZOLB _RW1,_CW1,_LW1
25            NOP
26   _RW1:    DATA LEFT ADD M COLUMN2
27            NOP
28   ;save last beta load alpha_init
29   _CW1:    EX_BETA_ALPHA
30            DATA RIGHT ADD M COLUMN2
31   ;gen ext
32   _LW1:   EXTCALC add i line2 EXT
33   ;save last alpha load beta_init if lastwindow
34   ;else exchange calculated alpha and beta
35            EXCH_WIN
36            NOP
37   _LOOP:   NOP
38   ;finish decoding and halt
39        PROC_STOP
```

Backward recursion

The *DATA_LEFT* instruction executes the backward recursion calculating the β metrics. At the end of the window processing in the backward direction, the *EX_BETA_ALPHA* instruction saves the last calculated $\beta\_int(w_n^{iter})(i = 0)$ state metric in the boundary state memory (internal FIFO) and loads the $\alpha\_int(w_{(n)}^{iter})$=$\alpha(w_{(n-1)}^{iter})$ metric of the previous window to the state metric registers (*RMC*).

Forward recursion

The *DATA_RIGHT* instruction executes the forward recursion calculating the α metrics, while the instruction *EXTCALC* calculates the extrinsic information (equation (1.37), page 20) and sends them to the other component decoder through the de-Bruijn NoC. The *EXTCALC* instruction also asserts the interleave/deinterleave generation logic to generate the address that forms the address field of the NoC message. In case of SBTC mode, the address generation logic is asserted twice to obtain the addresses of two bit extrinsic LLRs generated (equations (1.32) and (1.37), page 20). In DBTC mode, the address generation logic is asserted once to obtain the address for the normalized symbol LLRs. The *EXCH_WIN* instruction increments the current window counter $(n = n + 1)$ and forwards the last $\alpha\_int(w_{n-1}^{iter})(i = W - 1)$ values as $\alpha\_int(w_{(n)}^{iter})(i = 0)$. It also initializes the state metric registers (*RMC*) that now contain $\beta_{w_n^{iter}}(i = W - 1)$ with $\beta_{w_{n+1}^{iter-1}}(i = 0)$ of window $n + 1$, thus preparing the DecASIP for the next window processing.

The above two steps are repeated until all windows and all iterations are completed. During the last iteration, the *REPEAT* instruction sets a flag to generate the hard decisions while executing the forward recursion.

The trellis termination strategy used in the SBTC of the LTE standard is through zero padding and is achieved by inserting 3 zeros at the end of encoding process (Sub-section 1.2.1, page 13). This equates to processing an extra 1 and half symbols (after radix-4 compression) by DecASIPs 3 and 4. As per the specifications of the standard, these tail bits do not exchange extrinsic information but are needed to be processed to estimate the initial states for the backward recursion of the last window of DecASIPs 3 and 4. In the DecASIP$_{v1}$, this tail bits processing is achieved by rounding off the extra 1 and half symbols to 2 symbols by adding dummy zero LLRs to the last bit and processing these last 2 symbols separately as a window (as shown in the Listing 3.2). The instructions between lines @11 and @12 process these tail bits by initializing the fetch units to one more than the last window, i.e. L+1, and executing the two symbols in the backward direction. This initializes the $\beta_{w_{n=L}^{iter=0}}(i = W_L - 1)$ boundary state register.

After this tail bits processing is completed, the window counters are initialized back to 0 and the execution continues similar to the DBTC mode. During the tail bits processing, all other DecASIPs (0,1,2 and 5,6,7) execute idle (NOP) instructions.

Listing 3.2 — DecASIP$_{v1}$: Assembly code in SBTC mode. Example with a frame size of 1440 bits and 8-DecASIP system decoder

```
 1  ;initialize DecASIPs registers from configuration memory
 2          SET CONF single
 3  ; set number of windows to L+2
 4          SET_WindowsN 5
 5  ;set initial window counter to L+1 and k=1
 6          SET_WindowsInit 4, 1
 7  ; execute (11) and (12) twice
 8          ZOLB _RW,_RW,_LW
 9          NOP
10          ; replaced with NOP for DecASIPs 0..2 and 4..6
11  _RW:    DATA LEFT ADD metric column2
12  _LW:    NOP
13  ; exchange window boundary i.e. initialize window 3 with
14  ; tail bits state values
15  ; replaced with NOP for DecASIPs 0..2 and 4..6
16          EXCH_WIN
17  ;flush pipeline
18          NOP
19          NOP
20          NOP
21  ;set current window counter n=0
22          SET_WINDOW_ID 0
23  ;set max. number of windows
24          SET_WINDOW_N 3
25  ;set regular (W) and last window length W_L
26          SET_SIZE 63,52
27  ;execute normal window processing as in DBTC case
```

3.2.3 LDPC mode

In LDPC mode, each DecASIP operates as a variable node and check node processing engine. Each DecASIP$_{v1}$ processes 3 check nodes and its associated edges from the 3 consecutive variable node groups: e.g. DecASIP0 processes all the check nodes (3 at a time) that are associated with variable node groups VNG[0..2], while DecASIP1 processes those associated with VNG[3..5], etc. In other words, each DecASIP in this mode can process three CNs present in the same group and its corresponding variable nodes present in three different variable node groups.

3.2.3.1 Proposed scheduling illustrated with simple example using 2-DecASIP$_{v1}$ architecture

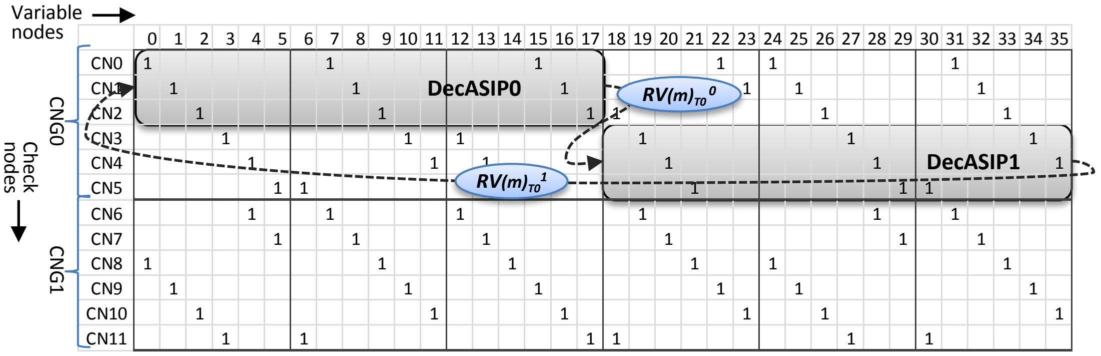
In order to illustrate the proposed original computational scheduling in LDPC mode towards the target scalable multi-ASIP channel decoder, let us consider an example of a simple LDPC check matrix and a 2-DecASIP architecture. The example of a simple LDPC check matrix with 36 variable nodes and 12 check nodes is represented with the H_{base} permutation matrix of Figure 3.12. This H_{base} matrix example has an expansion factor Z=6, thus 2 check node groups (CNGs) and 6 variable node groups (VNGs). The proposed scheduling for LDPC decoding is illustrated as follows:

I_0	I_1	I_3	I_4	I_1	I_2
I_4	I_1	I_0	I_1	I_4	I_1

Figure 3.12 — Simple LDPC H_{base} matrix example with N_b=6, M_b=2, and Z=6

at time $t = T_0$: DecASIP0 reads the LLR values associated with the check nodes m=(0,1,2). These LLRs correspond to $L(n)$ and $L(m,n)$ 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 memory banks should be organized in a way to enable simultaneous access to all these values.

Similarly, DecASIP1 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 3.13).



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
CN0	1							1								1							1		1							1				
CN1		1							1								1							1		1							1			
CN2			1							1								1	1								1							1		
CN3				1							1		1							1								1							1	
CN4					1							1		1							1								1							1
CN5						1	1															1								1	1					
CN6					1			1					1							1									1			1				
CN7						1			1					1							1									1			1			
CN8	1									1					1							1			1									1		
CN9		1									1					1							1			1									1	
CN10			1									1					1							1			1									1
CN11				1			1											1	1									1			1					

Figure 3.13 — DecASIP$_{v1}$: Proposed LDPC decoding schedule with 2-DecASIP architecture at time step t=T_0

Each DecASIP calculates variable to check node messages $L(n,m)$ according to (1.59). Additionally, it processes equation (equation 1.62) and produces 3 sets of messages (corresponding to the three check nodes) that we denote by $RV(m)^k_{T0}$, where k refers to the DecASIPs $[0,1]$. The $RV(m)^k_{T0}$ 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 $L(n,m)$ messages calculated for the check node m.

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

at time $t = T_1$: DecASIP0 reads the LLR values associated with the check nodes m=(3,4,5). These LLRs correspond to $L(n)$ and $L(m,n)$ values related to the 9 variable nodes n=[(3,4,5),(10,11,6),(12,13,14)]. Similarly, DecASIP1 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 3.14). Both DecASIPs generate $RV(m)^k_{T1}$ messages as in the previous step except that the new messages take into account the $RV(m)^k_{T0}$ messages received from the other DecASIP.

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

Similarly, $RV(m)^0_{T1}$ 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) $UV(m)^k$ that is circulated again to the next DecASIP as shown in Figure 3.14.

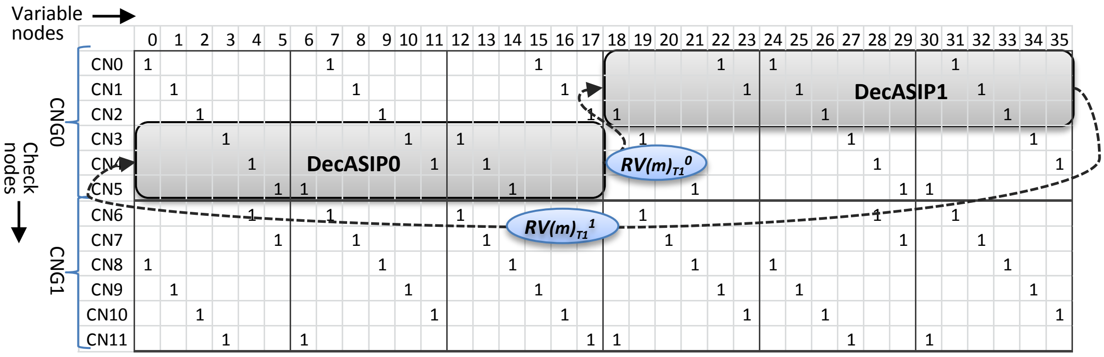


Figure 3.14 — DecASIP$_{v1}$: Proposed LDPC decoding schedule with 2-DecASIP architecture at time step t=T_1

at time $t = T_2$ DecASIP0 calculates the final aposteriori LLRs $L(n)$ using the $UV(m)^1$ messages and the $L(n,m)$ messages related to n=[(0,1,2),(7,8,9),(15,16,17)] and m=(0,1,2) according to equation (1.64).

Extrinsic messages $L(m,n)$ are generated according to equation (1.63) and stored in the extrinsic memory banks of the DecASIP.

Similarly, DecASIP1 calculates LLRs $L(n)$ using $UV(m)^0$ message, where n=[(19,20,21), (27,28,29), (34,35,30)] and m=(3,4,5). Both DecASIPs forward the $UV(m)$ message to the next DecASIP (Figure 3.15).

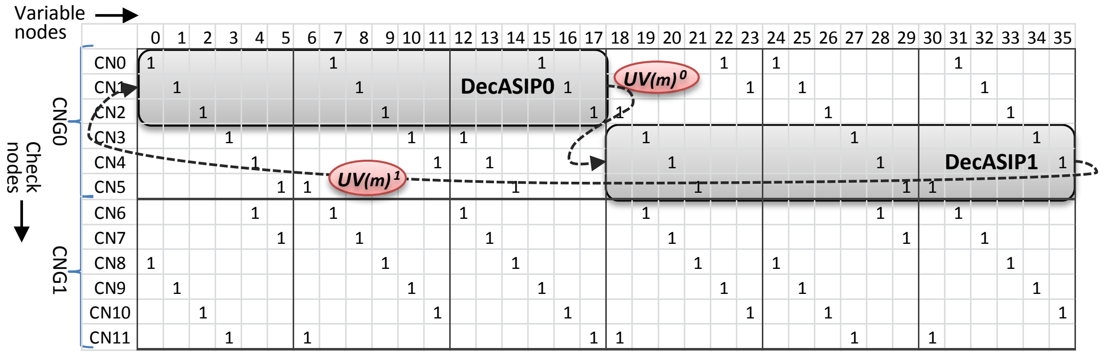


Figure 3.15 — DecASIP$_{v1}$: Proposed LDPC decoding schedule with 2-DecASIP architecture at time step t=T_2

at time $t = T_3$ Similar to the previous step at $t = T_2$, DecASIP0 calculates aposteriori LLRs $L(n)$ using $UV(m)$ where n=[(19,20,21),(27,28,29),(34,35,30)] and m=(0,1,2) and DecASIP1 calculates $L(n)$ for n=[(0,1,2),(7,8,9),(15,16,17)] using $UV(m)$ where m=(3,4,5) as shown in Figure 3.16.

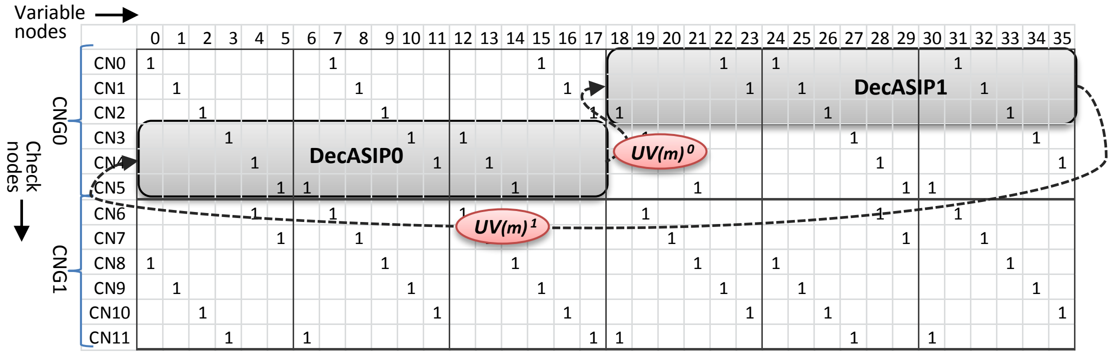


Figure 3.16 — DecASIP$_{v1}$: Proposed LDPC decoding schedule with 2-DecASIP architecture at time step t=T_3

The above 4 time steps completes one sub-iteration carried over a Check node group (CNG0) with 2 DecASIPs. 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 DecASIPs x (1RV+1UV)) where each DecASIP processes 3 check nodes associated with 3 VNGs simultaneously.

3.2.3.2 Proposed scheduling with 8-DecASIP$_{v1}$ architecture

All LDPC check matrices specified in WiFi and WiMAX standards contain 24 VNGs. With the proposed internal parallelism degree for DecASIP$_{v1}$ (3 check nodes and its associated edges from 3 consecutive VNGs), it is possible to process all the VNGs simultaneously with 8 DecASIPs. Thus, with the proposed scheduling, these 8 DecASIPs will be able to process

24 check nodes (8x3) simultaneously. Now, if the CNG contains 24 check nodes (Z=24), a sub-iteration is completed in 16 time steps (2 phases x 8 DecASIPs). We will illustrate in this section the proposed scheduling for an 8 DecASIPs architecture ($N_A = 8$) and with higher sub-martix size ($Z = 48$). In this case, each group of CNs should be divided into two sub-groups, each containing 24 check nodes.

For the ease of explanation, we denote n=$\mathbb{P}_x^{VNG_y}(m)$ as the variable node n of VNG$_y$ which is connected to the check node m of the x^{th} check node group of the LDPC H_{base} matrix. Check node and variable node update cycles over a check node sub-groups are scheduled in the following order:

1. at time t=T_0: DecASIPs calculate min0, min1 and sgn values as given below:

$$RV^k_{T0}(m) \left\| \begin{array}{l} \text{with VN n=}\mathbb{P}_0^{VNG_y}(m), \\ \text{for DecASIP0,}m\text{=[0,1,2], }x\text{=(0,1,2)} \\ \text{for DecASIP1,}m\text{=[3,4,5], }x\text{=(3,4,5)} \\ \vdots \\ \text{for DecASIP7,}m\text{=[21,22,23],}x\text{=(21,22,23)} \end{array} \right. \quad (3.7)$$

   $min0(m), min1(m)$ are the minimum and second least minimum calculated for the check node m. The location of the $min0(m)$ is identified through $LOC(m)$, which consists of ***ASIP ID*** and the variable node group number of the variable node n. The partial $min0$, $min1$ information thus obtained is communicated through the NoC to the next DecASIP that contain the next set of edges connected to the check node m. We refer the NoC message sent as ***Running Vector*** (RV) message. Thus, each DecASIP sends 3 RV messages corresponding to the 3 processed check nodes (Figure 3.18a).

$$RV^k_{T0}(m) = [min0, min1, LOC, sgn](m) \quad \forall\ k\text{=current DecASIP=[0,1,2..7]} \quad (3.8)$$

2. at time t=T_1: DecASIPs calculate the $min0$ and $min1$ as in the previous step, except that it now includes the $min0^{k1}_{T0}(m), min1^{k1}_{T0}(m)$ of the RV messages received from the previous DecASIP, hence the $min0$ and $min1$ calculations of equation (3.7) are modified as shown below:

$$min0(m) = \min(|L(n,m)|, min0^{k1}_{T0}(m)) \quad (3.9)$$
$$min1(m) = \min(|L(n,m)|, min1^{k1}_{T0}(m)) \neq min0(m) \quad (3.10)$$

$$RV^k_{T1}(m) = \begin{cases} \text{where n=}\mathbb{P}_0^{VNGx}(m), \\ \text{for DecASIP0,}m\text{=[21,22,23], }x\text{=(21,22,23)} \\ \text{for DecASIP1,}m\text{=[0,1,2], }x\text{=(0,1,2)} \\ \vdots \\ \text{for DecASIP7,}m\text{=[18,19,20],}x\text{=(18,19,20)} \\ \text{k1=DecASIP[(k-1)}\%N_A\text{], k=current DecASIP,} \\ N_A\text{=number of DecASIPs} \end{cases} \quad (3.11)$$

   Thus, the new RV message generated carries the $min0, min1$ information of 6 edges connected to a check node in CNG0 (Figure 3.18b). The process continues until time $t = T_7$

(Figure 3.19a) by which the $RV^k_{T7}(m)$ message carries the information related to all edges connected to the check nodes $m = [0, 1, 2, ..23]$. This final RV is now considered to be the variable node message *Update Vector* (UV) from which the variable node update messages can be calculated in the next time step.

$$UV^k(m) = RV^k_{T7}(m) = [min0, min1, LOC, sgn](m) \tag{3.12}$$

3. at time t=T_8: $UV^k(m)$ messages are used by the corresponding DecASIPs to calculate and update the variable nodes.

$$updtMag(m) = \begin{cases} min0(m) \text{ if } CurrLOC(m) \neq UV^k[LOC(m)] \\ min1(m) \quad \text{otherwise} \end{cases} \tag{3.13}$$

$$\begin{aligned} &L(m,n) = \alpha * sgn(m) * (updtMag(m)) \\ &L(n) = L(n,m) + L(m,n) \end{aligned} \begin{cases} \text{where n=}\mathbb{P}_0^{VNGx}(m), \\ \text{for DecASIP0,}m\text{=[0,1,2], }x\text{=(0,1,2)} \\ \text{for DecASIP1,}m\text{=[3,4,5], }x\text{=(3,4,5)} \\ \vdots \\ \text{for DecASIP7,}m\text{=[21,22,23],}x\text{=(21,22,23)} \end{cases} \tag{3.14}$$

The structure of the QC-LDPC codes implies that there is a unique set of edges between a check node group and a variable node group, i.e. a variable node has at maximum one single edge in a check node group. Therefore, the RV calculation of the check nodes $m >=$ 24, yet inside the same check node group, can be performed in parallel along with the UV message processing of the check nodes m=[0,1,..23]. Hence, for the considered sub-matrix size $Z = 48$, the DecASIPs perform the $RV^k_{T8}(m)$ of check nodes m=[24,25...47] in parallel to the UV of check nodes m=[0,1...23] as illustrated in Figure 3.19b.

$$RV^k_{T8}(m) = \begin{cases} \text{where n=}\mathbb{P}_0^{VNGx}(m), \\ \text{for DecASIP0,}m\text{=[24,25,26], }x\text{=(24,25,26)} \\ \text{for DecASIP1,}m\text{=[27,28,29], }x\text{=(27,28,29)} \\ \vdots \\ \text{for DecASIP7,}m\text{=[45,46,47],}x\text{=(45,46,47)} \\ \text{k1=DecASIP[(k-1)\%}N_A\text{], k=current DecASIP,} \\ N_A\text{=number of DecASIPs} \end{cases} \tag{3.15}$$

Therefore, the *VN-update* phase of the first sub-group can take place along with the *CN-update* of the second sub-group. This combined UV and RV phase continues until time t=T_{T15}.

4. at time t=T_{16} to t=T_{23}: DecASIPs calculate the UV phase for the second sub-group of check nodes m=[24,25...47] as illustrated in Figure 3.20.

Thus, one sub-iteration is completed in 24 time steps: 8 DecASIPs × (1RV+1RVUV+1UV). This scheduling based on sub-groups of check nodes enables us to handle efficiently all specified sub-matrix sizes 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:

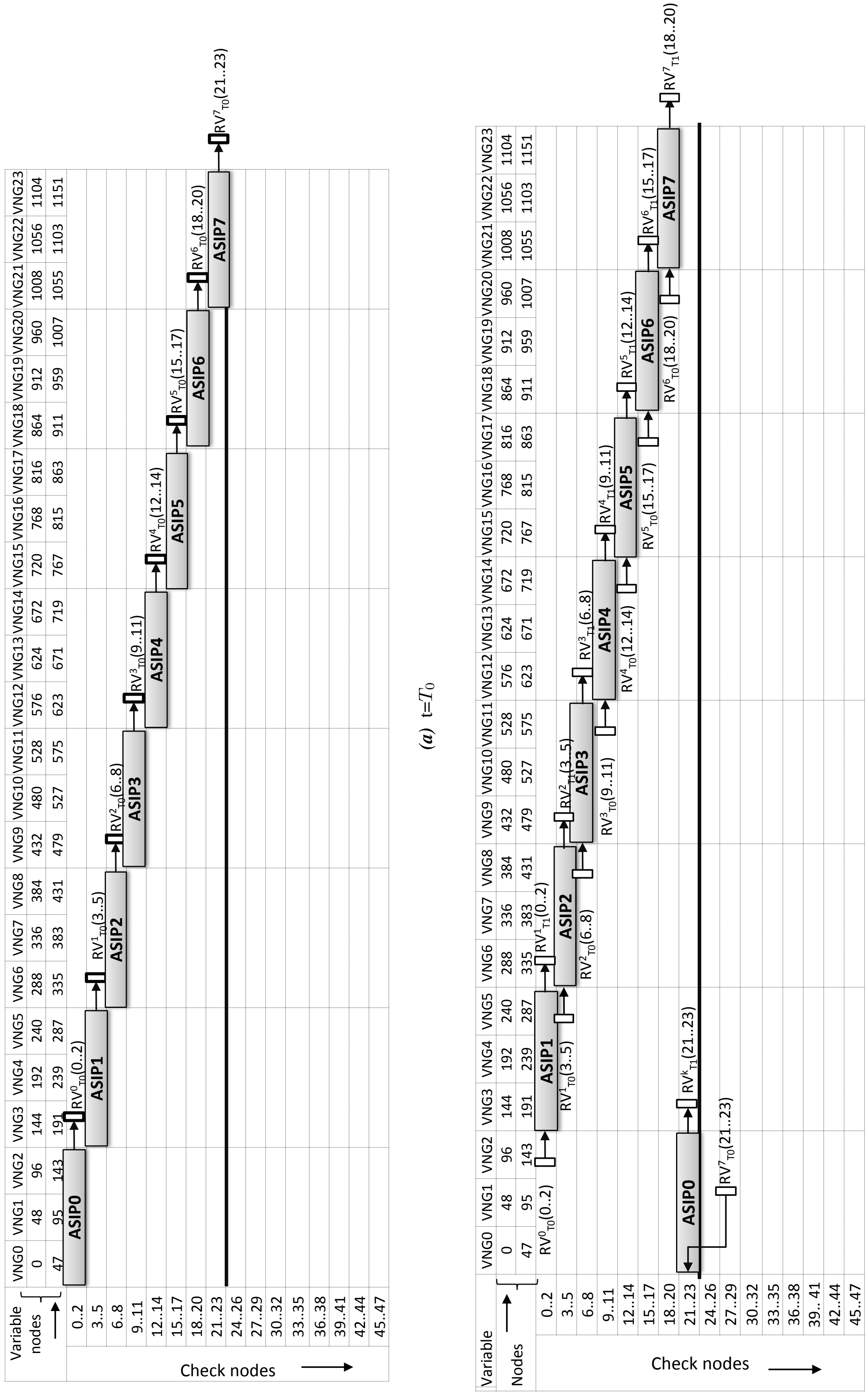
$$T_{sub-iteration} = N_A \times (1_{RV} + (\lceil Z/(3_{CNs} \times N_A) \rceil - 1)_{RVUV} + 1_{UV}) \tag{3.16}$$

where N_A=Number of DecASIPs.

It is to be noted that the LDPC check matrices are more sparse for low code rates (e.g. 1/2, Figure 3.17) than for high code rates (e.g. 5/6, Figure 1.19 in page 28). Thus, the proposed current scheduling scheme implies that the DecASIPs will be processing no check nodes if no check nodes are connected to the VNG in a time step but will only forward the RV and UV messages to the next DecASIP. Thus, the number of time-steps required for one iteration is independent from the check node degree, i.e. the maximum number of edges connected to a check node.

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

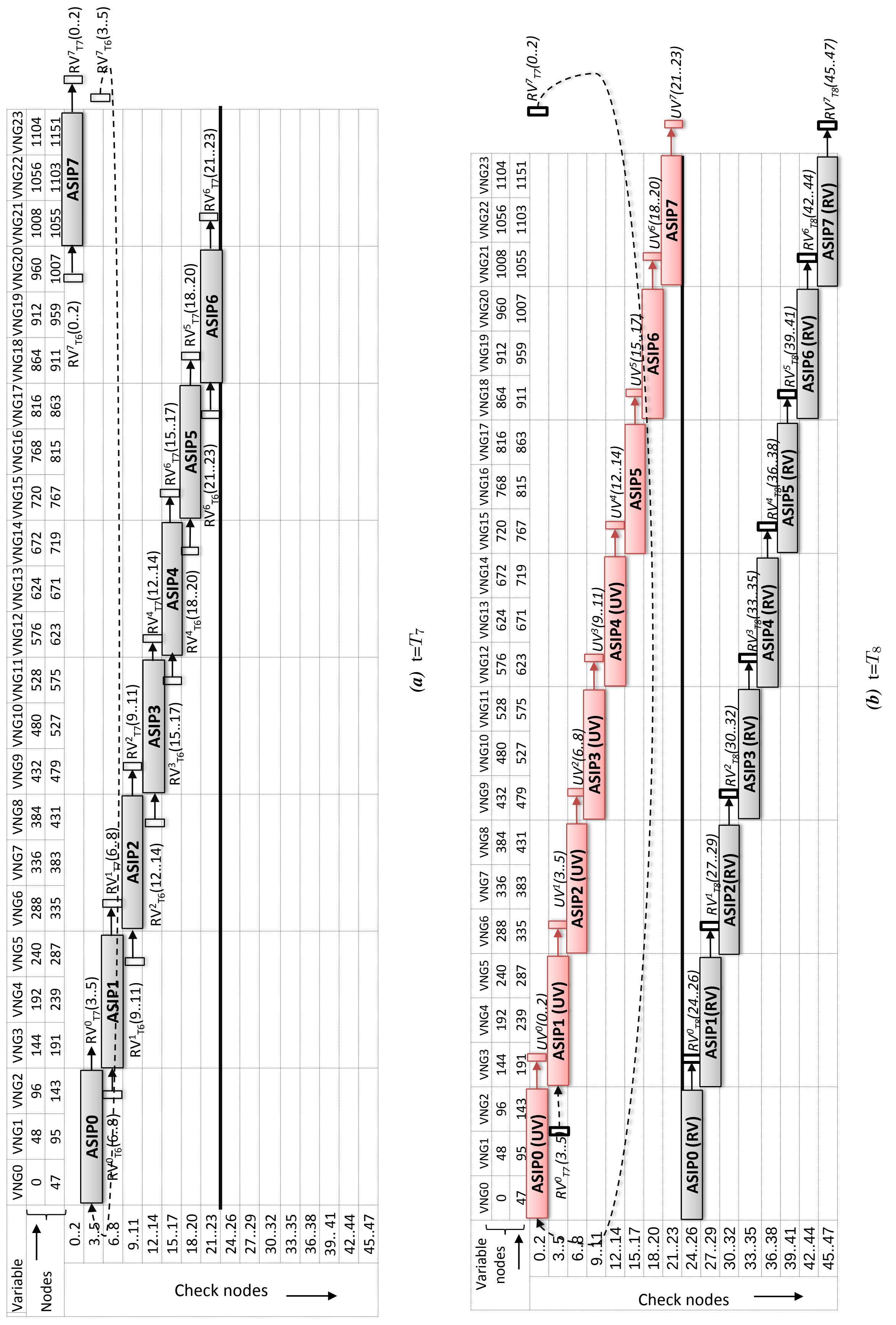
Figure 3.17 — LDPC check matrix H_{base} from the WiFi standard with code rate 1/2, sub-matrix size Z=81, and frame length of 1944 bits. This H_{base} matrix consists of $M_b \times N_b$ permutation sub-matrices of size Z (M_b=12, N_b=24, and Z=81 in this example)



(a) t=T_0

(b) t=T_1

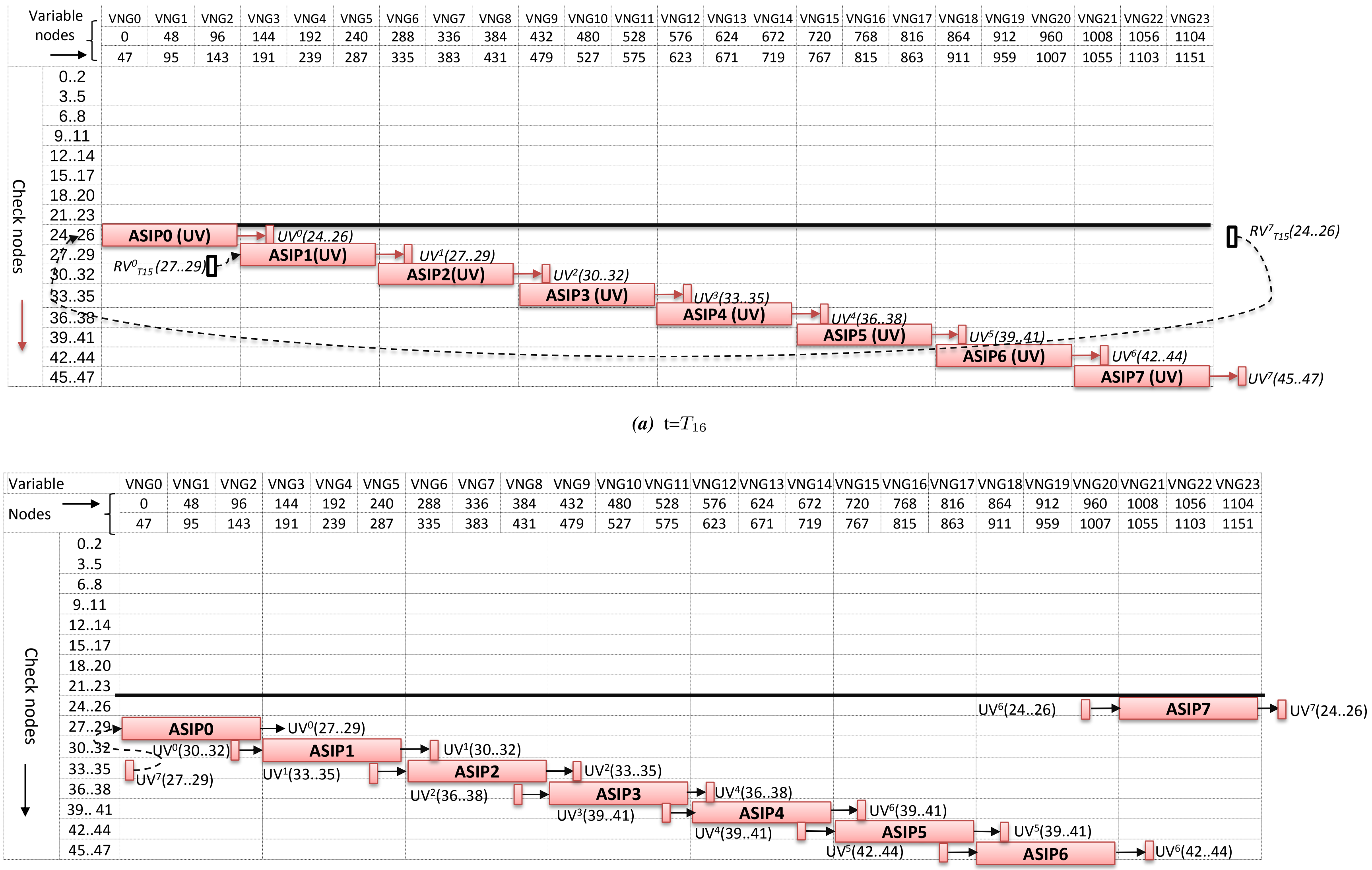
Figure 3.18 — DecASIP$_{v1}$: Proposed LDPC decoding schedule with 8-DecASIP architecture — RV phase


Variable Nodes
Check nodes
VNG0 VNG1 VNG2 VNG3 VNG4 VNG5 VNG6 VNG7 VNG8 VNG9 VNG10 VNG11 VNG12 VNG13 VNG14 VNG15 VNG16 VNG17 VNG18 VNG19 VNG20 VNG21 VNG22 VNG23
0 48 96 144 192 240 288 336 384 432 480 528 576 624 672 720 768 816 864 912 960 1008 1056 1104
47 95 143 191 239 287 335 383 431 479 527 575 623 671 719 767 815 863 911 959 1007 1055 1103 1151
0..2 3..5 6..8 9..11 12..14 15..17 18..20 21..23 24..26 27..29 30..32 33..35 36..38 39..41 42..44 45..47
ASIP0 ASIP1 ASIP2 ASIP3 ASIP4 ASIP5 ASIP6 ASIP7
RV0 T7(3..5) RV1 T7(6..8) RV2 T7(9..11) RV4 T7(12..14) RV6 T7(15..17) RV5 T7(18..20) RV6 T7(21..23) RV7 T7(0..2) RV7 T6(3..5)
RV0 T6(6..8) RV1 T6(9..11) RV2 T6(12..14) RV3 T6(15..17) RV4 T6(18..20) RV6 T6(21..23) RV7 T6(0..2)
ASIP0 (UV) ASIP1 (UV) ASIP2 (UV) ASIP3 (UV) ASIP4 (UV) ASIP5 ASIP6 ASIP7
UV0(0..2) UV1(3..5) UV2(6..8) UV3(9..11) UV4(12..14) UV5(15..17) UV6(18..20) UV7(21..23)
ASIP0 (RV) ASIP1(RV) ASIP2(RV) ASIP3 (RV) ASIP4 (RV) ASIP5 (RV) ASIP6 (RV) ASIP7 (RV)
RV0 T7(3..5) RV7 T7(0..2)
RV0 T8(24..26) RV1 T8(27..29) RV2 T8(30..32) RV3 T8(33..35) RV4 T8(36..38) RV5 T8(39..41) RV6 T8(42..44) RV7 T8(45..47)

(*a*) t=T_7

(*b*) t=T_8

***Figure 3.19* —** DecASIP$_{v1}$: Proposed LDPC decoding schedule with 8-DecASIP architecture — RV phase @t=T_7 and RV+UV phase @t=T_8



(a) t=T_{16}

(b) t=T_{23}

Figure 3.20 — DecASIP$_{v1}$: Proposed LDPC decoding schedule with 8-DecASIP architecture — UV only phase

3.2.3.3 Memory architecture

It has be seen from the bit error rate analysis of Section 3.1.2 that a minimum of 7 bits are needed for the quantization of the input channel LLRs in LDPC mode. Input memories are shared with Turbo mode by storing 2 channel LLRs onto one memory location of the input memory bank as shown in Figure 3.21. Each input memory bank holds the channel values associated with one variable node group. As the maximum value of $Z = 96$ (for WiMAX), the maximum number of valid locations in each input memory bank in the LDPC mode is 48.

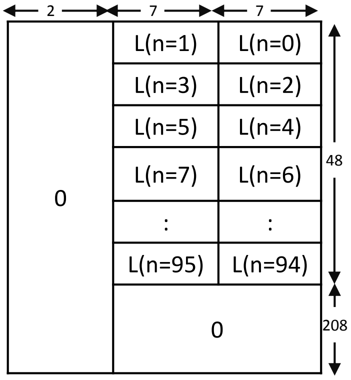


Figure 3.21 — DecASIP$_{v1}$: Input memory bank 0 organization in LDPC mode

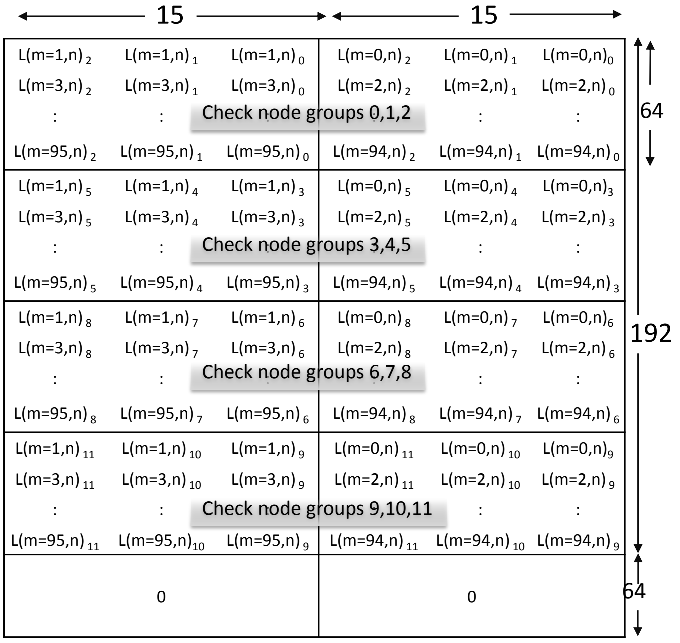


Figure 3.22 — DecASIP$_{v1}$: Extrinsic memory bank 0 organization in LDPC mode

There are 3 extrinsic memory banks per DecASIP that store the extrinsic messages $L(m, n)$, i.e. check nodes to variable nodes messages. As the VN degree is at most 12 (for WiFi standard), there can be at most 12 check node messages $L(m, n)$ for each VN. These messages are stored in th eextrinsic memory in 4 groups, each at an offset of 48 as illustrated in Figure 3.22 for the extrinsic memory bank 0. With this memory organization, the DecASIP$_{v1}$ needs two consecutive memory accesses (1 time step = 2 clock cycles) to process at most 3 check nodes, each associated with 3 edges.

Thus, the complete extrinsic memory bank (30 bits wide) is utilized for extrinsic messages

$L(m,n)$ from 12 check node groups connected to a variable node group. In case that some check nodes are not connected to the variable node group, the corresponding section of the memory bank is left unused.

In fact, the maximum depth of the input and extrinsic memories are constrained by the Turbo mode (LTE standard), whose maximum frame size is 6144 bits. Thus, the proposed memory organization allows for efficient memory sharing with LDPC mode where both WiMAX and WiFi specified LDPC frame lengths are well supported (2304 bits and 1944 bits respectively). Notice that the width of the extrinsic memory sub-banks was extended to 15 bits (which are not fully used in Turbo mode as shown in Figure 3.7, page 64) in order to enable to store 3 $L(m,n)$ messages per address location.

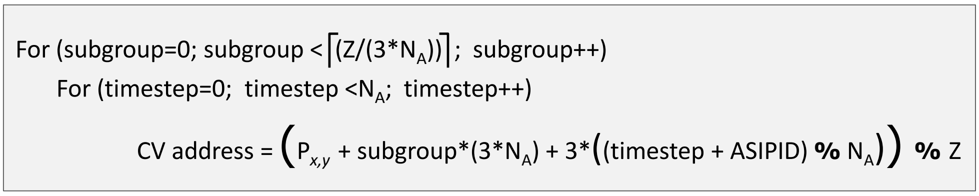
Furthermore, the two 80-bit wide cross memory banks of the Turbo mode (Figure 3.9, page 66) are shared in LDPC mode and used as FIFO's (FIFO1 and FIFO2). FIFO1 is used to buffer the variable node to check node messages $L(n,m)$ (9 messages each of 8 bits per location), while FIFO2 buffers the corresponding input memory bank addresses.

Address generation

Regarding addresses generation, as illustrated in the previous section, with 8 DecASIPs it is possible to process RV or UV phase of $8 \times 3 = 24$ check nodes (=maximum size of the sub-group) in 16 clock cycles. For each bank, the fetch addresses (6 bits each) of the input LLRs are stored in FIFO2 along with the variable node degree (4 bits each). The variable node degree is used during the update phase to regenerate the address for the extrinsic LLRs given as:

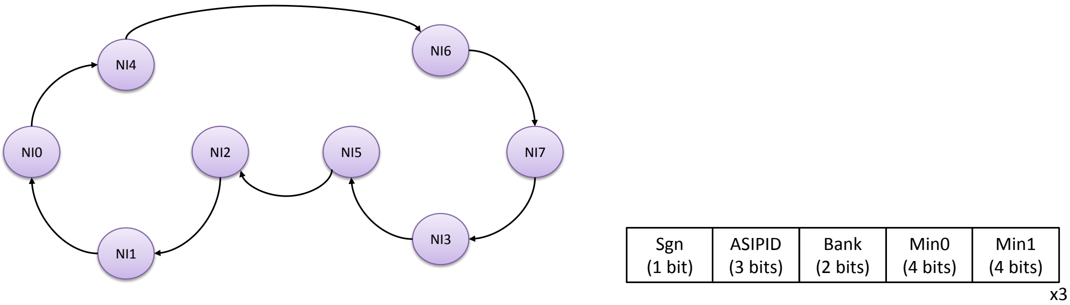
$$\text{Extrinsic Address} = \lfloor(\text{current variable node degree/3})\rfloor \times 48 + \lfloor(\text{CV address/2})\rfloor$$

The following pseudo code describes the generation of CV address sequences for the input channel value memories. $P_{x,y}$ is the permutation value of the VNG as described in the H_{base} matrix. As it can be seen, the implementation of the corresponding logic consists of few counters, modulo operators, and adders.



```
For (subgroup=0; subgroup < $\lceil(Z/(3*N_A))\rceil$; subgroup++)
    For (timestep=0; timestep <$N_A$; timestep++)
        CV address = $\left(P_{x,y} + subgroup*(3*N_A) + 3*\left((timestep + ASIPID) \% N_A\right)\right) \% Z$
```

Figure 3.23 — DecASIP$_{v1}$: Pseudo code for CV address generation of the input memory bank storing VNG$_y$ in LDPC mode



Sgn (1 bit)	ASIPID (3 bits)	Bank (2 bits)	Min0 (4 bits)	Min1 (4 bits)

(a) NoC interconnect in LDPC mode ***(b)*** NoC message

Figure 3.24 — DecASIP$_{v1}$: NoC interconnect and payload in LDPC mode

3.2.3.4 NoC messages

In the LDPC mode, the de-Bruijn NoC is reconfigured to form a unidirectional interconnect of 42 bits wide as shown in Figure 3.24a. The NoC message format is as shown in Figure 3.24b which is used for RV or UV messages sent by the DecASIPs. Notice that there is no addressing field as the message has a predetermined path and has to be routed only to the next DecASIP.

3.2.3.5 Pipeline architecture and assembly Code

Figure 3.25 presents the proposed pipeline architecture of DecASIP$_{v1}$ in the LDPC mode. As for the decoder pipeline in Turbo mode, the first three pipeline stages in LDPC mode are dedicated to instruction fetch and decode. The LDPC address generator block of the *OPF* pipeline stage generates address sequences for the input and extrinsic memories as previously described in Figure 3.23.

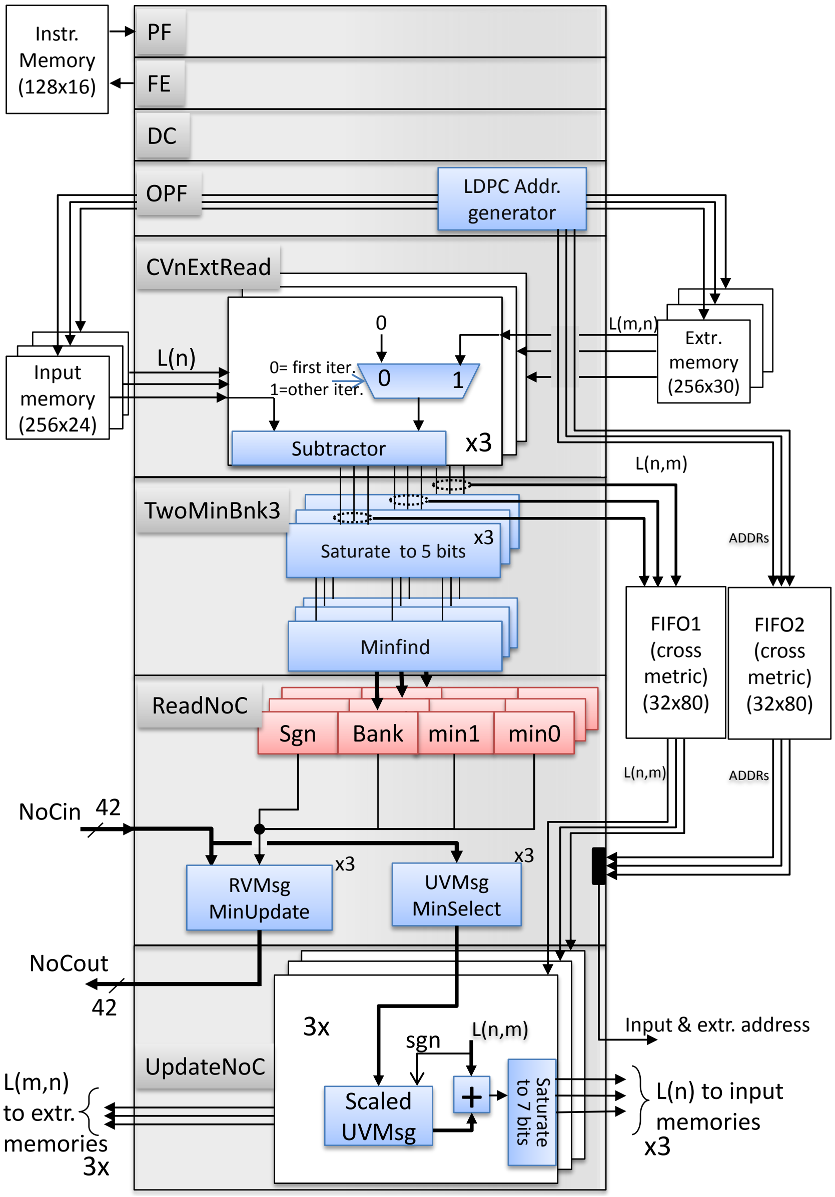


Figure 3.25 — DecASIP$_{v1}$: Pipeline architecture in LDPC mode

The rest of the pipeline stages incorporates the logic to process the instructions *RUNVEC*, *RUNVEC1*, *UPDATEVEC*, and *UPDATEVEC1* described below. The *CVnExtRead* pipeline stage integrates 3 × 3=9 subtractors which are used to calculate the $L(n, m)$ messages. The *TwoMinBnk3* pipeline stage integrates 9 units for saturation of the $L(n, m)$ messages to 5 bit width, in addition to 3 *MinFind* blocks. Each *MinFind* block computes the partial minimums (min0, min1) of the 3 messages $|L(n, m)|$ associated with the check node m. It also computes the sgn bit, which is the XOR of the sign bits of the 3 $L(n, m)$ messages. These $L(n, m)$ messages and the corresponding addresses are buffered in FIFO1 and FIFO2 respectively.

The *ReadNoc* pipeline stage contains the hardware resources (3 *RvMsgMinUpdate* units) to update the incoming RV message read from the *NoCin* bus with the *min0*, *min1*, and *sgn* fields computed in the previous pipeline stage *TwoMinBnk3*. It also integrates 3 *UVMsgMinSelect* units to select the *min0* from the *UV* part related to the current DecASIP. The last pipeline stage (*UpdateNoc*) incorporates 9 adders and 9 saturation units to compute 9 extrinsic messages $L(m, n)$ and to update the channel LLRs $L(n)$.

Assembly code

The DecASIP is first initialized with the parallelism degree (P = N_A × 3), sub-matrix size (Z), and the three columns of permutation values from H_{base} matrix as shown in Listing 3.3. As channel data are stored in couples, two clock cycles are needed to read/write the channel data associated with three check nodes. Thus, a read operation to the input and extrinsic memories is accomplished by *RUNVEC* and *RUNVEC1* instructions (refer to Listing 3.4). The *RUNVEC* instruction does not perform additional tasks, however as the *RUNVEC1* instruction moves through the pipeline stages, it selects the 3 channel LLRs and its associated extrinsic LLRs related to the check nodes under processing and the it calculates the $L(n, m)$ message (*CVExtRead* pipeline stage). In the pipeline stage *TwoMinBnk*, the instruction computes the partial *RV* message associated with the 3 check nodes under process. It also writes the $L(n, m)$ message to the FIFO1 and the associated read addresses to the FIFO2. In the pipeline stage *ReadNoC*, the partial *RV* message is updated with the *RV* message received from the previous DecASIP. Finally, in *UpdateNoc* pipeline stage, the updated *RV* message is forwarded to the next DecASIP. Executing *RUNVEC* and *RUNVEC1* instructions 8 times completes a CN-update phase on one check node sub-group.

Similar to the read operation, the write operation is accomplished by *UPDATEVEC* and *UPDATEVEC1* instructions which activate the last two stages of the pipeline. In *ReadNoC* pipeline stage, the *UPDATEVEC* instruction reads the *UV* message from the NoC along with the $L(n, m)$ message and the associated addresses from FIFO1 and FIFO2 respectively and computes the extrinsic message $L(m, n)$ and the aposteriori channel LLR $L(n)$. In the *UpdateNoC* pipeline stage, the *UV* message is forwarded to the next DecASIP along with writing the $L(m, n)$ message and the aposteriori channel LLR $L(n)$ into the extrinsic and input channel memories respectively.

Listing 3.3 — DecASIP$_{v1}$: Assembly code in LDPC mode. Example with a frame size of 1152 bits (WiMax, Z=48) and 8-DecASIP system decoder – initialization

```
;8 DecASIPs each processing 3 CN =24 at a time
LDPCSIZE PSize,24
;submatrix size
LDPCZSIZE Zsize,48
;rows,NumZerosTriplets
LDPCADDRREGINIT1 1,7
;SubRows=floor(PSize/Zsize),
;num of DecASIPs and RowRem=mod(Zsize,3)
LDPCAddrRegInit2 1,8,1
;set ASIP ID
LDPCASIPID 0
;writing the H matrix offset column 1 row 1 (−1 represented as 127)
LDPCADDRCONFIG1 0,127
writing the H matrix offset column 2 row 1
LDPCADDRCONFIG2 0,46
writing the H matrix offset column 3 row 1
LDPCADDRCONFIG3 0,25
writing the H matrix offset column 1 row 2
LDPCADDRCONFIG1 1,127
writing the H matrix offset column 2 row 2
LDPCADDRCONFIG2 1,27
writing the H matrix offset column 3 row 2
LDPCADDRCONFIG3 1,127
:
:
```

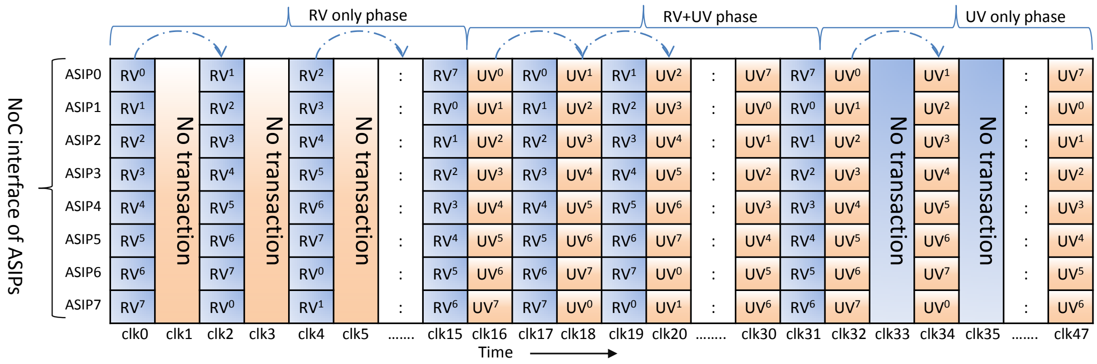
Listing 3.4 — DecASIP$_{v1}$: Assembly code in LDPC mode. Example with a frame size of 1152 bits (WiMAX, Z=48) and 8-DecASIP system decoder – frame decoding

```
Repeat until _ITER for 20 times
PUSH
Repeat until _LOOP0 for 8 times
load and initialize the address generator
LDPCAddrGenInit
RUNVEC
RUNVEC1
_LOOP0:Repeat until _LOOP1 for 8 times
LDPCAddrGenInit
RUNVECWITHUPT
RUNVECWITHUPT1
_LOOP1:Repeat until _LOOP2 for 8 times
LDPCADDRGENINIT
UPDATEVEC
UPDATEVEC1
_LOOP2:POP
;ceil(Zsize/2)=14}
_ITER:Repeat until _DEC for 14 times
NOP
HardDecision
_DEC:NOP
```

NoC scheduling

As mentioned in Section 3.2.3.2, RV and UV phases can be scheduled in parallel as they access different memory locations. In order to allow this operation, two instructions (*RUNVECWITHUPT* and *RUNVECWITHUPT1*) are designed to enable the execution of the VN update phase of the previous check node sub-group in parallel to the RV phase of the current check node sub-group.

Note that the *RV* and *UV* messages are always forwarded to the next DecASIP every two clock cycles and are sent on different time slots (as shown in Figure 3.26).



	clk0	clk1	clk2	clk3	clk4	clk5	....	clk15	clk16	clk17	clk18	clk19	clk20	....	clk30	clk31	clk32	clk33	clk34	clk35	....	clk47
ASIP0	RV^0	No transaction	RV^1	No transaction	RV^2	No transaction	:	RV^7	UV^0	RV^0	UV^1	RV^1	UV^2	:	UV^7	RV^7	UV^0	No transaction	UV^1	No transaction	:	UV^7
ASIP1	RV^1		RV^2		RV^3		:	RV^0	UV^1	RV^1	UV^2	RV^2	UV^3	:	UV^0	RV^0	UV^1		UV^2		:	UV^0
ASIP2	RV^2		RV^3		RV^4		:	RV^1	UV^2	RV^2	UV^3	RV^3	UV^4	:	UV^1	RV^1	UV^2		UV^3		:	UV^1
ASIP3	RV^3		RV^4		RV^5		:	RV^2	UV^3	RV^3	UV^4	RV^4	UV^5	:	UV^2	RV^2	UV^3		UV^4		:	UV^2
ASIP4	RV^4		RV^5		RV^6		:	RV^3	UV^4	RV^4	UV^5	RV^5	UV^6	:	UV^3	RV^3	UV^4		UV^5		:	UV^3
ASIP5	RV^5		RV^6		RV^7		:	RV^4	UV^5	RV^5	UV^6	RV^6	UV^7	:	UV^4	RV^4	UV^5		UV^6		:	UV^4
ASIP6	RV^6		RV^7		RV^0		:	RV^5	UV^6	RV^6	UV^7	RV^7	UV^0	:	UV^5	RV^5	UV^6		UV^7		:	UV^5
ASIP7	RV^7		RV^0		RV^1		:	RV^6	UV^7	RV^7	UV^0	RV^0	UV^1	:	UV^6	RV^6	UV^7		UV^0		:	UV^6

Figure 3.26 **—** DecASIP$_{v1}$: RV and UV messages scheduling with the 8-DecASIP architecture in LDPC mode

3.2.4 ASIC synthesis results

As in the TurbASIP case, DecASIP$_{v1}$ was modeled in LISA language using Processor Designer tool. The generated VHDL description was synthesized with general purpose $65nm$ CMOS technology (worst case 0.9v, 125C) that gave a logic area of $0.087mm^2$ per DecASIP$_{v1}$ with a maximum clock frequency $F_{clk} = 510MHz$. Table 3.2 presents the detailed synthesis results of the combined pipeline stages. The de-Bruijn network with 8 nodes, and a data width tailored to the application need, has an area of $0.15mm^2$. Thus, an 8-DecASIP$_{v1}$ system decoder requires a post-synthesis total area of 2.67 mm^2 (which includes a total memory area of 1.8 mm^2, representing 67%). Table 3.3 gives a summary of the memory bank partitions for a single DecASIP$_{v1}$ in the 8-DecASIP system decoder.

The throughput estimate in LDPC mode is given by the expression (3.17) below, where a sub-iteration is completed in $T_{sub-iteration}$ clock cycles (see equation (3.16)).

$$\text{Throughput in LDPC mode} = \frac{Z * N_b * Crate * F_{clk}}{T_{sub-iteration} * Clk_{CN} * M_b * N_{iter}} \tag{3.17}$$

The best throughput achieved is $306Mbps$ for WiMAX code rate $(Crate) = 5/6$, $Z = 96$, $M_b = 4$ block rows, $N_b = 24$ block columns and $N_{iter} = 10$ iterations. The architecture has $N_A = 8$ DecASIP$_{v1}$ each processing $CN_A = 3$ check nodes per $Clk_{CN} = 2$ clocks.

Similarly, equation (3.18) gives the throughput in Turbo mode. An average $N_{instr} = 4$ instructions are needed to give 1 symbol which is composed of $Bits_{sym} = 2$ bits (lines @26, @27, @30, @32 of the assembly code example given in Listing 3.1). Considering $N_{iter} = 6.5$ iterations, the maximum throughput achieved is $156Mbps$.

$$\text{Throughput in Turbo mode} = \frac{Bits_{sym} * F_{clk} * (N_A/2)}{N_{instr} * N_{iter}} \tag{3.18}$$

Table 3.4 presents the throughput results of DecASIP$_{v1}$ architecture for an 8-DecASIP$_{v1}$ system decoder, in addition the achieved architecture efficiency (using the equation 3.1). The discussion of these results and the comparison with state-of-the-art implementations are regrouped at the end of this chapter, i.e. after the presentation of the enhanced second version DecASIP$_{v2}$ in the following section.

Design unit	Area in um^2
PF	1087
FE	361
DC	5522
OPF	3855
BM1/CVnExtRead	4144
BM2/TwoMinBnk3	3695
EX/ReadNoC	8575
MAX1/UpdateNoC	5075
MAX2	1018
ST	2338
Register File	42338
Memory Interface	5310
Total logic	87233
Total logic for 4x4 system	697871
Total Memories for 4x4 system	1816505
de-Bruijn NoC	152292
Total 4x4 System	2666668

Table 3.2 — DecASIP$_{v1}$: ASIC synthesis results for the complete 8-DecASIP$_{v1}$ system decoder using $65nm$ CMOS technology @510 MHz (worst case 0.9v, 125C)

Memory	Number of banks	Depth	Width
Input(component dec0)	3	256	24
Input(component dec1)	6	256	6 (for S0',S1')
	3	256	12
Extrinsic	6	256	15
Cross metric	2	32	80
Config	1	64	24
Instruction memory	1	128	16

Table 3.3 — DecASIP$_{v1}$: Summary of the memory bank partitions for a single ASIP in the 8-DecASIP system decoder

Mode	Throughput (Mbps)	AE (bits/cycle/iter/mm^2)
DBTC, SBTC	156 @6.5iter	0.76
WiMAX (LDPC)	306[a] @10iter	2.3
WiFi (LDPC)	258[a] @10iter	1.9

[a] Best achieved throughput for code rate 5/6

Table 3.4 — DecASIP$_{v1}$: Throughput results and achieved architecture efficiency for an 8-DecASIP$_{v1}$ system decoder

3.3 DecASIP$_{v2}$

This section presents the proposed enhanced version of the DecASIP$_{v1}$ architecture. This new version of the ASIP is also considered for FPGA prototyping and ASIC integration (presented in chapter 4). The major added design features correspond to: (1) scalability support of 2 to 8 DecASIP decoding system architecture and (2) enhanced throughput in the LDPC mode (around 25%).

The DecASIP$_{v2}$ architecture design targets to achieve a decoder configurations in 1x1, 2x2 or 4x4 modes using 2,4 and 8 DecASIPs respectively. Additionally, in LDPC mode, each DecASIP$_{v2}$ processes 2 check nodes per clock cycle instead of 3 check nodes in 2 clock cycles (as in DecASIP$_{v1}$). Thus, a throughput enhancement of 25% w.r.t. DecASIP$_{v1}$ can be achieved. This feature implies also a different memory partitioning of the input and extrinsic banks.

3.3.1 System architecture

Figure 3.27 presents the system overview of a scaled down version of the architecture with 4-DecASIP$_{v2}$ system decoder. The de-Bruijn NoC in the previous architecture (DecASIP4$_{v1}$) was used to forward NoC messages to adjacent ASIPs in LDPC mode which results in inefficient use of NoC resources. Therefore, this de-Bruijn NoC is replaced by a simpler low complexity butterfly NoC in DecASIP$_{v2}$. This enables efficient forwarding of the NoC messages in Turbo mode. While the LDPC mode NoC messages are forwarded through the α-β buses of the component decoders (multiplexed to form a ring). Furthermore, the decoding operations are simplified by moving all the configuration data from the program memory to a dedicated configuration memory.

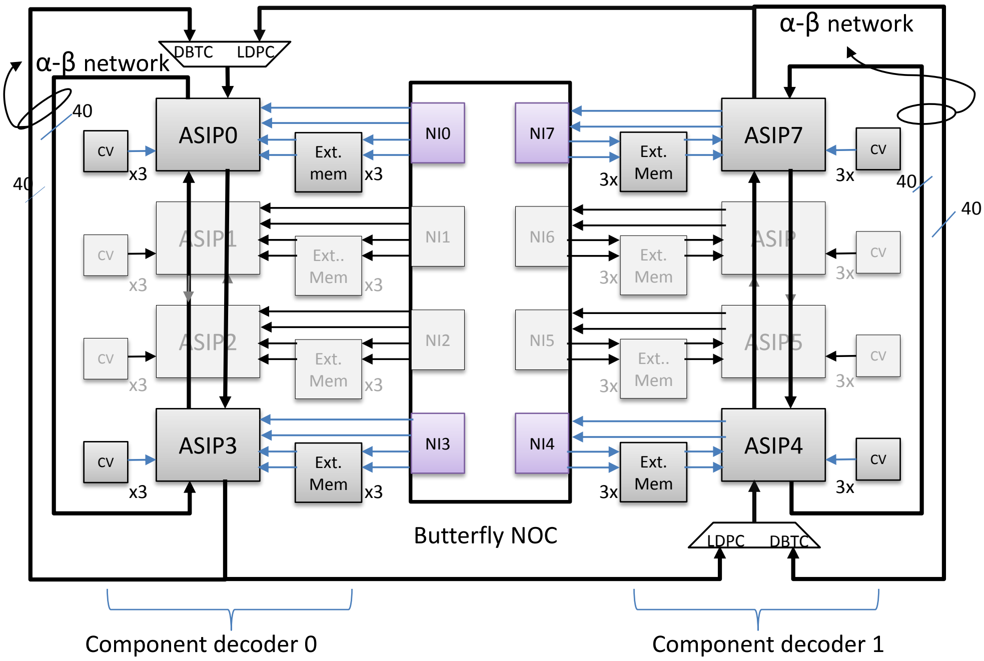


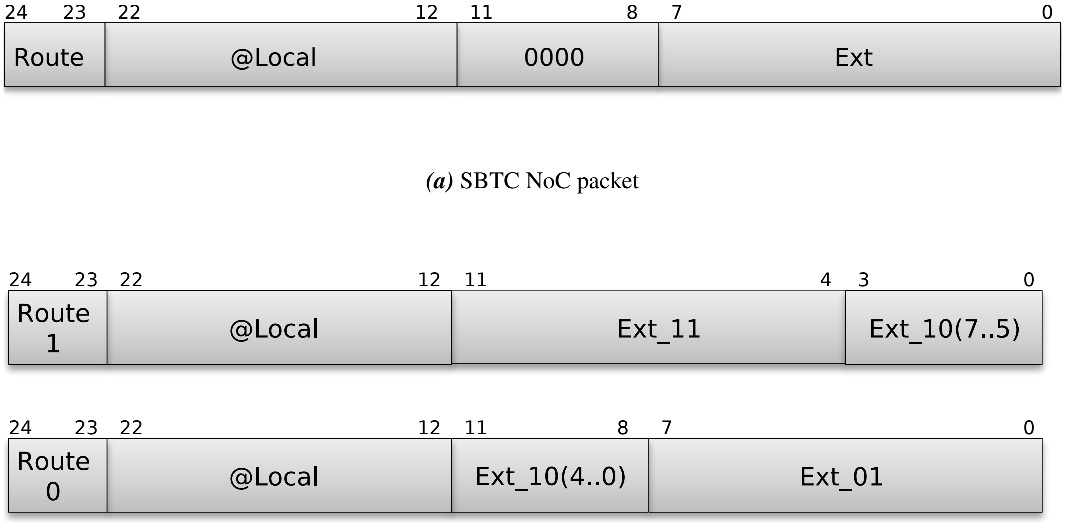
Figure 3.27 — DecASIP$_{v2}$ System Architecture

3.3.2 Turbo mode

As in the DecASIP$_{v1}$ case, shuffled decoding is used Turbo mode in DecASIP$_{v2}$ along with sub-blocking and windowing. Message passing is used for initialization of sub-block state metric boundary. The forward and backward 8 state metric values are exchanged in two clock cycles via two buses connecting the DecASIPs of the component decoders in two clocks cycles (i.e. 4 state metric values in each clock cycle).

In SBTC mode, two NoC packets are generated corresponding to the systematic LLR bits (S0,S1) processed as a symbol (by to radix-4 trellis compression technique). As these two address can be different they packetized over two NoC packets each of width 25 bits each. This does not cause any congestion in the NoC as two packets are generated every two clock cycles (two instructions for the forward recursion) as explained Section 3.2.3.5.

In DBTC mode, the three normalized extrinsic informations saturated to 8 bits are sent across the NoC in two packets as shown in Figure 3.28b. The address fields of these two NoC packets are the same as they are destined to the same location of the other component decoder as shown in Figure 3.28b.



(a) SBTC NoC packet

(b) DBTC NoC packet

Figure 3.28 — DecASIP$_{v2}$: NoC packets format using the Butterfly NoC

Unified SBTC and DBTC assembly code: A common assembly code for SBTC and DBTC mode is as shown in the Listing 3.5 and Listing 3.6. Notice that the three sections of the code in the program memory does not contain any parameters that has to be modified. *NumConfigParam* is the value read from the input pin of the DecASIP. Thus the number of parameters to be read from the configuration memory can be configured as DecASIP is powered up. The number of windows (L) in the sub-block and current window number at line (@9) and (@12) are read from registers initialized from the config memory. Thus reconfiguration of the system is just the rewriting necessary parts of the configuration memory. The exchange of sub-block state boundary metrics (4 at a time) are achieved through the instruction *WINDOW_INIT* with parameters *ALPHABETA_0_3* and *WINDOW_INIT ALPHABETA_4_7*. By initialization of appropriate values from configuration memory into registers during DecASIP$_{v2}$ power up, the presented ASM code can decode the SBTC or DBTC frame. The registers to be initialized are: NWindowReg, NWindowReg1, NWindowReg2, WindowId2, WindowLenReg1, WindowLenReg2, MaxIterationsReg and TailBitAddr. The values for these registers are stored in the configuration memory. Descriptions of these registers are given in section 3.3.4.

Listing 3.5 — DecASIP$_{v2}$: Unified assembly code for DBTC and SBTC modes – initialization

```
 1  ; load configuration data from configuration mem
 2        REPEAT until _INIT for NumConfigParam times
 3     NOP
 4        ASIP_INIT
 5  _INIT: NOP
 6  ; set number of windows to a number greater than
 7  ; actual number of windows without tail bits
 8  ; here, always set to NWindowReg2=NWindowReg+2
 9        SET_WINDOWSN NWindowReg2
10  ;WindowId1 always set to NWindowReg+1
11  ;WindowLenReg1 always set to 1
12        SET_WINDOWSINIT WindowId1, WindowLenReg1
13  ; load the address of the tail bits i.e. TailBitAddr
14        TAILBITS TailBitAddr
15  ; execute line (19) and (20) twice
16     ZOLB _RW,_RW,_RW
17     NOP
18  ;for If window id=0,7 treat the next line as NOP
19        DATA LEFT ADD metric COLUMN2
20  _RW: NOP
21        NOP
22        EXCH_BETA_ALPHA
23        NOP
24        NOP
25        NOP
26  ;load the number of windows NWindowReg1
27        SET_WINDOWSN NWindowReg1
28  ; current windowid, window size
29        SET_WINDOWSINIT WindowId2, WindowLenReg2
```

Listing 3.6 — DecASIP$_{v2}$: Unified assembly code for DBTC and SBTC modes – frame decoding

```
30  ;execute 10 iterations, i.e. (33) to (54) Max. Iterations times
31        Repeat until _LOOP for MaxIterations times
32        NOP
33  ;execute the lines (37) and (38), followed by
34  ;(42) and (43) "Window length" times
35     ZOLB _RW1,_CW1,_LW1
36     NOP
37  _RW1: DATA LEFT ADD metric COLUMN2
38     NOP
39  ;store last window alpha and beta and
40  ;load the next window beta state boundary metrics
```

```
_CW1:  EXCH_BETA_ALPHA
       DATA RIGHT ADD metric COLUMN2
_LW1: EXTCALC ADD info LINE2
;end of sub−block state boundary metrics
       WINDOW_INIT ALPHABETA_0_3
       NOP
       WINDOW_INIT ALPHABETA_4_7
;flush pipeline to start new iteration
       NOP
       NOP
       NOP
       NOP
       NOP
_LOOP: NOP
;finish decoding and halt
       PROC_STOP
```

3.3.3 LDPC mode

The LDPC mode DecASIP functionality is optimized to gain better throughput and resource utilization. The input memory banks are split in a way to store one LLR value in each location. Each DecASIP processes 2 check nodes connected to 3 variable node groups. The RV and UV calculations are made in one clock cycle. Since only four DecASIPs are available RV/UV calculations takes place in 2 sub-phases. In the first sub-phase the DecASIPs process the even variable node groups. The partial RV/UV messages thus obtained are then updated by calculating the RV/UV messages for the odd variable node groups in the second sub-phase. As each ASIP can process two CNs in parallel, 4 DecASIPs in combination process a minimum P=8 CNs (P=maximum parallelism level in LDPC mode=2 $\times N_A$). Each DecASIP has 3 memory banks of size 24×256 used to store the input channel LLR values ($L(n)$) (Figure 3.29). As seen in the figure, each memory bank is internally divided into two sub-banks of 12 bits width. There are 3 other extrinsic memory banks of size 24×256 used for storing check node to variable node extrinsic information ($L(m,n)$) (Figure 3.30). As it is shown, the extrinsic values (each of word length 5 bits) belonging to one VNG, are stored in couples i.e. $L(m,n)$ for $m = 0, 2, ..\lceil Z/2 \rceil$ at bit location (0..4) MemL and $m = 1, 3, ...\lceil Z/2 \rceil$ at the (5..9) of MemU. Subsequent, even and

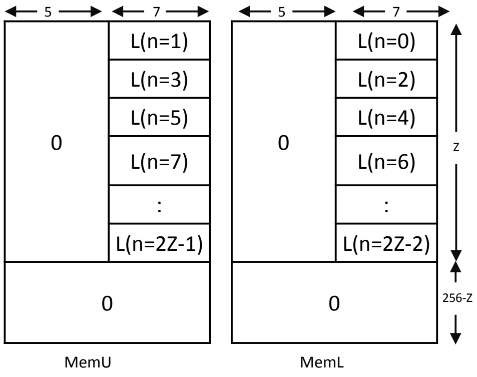


Figure 3.29 — DecASIP$_{v2}$: Input memory bank 0 organization in LDPC mode

odd sequence of extrinsic values generated by the DecASIP are stored in the in the MemL and MemU sub-banks respectively.

The cross metric memories act as FIFO's to store variable node to check node $L(n,m)$, generated input addresses and extrinsic memory offset values. The FIFO's are used to store intermediate values and generated addresses during RV phase and to be used in subsequent UV phase.

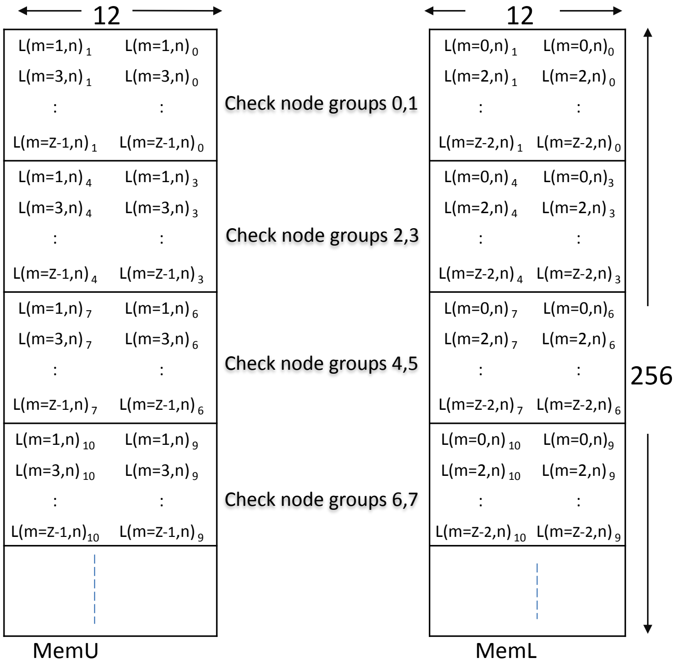

12
12
L(m=1,n)$_1$ L(m=1,n)$_0$
L(m=3,n)$_1$ L(m=3,n)$_0$
L(m=Z-1,n)$_1$ L(m=Z-1,n)$_0$
Check node groups 0,1
L(m=0,n)$_1$ L(m=0,n)$_0$
L(m=2,n)$_1$ L(m=2,n)$_0$
L(m=Z-2,n)$_1$ L(m=Z-2,n)$_0$
L(m=1,n)$_4$ L(m=1,n)$_3$
L(m=3,n)$_4$ L(m=3,n)$_3$
L(m=Z-1,n)$_4$ L(m=Z-1,n)$_3$
Check node groups 2,3
L(m=0,n)$_4$ L(m=0,n)$_3$
L(m=2,n)$_4$ L(m=2,n)$_3$
L(m=Z-2,n)$_4$ L(m=Z-2,n)$_3$
L(m=1,n)$_7$ L(m=1,n)$_6$
L(m=3,n)$_7$ L(m=3,n)$_6$
L(m=Z-1,n)$_7$ L(m=Z-1,n)$_6$
Check node groups 4,5
L(m=0,n)$_7$ L(m=0,n)$_6$
L(m=2,n)$_7$ L(m=2,n)$_6$
L(m=Z-2,n)$_7$ L(m=Z-2,n)$_6$
256
L(m=1,n)$_{10}$ L(m=1,n)$_9$
L(m=3,n)$_{10}$ L(m=3,n)$_9$
L(m=Z-1,n)$_{10}$ L(m=Z-1,n)$_9$
Check node groups 6,7
L(m=0,n)$_{10}$ L(m=0,n)$_9$
L(m=2,n)$_{10}$ L(m=2,n)$_9$
L(m=Z-2,n)$_{10}$ L(m=Z-2,n)$_9$
MemU
MemL

Figure 3.30 — DecASIP$_{v2}$: Extrinsic memory bank 0 organization in LDPC mode

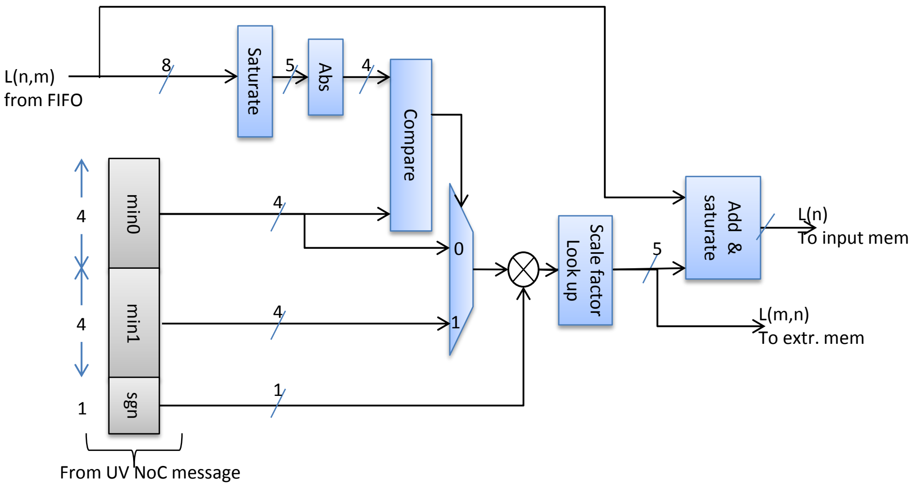


Figure 3.31 — DecASIP$_{v2}$: Variable node update unit in LDPC mode

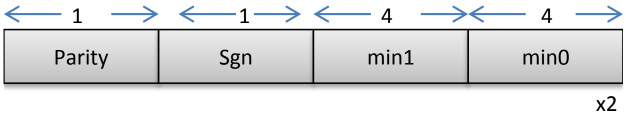
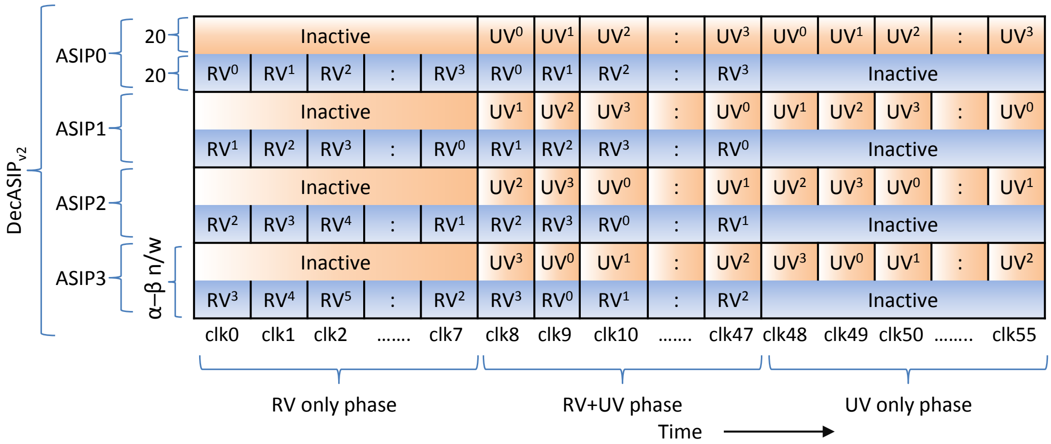


Figure 3.32 — DecASIP$_{v2}$: NoC message format in LDPC mode

3.3.3.1 NoC messages and NoC schedule

In DecASIP$_{v2}$, the RV and UV messages widths are reduced to 20 bits by omitting the location or edge identifier (ASIPID, Ind) present the DecASIP$_{v1}$. During the variable update phase, the variable node that sent the min0 is identified by comparing the min0 of the UV message with the saturated $L(n, m)$ message sent during the RV phase (Figure 3.31). RV and UV NoC messages formed are of the format as shown in Figure 3.32. In addition to $Sgn, min1, min0$ fields, a $Parity$ field is also introduced to enable partity check and early termination.

Figure 3.33 shows the RV and UV messages passed over the NoC interface, i.e. α-β network that connects all four DecASIPs. As it is seen, the RV and UV messages are transmitted over the NoC simultaneously over the 40 bit bus. Each sub-group now consists of 8 check nodes (i.e. 4 DecASIPs with 2 CNs each). Each sub-group is processed in 2 sub-phases (N_{Ph}=2) where in the first sub-phase even numbered VNGs associated with the sub-group are processed, followed by the odd numbered VNGs. Thus, an RV only phase on a sub-group of check nodes needs 8 clock cycles. Similarly, UV or RV+UV phase takes 8 clock cycles. RV and UV messages are sent on lower and upper 20 bits of the α-β network respectively.



	clk0	clk1	clk2	....	clk7	clk8	clk9	clk10	....	clk47	clk48	clk49	clk50	....	clk55
ASIP0	Inactive					UV^0	UV^1	UV^2	:	UV^3	UV^0	UV^1	UV^2	:	UV^3
	RV^0	RV^1	RV^2	:	RV^3	RV^0	RV^1	RV^2	:	RV^3	Inactive				
ASIP1	Inactive					UV^1	UV^2	UV^3	:	UV^0	UV^1	UV^2	UV^3	:	UV^0
	RV^1	RV^2	RV^3	:	RV^0	RV^1	RV^2	RV^3	:	RV^0	Inactive				
ASIP2	Inactive					UV^2	UV^3	UV^0	:	UV^1	UV^2	UV^3	UV^0	:	UV^1
	RV^2	RV^3	RV^4	:	RV^1	RV^2	RV^3	RV^0	:	RV^1	Inactive				
ASIP3	Inactive					UV^3	UV^0	UV^1	:	UV^2	UV^3	UV^0	UV^1	:	UV^2
	RV^3	RV^4	RV^5	:	RV^2	RV^3	RV^0	RV^1	:	RV^2	Inactive				

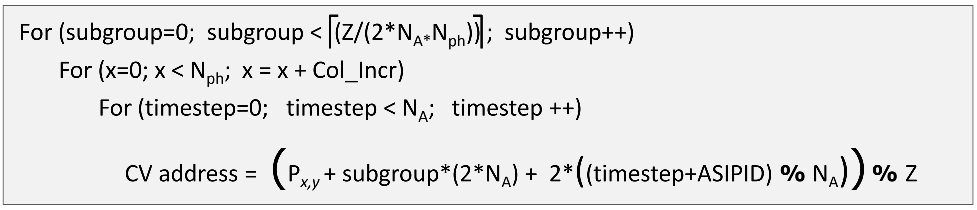
Figure 3.33 — DecASIP$_{v2}$: NoC message passing in LDPC mode. Example for CNG0 processing with Z=48.

Overall, 4 DecASIPs process 8 check nodes in two sub-phases, where each sub-phase takes 4 clock cycles. Thus, the required number of clock cycles to process one complete sub-iteration can be given by the following expression:

$$T_{sub-iteration_{v2}} = N_A \times (1_{RV} + (\lceil Z/(2_{CNs} \times N_A) \rceil - 1)_{RVUV} + 1_{UV}) \times N_{Ph} \qquad (3.19)$$

The address generation scheme presented in section 3.2.3.3 is now modified to support sub-phase decoding schedule as shown in Figure 3.34. Here N_{ph} indicates the number of sub-phases which can be (1,2,4). The actual implementation of the logic presented involves the use of internal address FIFO's to store the $P_{x,y}$+subgroup*(2*N_A) and modulo- counters.

It is to be noted here that though the architecture presented illustrates a scaled down version with 4-DecASIP$_{v2}$, the presented addressing logic facilitates scalability with 8 or 2 ASIPs systems in LDPC and Turbo modes.



```
For (subgroup=0;  subgroup < $\lceil (Z/(2*N_{A}*N_{ph})) \rceil$;  subgroup++)
    For (x=0; x < $N_{ph}$;  x = x + Col_Incr)
        For (timestep=0;   timestep < $N_A$;   timestep ++)

            CV address = $\left(P_{x,y}\right.$ + subgroup*(2*$N_A$) +  2*((timestep+ASIPID) % $N_A$)) % Z
```

Figure 3.34 — DecASIP$_{v2}$: Pseudo code for CV address generation in LDPC mode

3.3.3.2 LDPC assembly code

The assembly code of LDPC decoding is as shown in Listing 3.7, Listing 3.8, and Listing 3.9. The instruction set in LDPC mode mainly consist of 4 instructions:

- RVEC: As in the *RunVec* instruction in DecASIP$_{v1}$, this instruction fetches 6 variable node values corresponding to the 2 check nodes under processing. It calculates the absolute min0 and min1, the partial *sgn* and partial *parity*. Based on these values, it updates the NoC RV received from the adjacent DecASIP and forwards to the next DecASIP.
- UVEC: Similar to the *UpdateVec* of DecASIP$_{v1}$, this instruction calculates the $L(m, n)$ i.e. the update value of th VN of the DecASIP connected to the check node under processing from the UPDT NoC message received from the previous DecASIP and forwards the UV NoC message (unmodified) to the next DecASIP.
- RVEC_UVEC: This instruction is a combination of the above two instructions. By this instruction RV messages are updated for the current sub-group while the variable nodes are updated for the previous sub-group.
- LDPCHardDecCalc: This instruction reads the input memories to give hard decisions (i.e. 7^{th} bit of each sub-bank) in the LDPC mode.

Listing 3.7 — DecASIP$_{v2}$: Assembly code in LDPC mode – initialization RV only phase

```
      Repeat until _INIT for NumConfigParam times
    NOP
       ASIP_INIT
_INIT: LDPCAddr_INIT
; current windowid=0, window size=19
       SET_WINDOWSINIT WindowId1, WindowLenReg1
       REPEAT until _ITER for MaxIterations times
       LDPCAddr_INIT
       PUSH
; execute RV, RV+UV and UV phases for NumRows+1 check node groups
       REPEAT until _LOOP for NumRows times
       NOP
;8 running vector instructions to complete RV calculation
;of 8 check nodes associated with 24 VNGs
```

```
15      RVEC
16    RVEC
17      RVEC
18    RVEC
19      RVEC
20    RVEC
21      RVEC
22    RVEC
```

Listing 3.8 — DecASIP$_{v2}$: Assembly code in LDPC mode – RV+UV phase and UV only phase

```
23  ; execute (27) −(28) 19∗2 times
24  ;∗note instruction @(28) is executed 2 times more than
25  ;the instruction at @27 , hence a total of 40 (RV+UV)
26  ;are executed.
27    ZOLB _CW1,_CW1,_CW1
28      UVEC
29      RVEC_UVEC
30  _CW1: RVEC_UVEC
31  ;execute 8 UV calculations note the first UV only is at line (26)
32      UVEC
33    UVEC
34      UVEC
35    UVEC
36      UVEC
37    UVEC
38      UVEC
39  ;flush the pipeline to start a new group/layer
40      NOP
41      NOP
42      NOP
43      NOP
44      NOP
45      NOP
46      NOP
47  _LOOP: POP
48      NOP
49      NOP
50  _ITER: NOP
```

Listing 3.9 — DecASIP$_{v2}$: Assembly code in LDPC mode – Hard decision

```
51  ;read and generate hard decision for input bank 0 (sub−banks 0,1)
52      Repeat until _HardDec0 for Z times
53      NOP
54      LDPCHardDecCalc 0
55  ;read and generate hard decision for input bank 1 (sub−banks 0,1)
56  _HardDec0:  Repeat until _HardDec1 for Z times
```

```
        NOP
        LDPCHardDecCalc 1
;read and generate hard decision for input bank 2 (sub−banks 0,1)
_HardDec1:  Repeat until _HardDec2 for Z times
        NOP
        LDPCHardDecCalc 2
_HardDec2: nop
;finish decoding and halt
        PROC_STOP
```

3.3.4 Configuration memory

One of the added features in the DecASIP$_{v2}$ is that all the parameters that define the operating mode of the design is moved to a configuration memory. This memory is initialized at the start of the decoding process and enables quick reconfiguration between operating modes. The different words of the configuration memory are as shown in Table 3.5, and the different fields are explained below.

- Mode: This field configures DecASIP in any of the three supported decoding modes: DBTC, SBTC, and LDPC.
- ExtrMemRead: This field controls if the extrinsic memory is to read in the first iteration or not during shuffled decoding in Turbo mode (used for debug purpose).
- C.D.: This field identifies the DecASIP to be in either the part of the component decoder (C.D.) 0 or 1.
- TurboInitIteration: sets the initial number of iteration counter, used for debug.
- State1,State2 : Sets the state values in for Turbo interleave and deinterleave address generation (Figure 3.10), these bits are used to control the load of Seed values at the start of the address generation process.
- StepIndex: Sets the initial value of the counter in for Turbo interleave and deinterleave address generation (Figure 3.10).
- Scaling factor: Turbo mode extrinsic scaling factor.
- NumPhases: Number of phases in the LDPC mode.
- Parity check: Expected parity check bits in the LDPC mode. They can be maximum of 12 bits , i.e. one expected check bit per check node group.
- MaxIterationsReg: Maximum iterations that is to be executed.
- LastWindowSize: Length of the last window in Turbo mode.
- NumRows: Number of rows in the LDPC check matrix.
- zsize: Permutation matrix size of the LDPC check matrix.
- Param0= $N_A - \lceil Z(2 * N_A) \rceil \% N_A$, which indicates the number of empty clock cycles when processing the last subgroup.

- Param1= 2*N_A, Maximum number of check nodes processed in one RV cycle.
- Param2= $\lceil \frac{Z}{2*N_A} \rceil$, number of subgroups, (NumSubGroups).
- $\mathbb{P}_{x,y}$= permutation value of xth row and yth column in the LDPC H$_{base}$ matrix.

Addr	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	-	StepIndex			CD	ExtrMemRead	State		TurboInitIteration				Mode				ASIPId				N_A			
1	-	-	-	-	NumPhases				$\lceil \frac{Z}{(2*N_A)} \rceil$				-	-	-	-	NumRows				-	(Z-1)%2		
2	Param0								Scaling factor								LastWindowSize							
3	Param2								zsize								Param1							
4	-												Parity check											
5	MaxIterationsReg												Tail bit address											
6	NWindowReg1								WindowId1								WindowLenReg1							
7	-								CVLDPCBankDepth								WindowLenReg2							
8	-											Turbo step 0 (Figure 3.10)												
9	-											Turbo step 1 (Figure 3.10)												
:	:																							
:	:																							
14	-											Turbo step 6 (Figure 3.10)												
15	-											Turbo step 7 (Figure 3.10)												
16	-											Turbo seed 0 (Figure 3.10)												
17	-											Turbo seed 1 (Figure 3.10)												
18	-											Turbo/LDPC frame length in Bits (N) (Figure 3.10)												
19	-											Turbo prev step (Figure 3.10)												
24	$\mathbb{P}_{1,1}$								$\mathbb{P}_{1,3}$								$\mathbb{P}_{1,5}$							
25	$\mathbb{P}_{1,2}$								$\mathbb{P}_{1,4}$								$\mathbb{P}_{1,6}$							
26	$\mathbb{P}_{2,1}$								$\mathbb{P}_{2,3}$								$\mathbb{P}_{2,5}$							
27	$\mathbb{P}_{2,2}$								$\mathbb{P}_{2,4}$								$\mathbb{P}_{2,6}$							
:	:								:								:							
:	:								:								:							
48	$\mathbb{P}_{12,2}$								$\mathbb{P}_{12,4}$								$\mathbb{P}_{12,6}$							

Table 3.5 — DecASIP$_{v2}$: Configuration memory contents for DecASIP 0

3.3.5 ASIC synthesis results

Table 3.6 details the post-synthesis results obtained for the DecASIP$_{v2}$ using general purpose 65 nm CMOS technology (510MHz, worst case 0.9v, 125C). The total logic area of one DecASIP$_{v2}$ is 0.126 mm^2. The total area occupied by the 2x2 system is 1.37 mm^2 that includes the area of memories (0.86 mm^2) and the butterfly NoC (0.09 mm^2). Table 3.7 gives a summary of the memory bank partitions for a single DecASIP$_{v2}$ in the 4-DecASIP system decoder.

For an 8-DecASIP$_{v2}$ system architecture, the estimated total area is 2.74 mm^2 achieving a maximum throughput of 437 Mbps and 156 Mbps in LDPC and Turbo modes respectively. The best throughput achieved for the LDPC mode in 8-DecASIP$_{v2}$ architecture is 437 Mbps, when calculated for WiMAX LDPC frames with code rate $Crate = 5/6$, sub-matrix size $Z = 96$, number of rows $M_b = 4$ and number of columns $N_b = 24$ in the check matrix H_{base} (equation 3.20). The number of iterations considered is $N_{iter} = 10$. The required number of clock cycles to process one complete sub-iteration $T_{sub-iteration_{v2}}$ is given by the equation (3.19). The clock speed considered is F_{clk}=$510MHz$.

$$Throughput_{LDPC} = \frac{Z * Crate * F_{clk}}{T_{sub-iteration_{v2}} * (\frac{M_b * N_{iter}}{N_b})} \tag{3.20}$$

As no changes to the architecture have been done for DecASIP$_{v2}$ in Turbo mode, the throughput calculation for the Turbo mode remains the same as in equation 3.18. It can be observed, when comparing with the results of related to DecASIP$_{v1}$ in Table 3.2, that there is a slight increase in area which is attributed to the increase in logic area of DecASIP$_{v2}$. The decode pipeline stage (DC) of DecASIP$_{v2}$ design occupies more area when compared to DecASIP$_{v1}$, this is in fact due to the address decoding logic (Figure 3.34) which is simpler in DecASIP$_{v1}$ (see the pseudo code in Figure 3.23). The complexity in case of DecASIP$_{v2}$ arises from the storage of address counter values in temporary FIFOs when changing from even block column processing (first sub-phase) to odd block column processing (during the second sub-phase). Furthermore, the presented architecture is scalable supporting a decoding system with 2 DecASIP$_{v2}$ (1x1) processing LDPC in 4 sub-phases. Similarly, the EX pipeline stage of this design occupies also more area than in DecASIP$_{v1}$ due to the increased multiplexing complexity introduced by routing LDPC NoC messages through α-β network.

Design unit	Area in um^2
PF	1574
FE	517
DC	12619
OPF	6853
BM1/CVnExtRead	4520
BM2/TwoMinBnk3	4764
EX/ReadNoC	14140
MAX1/UpdateNoC	9790
MAX2	7790
ST	3920
Register file	47398
Memory interface	9645
Total logic	126040
Total logic for 2x2 system	504159
Memories for 2x2 system	864718
Butterfly NoC	9490
Total 2x2 DecASIP$_{v2}$ System	1378367

Table 3.6 — DecASIP$_{v2}$: ASIC synthesis results for the complete 4-DecASIP$_{v2}$ system decoder using $65nm$ CMOS technology @510 MHz (worst case 0.9v, 125C)

Memory	Number of banks	Depth	Width
Input(component dec0)	6	256	12
Input(component dec1)	6	256	6 (for s0',s1')
	3	256	12
Extrinsic	6	256	12
Cross metric	2	32	80
Configuration	1	64	24
Instruction	1	64	16

Table 3.7 — DecASIP$_{v2}$: Summary of the memory bank partitions for a single ASIP in the 4-DecASIP system decoder

Table 3.8 compares the AE obtained for the DecASIP$_{v2}$ to other state of the art architectures and to that of DecASIP$_{v1}$ system (taken from Table 3.4). Comparing with DecASIP$_{v1}$ system, the proposed DecASIP$_{v2}$ system achieves a marginal increase in AE in LDPC mode. In fact, the increased throughput in this mode does not lead to the expected increase in AE. This is due to the increased area caused by the scalability feature added for LDPC mode. On the other hand, in Turbo mode, the throughput remains identical as in DecASIP$_{v1}$ system which leads to lower AE. Furthermore, 8-DecASIP$_{v2}$ (which support maximum of 3072 symbols in Turbo mode) architecture obtains the same AE of 4-DecASIP$_{v2}$ system (which support maximum of 1536 symbols in Turbo mode) as the memory and logic duplicate with number of processors in the system to support larger block sizes.

		DecASIP$_{v2}$ 4 ASIPs	DecASIP$_{v2}$ 8 ASIPs	DecASIP$_{v1}$ 8 ASIPs	[107]	[113]	[110]	[115]
Standard supported		SBTC DBTC LDPC			SBTC, LDPC,	SBTC, CC, DBTC, LDPC	LTE, DVB-SH LDPC	DBTC, SBTC LDPC
Tech (nm)		65			90	65	45	90
Core area (mm^2)		1.37^{f}	2.74	2.6	3.2	0.62	0.9	3.17
$Area_{Norm}$ @65nm (mm2)		1.37	2.74	2.6	1.67	0.62	1.8	1.65
Throughput	SBTC	78 ^{a}	156 ^{a}	156 ^{a}	450 ^{d}	37.2 ^{f}	18 ^{e}	72 ^{e}
(Mbps)	DBTC	78 ^{a}	156 ^{a}	156 ^{a}	-	18.6 ^{f}	18 ^{e}	18 ^{e}
	LDPC	235 ^{b}	437 ^{b}	306 ^{b}	600 ^{b}	237.8 ^{b}	70 @8 iter	93 @6 iter
	LDPC(WiFi)	215 ^{c}	368 ^{c}	258 ^{c}	600 ^{c}	257 ^{c}	100 @8 iter	165 @10 iter
Parallel MAPs		4	8	8	12	1	12	22
Bits/cycle/iter (Turbo)		4	8	8	12	2	12	-
f_{clk} (MHz)		510	510	510	500	400	150	300
AE (bit/cycle/iter/mm^2) (Turbo)		0.72	0.72	0.76	21.5	0.75	2.1	1.12
AE (bit/cycle/iter/mm^2) (LDPC)		3.3	3.12	2.3	5.3	10.3	2.9	3.3

[a] @6.5iter (SBTC/DBTC)

[b] (LDPC-WiMAX, C_{rate}=5/6, Z=96, 10 iter)

[c] (LDPC-WiFi, C_{rate}=5/6, Z=81, 10 iter)

[d] @6iter (SBTC)

[e] (SBTC/DBTC),@8 iter

[f] (SBTC/DBTC),@5 iter

[g] max. frame size is 3072 bits in LTE mode

Table 3.8 — DecASIP$_{v2}$: Results comparison with few recent state-of-the-art implementations

3.3.6 Discussions and analysis of recent related implementations

During the same period of time as this thesis work, several other initiatives and solutions have emerged with the target of supporting flexible channel decoders for Turbo and LDPC decoding. Each of which has different design approaches and targets, besides flexibility, one or a combination of different design objectives, namely: high throughput, optimized area, energy efficiency, scalability.

Table 3.8 summarizes the achieved results after logic synthesis of some recent related implementations. From the available published results, we made a normalization effort to evaluate the impact of our design choices w.r.t. these proposed architectures. The architecture efficiency (Sub-section 3.1.1) is computed, besides the supported flexibility, while power consumption results are omitted as it is out of the scope of this thesis. It has to be noted that besides the architecture efficiency, flexibility and scalability should be considered as design features.

From this table, we can notice the high architecture efficiency AE of 21.5 is achieved in Turbo mode in the work presented in [107]. This is due mainly to the high sub-block parallelism degree exploited in this architecture: 12 parallel Max-Log MAP SISO decoders are instantiated. In fact, this parallelism level does not impact input and extrinsic memories, however only the SISO decoders are duplicated. Using such high degree of sub-block parallelism should also be supported by the interleaving scheme of the target standards. For LDPC mode, the SPC algorithm is adopted and the computational resources are shared with that for the Max-Log MAP algorithm for Turbo decoding. Furthermore, high parallelism degree is exploited in LDPC mode by processing 96 check nodes in parallel using a barrel shifter for high memory bandwidth resulting in AE of 5.3. The proposed architecture, however, does not support DBTC codes of WiMAX. It uses a parameterized architecture model that target only the Turbo codes of LTE standard along with LDPC codes of WiMAX and WiFi. Thus, high AE is obtained with careful choice of target standards, codes, decoding algorithms and allowed high parallelism degree.

Similar design choices and architecture model are used in the work presented in [110]. The high area occupancy can be attributed to the additional support of DVB-SH and 3GPP standards. Low operational clock frequency is set in order to reduce power consumption while just fulfilling the throughput requirements of the target standards (up to 100 Mbps in LDPC mode). However, memory partitioning and interleaving (which seems to be considered as not part of the decoder) are not discussed in the presented architecture.

The single core ASIP-based architecture presented in [113] achieves a high architecture efficiency AE of 10.3 in LDPC mode by processing 27 check nodes in parallel and using NMS algorithm with no computational logic sharing with Turbo decoding mode. It also supports multi-mode operation of any SBTC (up to 16 states) and DBTC (up to 8 states), however it achieves low throughput and architecture efficiency AE of 0.75. Furthermore, the presented single core decoder lacks scalability to enable higher throughputs.

The last work presented in [115] proposes a NoC based multiprocessor highly scalable architecture targeting current and future possible wireless standards supporting LDPC and Turbo codes. The high scalability implies more area and thus the acheiving the AE of 1.12 and 3.3 for Turbo and LDPC modes respectively. Similarly, for the 8-DecASIP$_{v2}$ system, the high scalability feature implies additional area overhead, which results in the AE of 0.72 and 3.12 for Turbo and LDPC decoding modes respectively. Nevertheless, the acheived results are still better than the 8-DecASIP$_{v1}$ system.

Even though a fair comparison cannot be drawn w.r.t. the architecture presented in this chapter, some important conclusions can be derived, which are summarized as follows:

1. High AE is possible by maximising the usage of sub-block parallelism allowed by the standard that specify hardware-aware interleavers.

2. High scalability can be achieved through the use of multiprocessing and NoC based architectures. However, for high AE, a special attention should be paid to keep the communication and the logic overhead to the minimum possible.

3. High throughputs and AE in LDPC mode are possible by increasing the parallelism degree and the memory bandwidth with adequate communication interconnect (e.g. barrel shifters).

4. Both ASIP and parameterized core architecture models are good candidates to achieve high AE as can be noticed from [113] in LDPC mode and from [107] in Turbo mode (Table 3.8).

Experiences in design of DecASIP$_{v1}$ and of DecASIP$_{v2}$ opened up for us following new research directions for further explorations:

- Designing a new optimized flexible Turbo decoder with the following design goals: (1) investigate the maximum attainable architecture efficiency for ASIP-based Turbo decoding when maximising the usage of sub-block parallelism and as the number of parameters in Turbo mode (Table 3.5) is limited (2) investigate the possibility to design application-specific parametrized cores using the available ASIP design flow. The idea of this last aim is to evaluate the benefits from removing the need of a program memory and the related instruction decoder. Related work is presented in Chapter 5.

- Design of a new optimized flexible LDPC decoder with the following design goals: (1) investigate the maximum attainable architecture efficiency for ASIP-based LDPC decoding by increasing the parallelism degree and the memory bandwidth with flexible barrel shifters and (2) explore the possibility of realizing a flexible decoder that allows to implement and validate new/incremental algorithm changes with fast turnaround time in design. Related work is presented in Chapter 6.

3.4 Summary

This chapter has presented our first contributions in the design of scalable, flexible and optimized channel decoder supporting Turbo and LDPC codes using a multi-ASIP NoC based architecture model. Several design goals that were targeted at the starting of this work have been achieved and can be summarized as follows:

1. Resource sharing between the LDPC and the Turbo decoding modes is achieved through efficient reuse of memories and communication network resources. In fact, the computational logic required for the low complexity NMS algorithm (LDPC mode) has relatively low contribution to the overall decoder area and has no direct commonalities with that required for the Max-Log-MAP algorithm (Turbo mode).

2. Scalability is obtained through multi-ASIP architecture connected through an application-specific NoC interconnect. Two NoC architectures were explored: the first one is based on the binary de-Bruijn direct topology which was later replaced by a more area efficient NoC based on the Butterfly indirect topology. Thus, the final system decoder with 4 DecASIP$_{v2}$ can achieve a maximum throughput of 80 Mbps and 235 Mbps for Turbo (DBTC, SBTC) and LDPC modes respectively. When scaled to an 8 DecASIP based system decoder, the proposed architecture allows throughputs in Turbo and LDPC modes of 160 Mbps and 437 Mbps respectively.

3. New LDPC decoding schedule adapted to the base TurbASIP architecture and to the target scalable multi-ASIP channel decoder has been proposed. It implies a ring interconnect network to transfer messages across the multiple DecASIP. The DecASIPs here act as variable and check node processing engines.

4. Compatible with the above architectural choices, possible parallelism techniques have been explored through the use of shuffled decoding and sub-blocking in Turbo mode. In LDPC mode, partial parallelism and layered decoding have been explored where the proposed multi-DecASIP decoder processes disjoint set of check nodes within a check node group.

5. Rapid reconfigurability between the different supported decoding modes is provided by regrouping all the parameters in a well structured configuration memory and by unifying the program memory for both SBTC and DBTC modes.

The proposed scalable multi-ASIP LDPC/Turbo decoder fairs reasonably well when compared to the related state of the art implementations. It achieves a high throughput in LDPC mode with an architecture efficiency of 3.12 bit/cycle/iteration/mm^2. The Turbo mode architecture efficiency is lower achieving a 0.72 bit/cycle/iteration/mm^2, yet the throughputs target of 150 Mbps is met. Finally the chapter concludes with design strategy analysis which serves as motivation for the contributions which are presented in Chapter 5 and 6.

CHAPTER

4 FPGA and ASIC Prototyping of DecASIP

On BOARD validation is a crucial step to fully validate any proposed novel hardware architecture. Two additional benefits can be mentioned in this context: the On board validation can lead to valuable feedbacks to the architecture design particularly regarding system-level interfacing of the channel decoder, additionally the obtained hardware prototype can be used as a rapid simulation environment for digital communication applications; e.g. to explore various system parameter associations.

It is, however, a complex task in the context of ASIP-based multiprocessor implementations and flexible channel decoders, beside the fact that a fully flexible environment should be designed. Hence, this chapter is dedicated to the presentation of our efforts towards FPGA and ASIC prototyping of the proposed flexible channel decoder. A complete FPGA-based prototype of the proposed multi-standard Turbo/LDPC decoder is demonstrated. The functional prototype implements a full communication system including encoder, channel model, ASIP-based decoder and performance counters. All components are flexible and are dynamically configurable through a dedicated GUI (Graphical User Interface). The proposed prototype supports all communication modes defined in LTE, WiFi and WiMAX wireless communication standards.

Furthermore, as a joint effort with another PhD student at the CEA-LETI (Pallavi Reddy), an ASIC integration of the proposed flexible channel decoder has also been elaborated. A 4-DecASIP channel decoder is integrated in the latest Telecom chip (namely MAG3D) designed by the CEA-LETI targeting 4G communication applications.

The chapter is organised as follows: The first section reiterates the communication system model presented in the first chapter drawing parallels to the implemented design units on the target FPGA prototype. Two blocks are detailed in the second section of this chapter: the flexible Turbo and LDPC encoders. The third section briefly recalls the implemented 4-DecASIP system decoder, while the fourth section illustrates the global system controller and the features supported in the control/execution of the DecASIP system decoder. It also presents the other system blocks including the AWGN channel emulator and the proposed GUI interface. The fifth section summarizes the main results obtained from the proposed FPGA prototype. It also presents performance results in terms of BER and FER obtained from the platform and compared against the C-reference curves. Finally, the chapter ends with a summary on the ASIC integration in the MAG3D chip from CEA-LETI.

4.1 Overview of the proposed FPGA 4-DecASIP system prototype

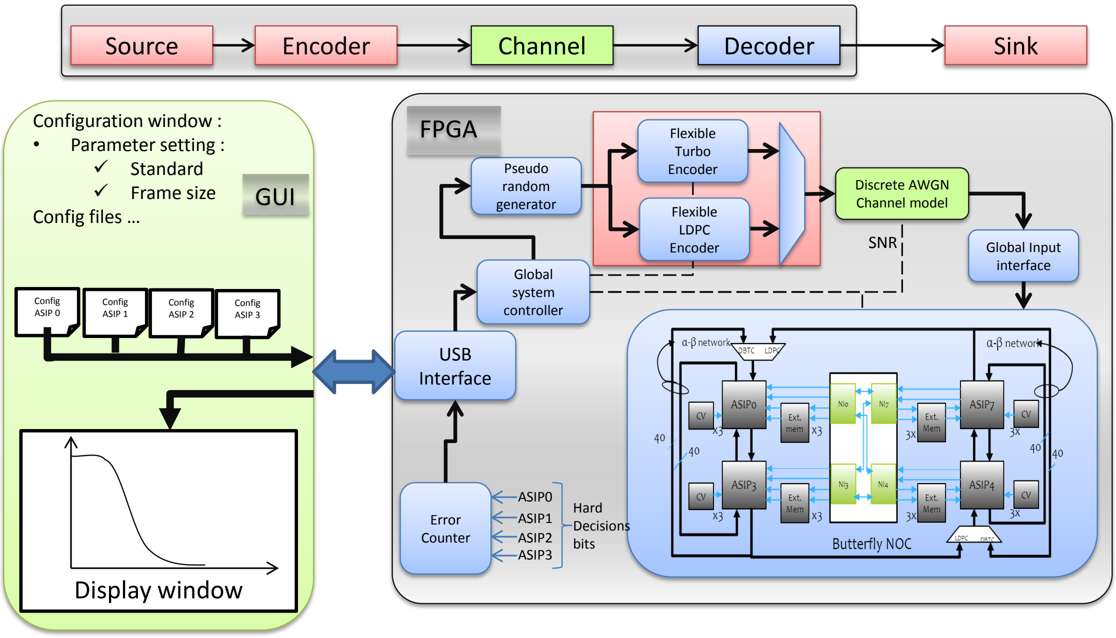


Figure 4.1 — Global FPGA platform architecture overview

On board test 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. Figure 4.1 shows the overview of the implemented prototype on the FPGA, highlighting the hardware blocks that draw parallels to the communication system model (Figure 1.1) presented in Chapter 1.

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 target standards, 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 developed: a source of data, a flexible turbo encoder, a channel model, the Turbo decoder architecture which includes 4-DecASIPs, 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. Furthermore, a flexible LDPC encoder is also implemented in hardware to generate LDPC frames. Both encoder blocks can be configured on-the-fly to encode data blocks of any frame length as specified the target standards. The channel block emulates an AWGN (Additive White Gaussian Noise) model. The decoder is made up of 4-DecASIPs working in shuffled mode, an interleaving address generator, an input interface (to fill the input memories), and a butterfly NoC to manage the exchange of extrinsic information between the 4 DecASIPs.

The FPGA board used for this platform prototype is the DN9000k10pci board from the DiniGroup company. This platform integrates 6 Xilinx Virtex 5 XC5VLX330 of which only one Virtex 5 has been used for the proposed prototype with occupancy of 60%.

Following sections elaborate the implementation of each of the above mentioned blocks in the proposed hardware prototype which is depicted in Figure 4.1.

4.2 Flexible channel encoder

As there is little commonality between the encoding process of Turbo and LDPC codes, we proposed and implemented two separate (yet flexible) encoders.

4.2.1 Flexible Turbo encoder

The flexible Turbo encoder is presented in Figure 4.2. The implemented design supports both SBTC (used for LTE) and DBTC used for WiMAX. It provides a simple flexible architecture to provide both encodings and supports all frame size of LTE and WiMAX standards. To achieve this the convolutional encoder and the interleaver are designed to be flexible. Flexibility is obtained by making the connections ConfigXorBack, ConfigIn, ConfigOutA and ConfigOutB configurable (Figure 4.3). These configurable connections realize adequate XOR gating based on the communication mode. Two instances of this convolutional encoder read the data to be encoded from a dual-port RAM: one in natural sequence and the other in interleaved sequence. When the source completes the generation of the whole frame of data, the memory is read to provide the input to the natural and interleaved encoders at the same time.

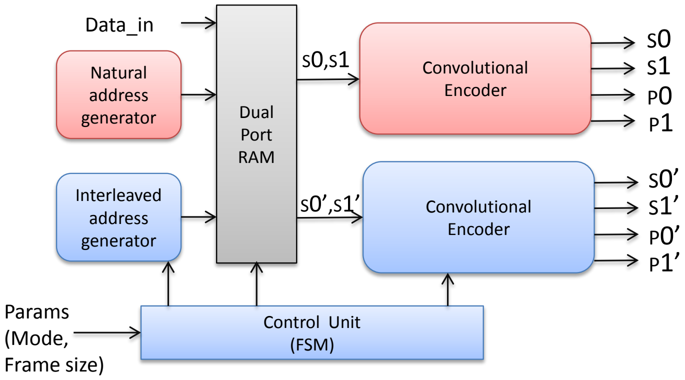


Figure 4.2 — Flexible Turbo Encoder

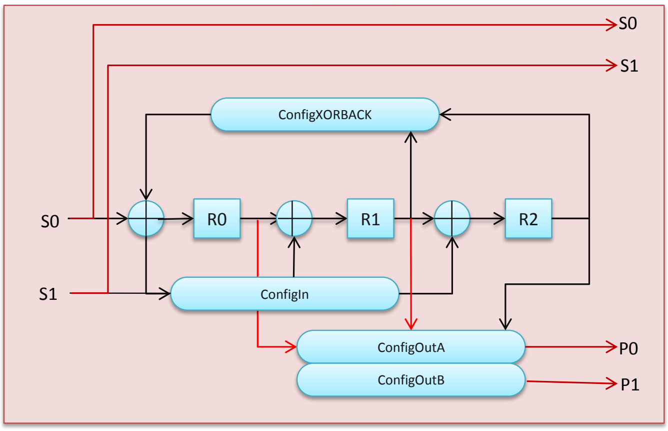


Figure 4.3 — Flexible convolutional Encoder

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) and DBTC (WiMAX) QPP and ARP interleaving address generator based on the work presented in Chapter 3 (see Sub-section 3.2.2.4, page 65) has been used.

For SBTC mode, only the outputs $S0$, $P0$, $S0'$ and $P0'$ are used. $S'1_i$ is used to initialize the memories of the decoder. $S0$ and $S0'$ are also sent to verification module in order to compute BER/FER for the component decoder0 (DecASIP[0,1]) and component decoder1 DecASIP[2,3] (natural and interleaved order processing elements).

The 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 [5], 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.

4.2.2 Flexible LDPC encoder

Compared to Turbo codes, LDPC codes have much higher encoding complexity. However, this complexity is alleviated for the block-structured LDPC codes which are defined in the standards of WiFi and WiMAX. For these codes, the parity check matrix is partitioned into an array of block matrices where each block matrix is either a zero matrix or a right cyclic shift of an identity matrix.

Efficient encoding method based on the parity check matrix H of WiFi LDPC codes has been proposed in [120]. This method enables to calculate the parity bits $\mathbf{p}$ of the codeword from the $\mathbf{m}$ systematic bits through simple equations. In fact, for these codes, the codeword $\mathbf{c_b}$ is divided in groups of Z bits, where Z is the size of the sub-matrix as defined in the standard for each different H matrix and code rate.

$$\mathbf{c_b} = [\mathbf{m}, \mathbf{p}] = [\mathbf{m_o},\ \mathbf{m_1},\ ...,\ \mathbf{m_{k_b-1}},\ \mathbf{p_0},\ \mathbf{p_1},\ ...,\ \mathbf{p_{m_b-1}}] \tag{4.1}$$

With this particular structure of the parity check matrix H, the first block of Z parity bits can be computed through the following expression:

$$\mathbf{p_0} = \sum_{\mathbf{i=0}}^{\mathbf{m_b-1}} \sum_{\mathbf{j=0}}^{\mathbf{k_b-1}} \mathbf{h_{i,j} m_j} \tag{4.2}$$

where $\mathbf{h_{i,j}}$ is a block matrix and $\mathbf{h_{i,j} m_j}$ is a cyclic shift of $\mathbf{m_j}$. By defining $\lambda_\mathbf{i}$ coefficients as follows:

$$\lambda_\mathbf{i} = \sum_{\mathbf{j=0}}^{\mathbf{k_b-1}} \mathbf{h_{i,j} m_j} \quad \text{for } i = 0, ..., m_b - 1 \tag{4.3}$$

the second block of Z parity bits can be given by the expression:

$$\mathbf{p_1} = \lambda_\mathbf{0} + \mathbf{h_{0,k_b} p_0} \tag{4.4}$$

Similarly, for the subsequent blocks of parity bits $\mathbf{p_{i+1}}$ where they can be obtained recursively using $\mathbf{p_i}$ and $\lambda_\mathbf{i}$ [120].

Based on this method, we proposed a parallel and efficient architecture which supports all LDPC code parameters (H matrix type, frame size, code rate) specified in WiFi and WiMAX standards. The proposed architecture (Figure 4.4) is composed of 4 mains units: (1) a memory to store the definition of parity check matrices, (2) a low complexity pipelined unit for the generation of λ_i coefficients, (3) a parallel unit for the generation of parity bits, and (4) a controller unit (Finite State Machine) to manage and schedule the encoding process. An efficient architecture is proposed for the λ_i generator unit which makes use of few counters and cascaded registers instead of using a conventional complex Barrel Shifter.

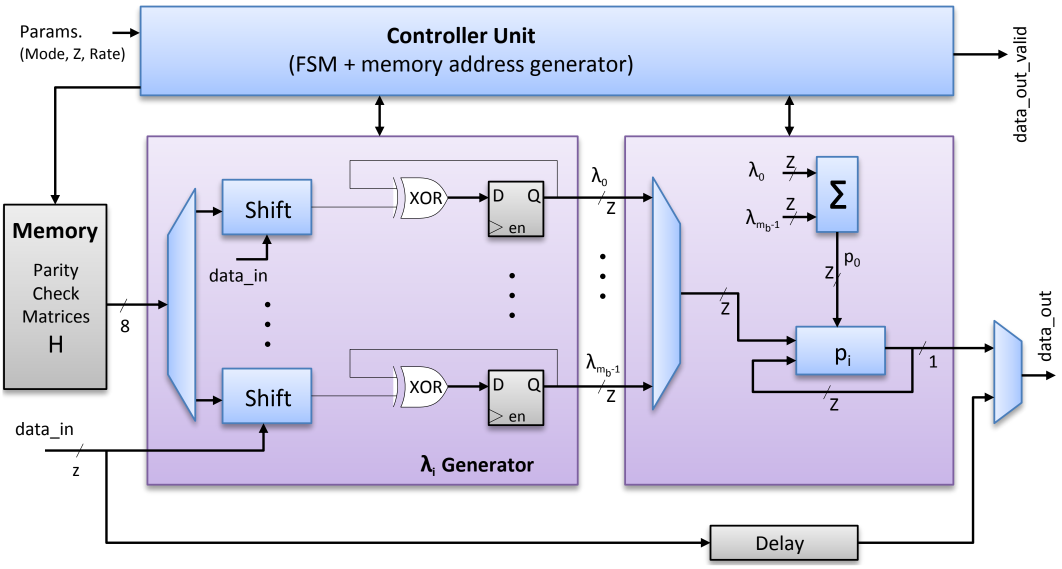


Figure 4.4 — Flexible LDPC Encoder

4.3 Flexible Turbo/LDPC decoder

The channel decoder is the main unit, for which this complete system prototyping has been developed. Its original architecture is based on a multi-ASIP architecture model. We consider in this work an enhanced version of the flexible high-throughput architecture (DecASIP$_{v2}$) presented in previous chapter.

Figure 4.5 gives an overview of the flexible channel decoder architecture with 4 DecASIPs. Besides ASIPs, the decoder architecture integrates several memory banks, configuration and communication networks to handle the extrinsic data exchanges. Three modes are supported namely SBTC, DBTC and LDPC. Each DecASIP is connected to 4 types of memories as given in Table 4.1. All the memories here are synchronous on read and write accesses. They are mapped on Xilinx dedicated block RAMs, allowing better timing and area occupation.

In all modes, appropriate program needs to be loaded into the program memory while the related configurable parameters are loaded into the configuration memory.

4.4 Other blocks of the system prototype

The previous two sections have focused on the channel encoder and decoder, this section groups the presentation of the remaining blocks of the system prototype.

Memory	Number of banks	Depth	Width
Input(component dec0)	6	256	12
Input(component dec1)	6	256	6 (for s0',s1')
	3	256	12
Extrinsic	6	256	12
Cross metric	2	32	80
Config	1	64	24
Instruction memory	1	16	64

Table 4.1 **—** FPGA platform 2x2 DecASIP memory requirements of the banks

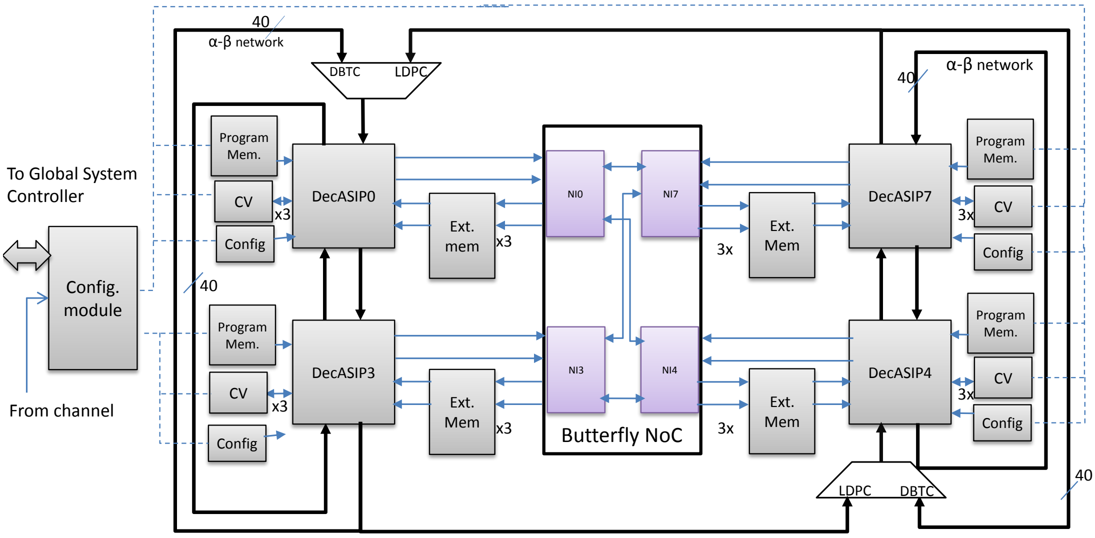


Figure 4.5 **—** DecASIP decoder system architecture

4.4.1 Pseudo random generator

In order to emulate an equi-probable data source, a pseudo random data generator is implemented in hardware using a simple linear feedback shift register (LFSR) of 32 bits. This length ensures that there is no correlation between the data frames during the long simulation runs.

4.4.2 Flexible channel model

The communication channel considered in this prototype is an Additive White Gaussian Noise (AWGN) as shown in Figure 4.6. The hardware implementation used in this prototype emulates the model proposed in [121] and available as a black box at the Electronics Department of Telecom Bretagne. The quantization of the transmitted soft-bits is different for Turbo (6 bits) and LDPC (7 bits) codes. Additionally quantized outputs for 4 bits and 5 bits are also provided for DBTC and SBTC mode. The noise level is represented on 17 bits. The quantization depends on the communication mode. It is automatically selected after the configuration process. The formula to compute the value to apply at the input σ^2 port of the channel module for a target SNR is as follow:

$$\sigma = \sqrt{\frac{10^{\frac{-SNR}{10}}}{2 \times log_2(m) \times R}} \times 2^{16} \tag{4.5}$$

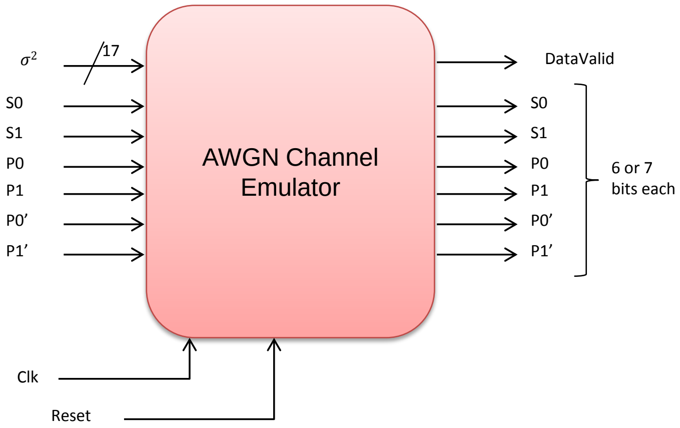


Figure 4.6 — AWGN channel input and output ports

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 $\sigma^2 = 10011000010100011$.

4.4.3 Global input interface

In order to correctly feed the incoming channel LLR to the DecASIP's input memories, a Global Input Interface has been proposed and designed. This flexible block considers the system parameters in terms of communication mode (LDPC, Turbo), frame size, and the interleaving rules. In Turbo mode, DecASIP 0 and 1 process the frame in natural order, while DecASIP 2 and 3 process it in interleaved order. In order to fill the input memories of ASIPs 2 and 3, interleaving addresses are required. Instead of using look-up tables to store interleaving addresses for all supported standards and frame sizes, the recursive address generator presented in figure 3.10 (in 67) is used. As the ring bus connects the ASIPs in this order. For LDPC mode the ring bus connects the ASIPs in the following order: DecASIP0, DecASIP1, DecASIP2 and finally DecASIP3, thus all the input memories are filled sequentially by the same order

4.4.4 Error counter

In order to evaluate error performance, an error counter is added to the FPGA system prototype. This error counter stores the transmitted frame and compares it to the decoded one. To be able to perform long simulations, the error counter is dimensioned to 64 bits. In Turbo mode, the number of errors in the interleaved frame is also calculated for debugging and performance exploration purposes.

4.4.5 Configuration module

The configuration module is used to initialize the platform at the beginning of a simulation. It analyses the configuration data (transmitted through the USB interface) and fills the DecASIP's configuration memories accordingly. In addition, this module initializes the corresponding registers that configure the whole platform. For example, communication mode, frame size, and interleaving seeds used in the global input interface are set through this module. During the configuration, the platform goes to the ”reset” state. When the whole platform is configured, the encoding/decoding processes start.

4.4.6 Global system controller

The Global System Controller module implements the Finite State Machine (FSM) that drives the different modules of the system prototype and performs in sequence the different simulation steps. First, the controller waits for the end of the configuration, initiated by the GUI. Depending on the configuration parameters, the controller starts the Turbo or LDPC encoders and then the corresponding decoding. After the end of the decoding and the bit/frame error counting, the controller starts again the encoding with the same parameters. This controller acts as slave for the GUI. Figure 4.4.6 illustrates the different system states. In addition to the regular scheduling of DecASIPs in shuffled mode (all ASIPs active at the same time) for Turbo mode processing, there is a support to activate ASIPs in sequential mode (run component decoder0 followed by component decoder1) and serial mode (run each ASIP serially). In LDPC mode, the input memories can also be initialized by a text file and is used for debugging.

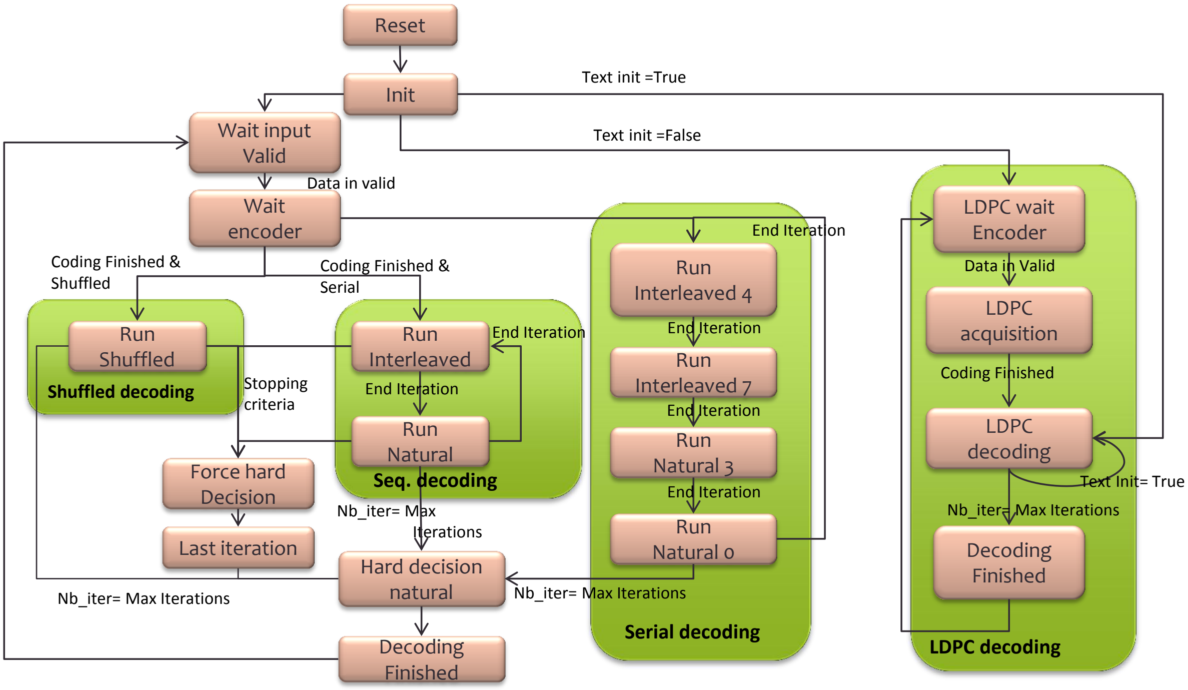


Figure 4.7 — FSM of the global system controller

4.4.7 Graphical User Interface (GUI)

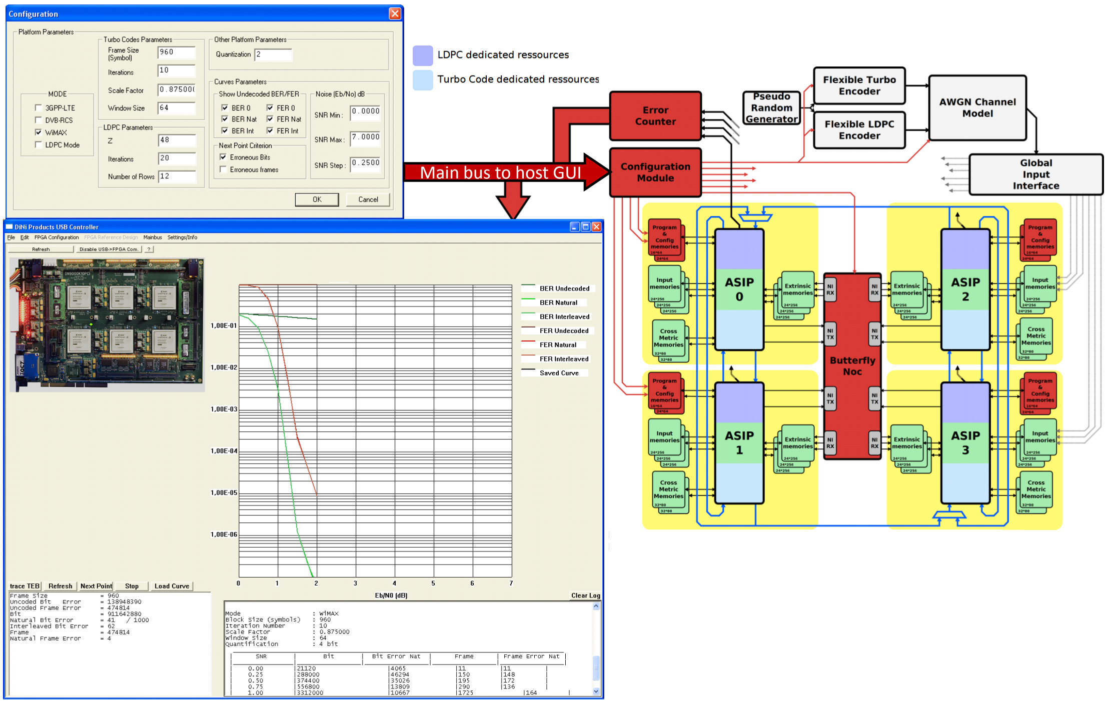
A GUI has been developed to permit the configuration of the platform, launch the simulation, and get the error rate results (Figure 4.8). The proposed GUI consists of two windows: one for the configuration and one for results display on the fly during the simulation. The configuration allows to set the following parameters:

- Communication mode (Turbo: 3GPP-LTE/WiMAX or LDPC WiFi/WiMAX),

- Turbo mode parameters: frame size, window size, code rate, number of iterations, extrinsic scaling factor,
- LDPC mode parameters: frame size, sub-matrix size, code rate, number of iterations,
- Quantization used in the channel model
- Display of different performance curves (BER/FER, natural, interleaved, uncoded)
- SNR range and step values that determine the number of SNR points to be had to plot the error rate curves.

Once all the parameters are validated by the user, the interface generates corresponding configuration text files. Then it sends these configuration files to the FPGA platform through the USB interface.

Furthermore, the GUI reads continuously the values of the error counter from the FPGA. When the number of errors reaches a parameterized threshold, the SNR is incremented by the step defined by the user. When the SNR reaches the limit defined by the user, the simulation is stopped. All the results, in addition to the BER/FER curves plots, are saved to an output text file along with all the simulation parameters. This allows to exploit the results by external tools.



MODE: ☐ 3GPP-LTE ☐ DVB-RCS ☑ WiMAX ☐ LDPC Mode

Turbo Codes Parameters: Frame Size (Symbol) 960; Iterations 10; Scale Factor 0.875000; Window Size 64

LDPC Parameters: Z 48; Iterations 20; Number of Rows 12

Other Platform Parameters: Quantization 2

Show Undecoded BER/FER: ☑ BER 0 ☑ FER 0 ☑ BER Nat ☑ FER Nat ☑ BER Int ☑ FER Int

Next Point Criterion: ☑ Erroneous Bits ☐ Erroneous frames

Noise (Eb/No) dB: SNR Min : 0.0000; SNR Max : 7.0000; SNR Step : 0.2500

```
Frame Size              = 960
Uncoded Bit   Error     = 138948390
Uncoded Frame Error     = 474814
Bit                     = 911642880
Natural Bit Error       = 41   / 1000
Interleaved Bit Error   = 62
Frame                   = 474814
Natural Frame Error     = 4
```

```
Mode                   : WiMAX
Block Size (symbols)   : 960
Iteration Number       : 10
Scale Factor           : 0.875000
Window Size            : 64
Quantification         : 4 bit

 |     SNR     |     Bit      |  Bit Error Nat |     Frame     | Frame Error Nat |
 |_____________|______________|________________|_______________|_________________|
 |    0.00     |21120         |       |4065    |     |11       |11           |
 |    0.25     |288000        |       |46294   |     |150      |148          |
 |    0.50     |374400        |       |35026   |     |195      |172          |
 |    0.75     |556800        |       |13809   |     |290      |136          |
 |    1.00     |3312000       |       |10667   |     |1725     |     |164         |
```

Figure 4.8 — Graphical User Interface (GUI) for the DecASIP FPGA prototype

4.4.8 USB interface

The communication between the FPGA platform (from DiniGroup [122]) and the host computer (running the GUI) is realized through a USB interface. An interface controller module (for the FPGA platform side) and a software driver (for the host computer side) are provided by DiniGroup for this purpose.

4.5 Results of the FPGA prototype

4.5.1 FPGA synthesis results

The FPGA synthesis for Xilinx Virtex-5 (xc5vlx330-1ff1760) device yielded the results shown in Table 4.2. The total design occupies 31497 slices which represent around 60% of the logic slices available on the target device.

Component	Slices	RAM blocks	DSP48E
LDPC/Turbo decoder (2x2 DecASIPs)	21164	82	0
LDPC Encoder	6096	1	0
Turbo Encoder	343	1	0
Channel	2622	7	56
Error Counter	884	4	0
Global system controller	34	0	0
Total (design+Glue Logic)	31497	95	56

***Table 4.2* —** FPGA prototype synthesis on Xilinx Virtex-5 LX330 FPGA

4.5.2 Speed of reconfiguration between different decoding modes

The configuration (or programming) time is defined here as the number of clock cycles needed to write a new program memory contents to the DecASIPs along with writing the configuration memory contents. It should be emphasized that when we are changing between LDPC to another configuration of LDPC (or Turbo to another configuration of Turbo), only few parameters need to be updated in the configuration memory while the program memory content remains the same. Table 4.3 gives an estimate of the number of clock cycles needed to change the decoding mode in the proposed prototype. We can see that the decoding mode can be changed with at most 167 clock cycles. The reinitialization of program memory to LDPC mode take 51 (length of the assembly code in LDPC mode) clock cycles. Similarly, reinitialization the program memory to Turbo mode takes 37 clock cycles.

From mode	To mode	Clock cycles needed
Turbo	Turbo	17*4 = 68
Turbo	LDPC	51+(5+24)*4 = 167
LDPC	LDPC	(5+24)*4 = 116
LDPC	Turbo	37+(17*4) = 105

***Table 4.3* —** Required number of cycles to change the decoding mode for the 4-DecASIP system

4.5.3 Scalability and throughput

The architecture presented here has four DecASIPs operating in 2x2 mode for Turbo decoding and 4 DecASIPs connected to a unidirectional ring network for LDPC decoding. The DecASIPs can also be configured to 1x1 mode for lower throughput and/or shorter frames. The number of available memory blocks depends on the number of DecASIPs in the system. The frame sizes supported in different modes are as shown in Table 4.4. It has to be noted that for the system with 4 or 2 DecASIPs in LDPC mode, the extrinsic memory depth for each DecASIP must be large enough to store all the check to variable node messages processed by it.

The throughput estimate for LDPC mode is given by equation (4.6). The best throughput achieved for the presented configuration is $36.9\ Mbps$ for WiMAX with code rate $Crate = 5/6$,

Number of DecASIPs	Throughput		Frame size in bits	
	Turbo @ 9 iterations	LDPC @ 10 iterations	Turbo	LDPC
8	17.7	68.5	6144	576,628..2048
4	8.88	36.9	3072	576,628..2048
2	4.44	19.2	1536	768

Table 4.4 — Scalability and throughput achieved for FPGA prototype implementation and operating frequency of 80 MHz

sub-matrix size $Z = 96$, number of rows $M_b = 4$ and number of columns $N_b = 24$ in the check matrix H_{base}. The number of iterations considered is $N_{iter} = 10$. The required number of clock cycles to process one complete sub-iteration $T_{sub-iteration_{v2}}$ is given by the equation (3.19) in page 92. The clock speed considered is F_{clk}=80MHz which is the maximum frequency achieved for the FPGA implementation.

$$Throughput_{LDPC} = \frac{Z * Crate * F_{clk}}{T_{sub-iteration_{v2}} * (\frac{M_b * N_{iter}}{N_b})} \tag{4.6}$$

Similarly, equation (4.7) gives the throughput in Turbo mode. An average $N_{instr} = 4$ instructions are needed to give a symbol which is composed of $Bits_{sym} = 2$ bits. Considering $N_{iter} = 9$ iterations, the maximum throughput achieved is 8.88 $Mbps$.

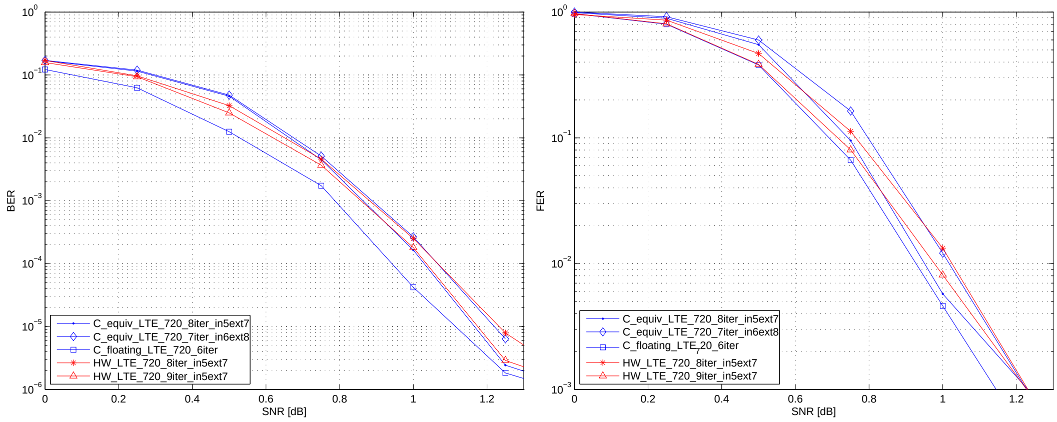
$$Throughput_{Turbo} = \frac{Bits_{sym} * F_{clk} * (N_A/2)}{N_{instr} * N_{iter}} \tag{4.7}$$

4.5.4 Performance results

Using the proposed complete system prototype we are able to evaluate the error performance on hardware for any communication mode of the supported standards. Figures 4.9, 4.10, and 4.11 show the BER and FER curves obtained after FPGA evaluation along with corresponding software model reference curves. For SBTC mode, Figure 4.9 presents the performance results for LTE standard using frames of length 1440 bits encoded with 1/3 code rate. For DBTC mode, Figure 4.10 considers the WiMAX standard with frames of 960 symbols and a code rate of 1/3. For LDPC mode, Figure 4.11 shows the performance results obtained for WiMAX and WiFi frames of sub-matrix size Z=48 and Z=54 with a code rate of 1/2. These results illustrate how the prototyped flexible communication system is correcting transmission errors as expected with acceptable performance degradation of less than 0.20 dB w.r.t. to the reference C-simulation curves.

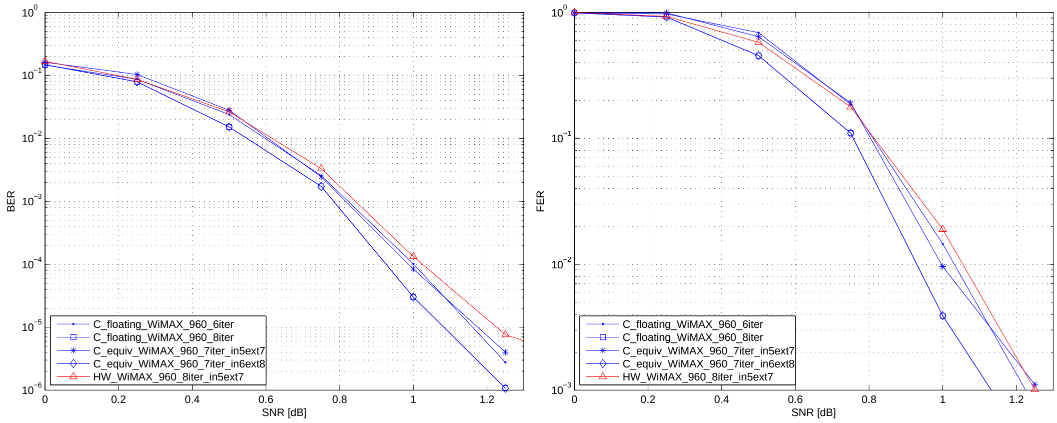
4.6 ASIC integration of DecASIP

The FPGA prototyped 4 DecASIP system was in addition integrated within the digital baseband platform MAG3D designed by the CEA-LETI. In this section we will introduce briefly the MAG3D platform; followed by the integration details and results in terms of area and performance. This ASIC platform is still under test at the time of this manuscript. This work was done as a joint effort with another PhD student at the CEA-LETI: Pallavi Reddy.



(a) BER results ***(b)*** FER results

Figure 4.9 — FPGA prototype BER and FER results in SBTC mode for LTE frame size of 1440 bits and code rate of 1/3

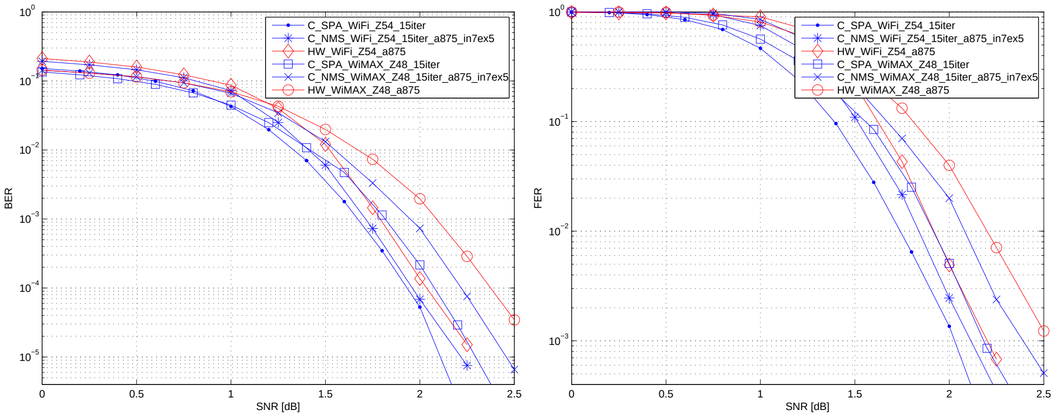


(a) BER results ***(b)*** FER results

Figure 4.10 — FPGA prototype BER and FER results in DBTC mode for WiMAX frame size of 1920 bits and code rate of 1/3

4.6.1 MAG3D chip from CEA-LETI

The MAG3D MPSoC chip intends to map multiple digital baseband IPs of complex telecommunication protocols onto a single 3D chip. As a 4G digital baseband chip, it is aimed at leveraging several objectives, both in terms of target application and system design. At the application level, the main objective is to support flexible baseband processing needed by the current and future telecommunication standards namely, 3GPP-LTE and its potential evolutions, with a possibility to support other standards based on OFDM techniques. Wherein, throughputs up to Gbits/s are achieved through support of multi antennas schemes (MIMO) with advanced features such as Hybrid Automatic Repeat reQuest (H-ARQ). Additionally, the aim is to support quick reconfiguration between different protocols/applications with autonomous reconfiguration for the different IPs on MAG3D. The idea here is to share resources between two or more protocols, using the same baseband processor for applications like Cognitive Radio (CR). At the system design level, the objective was to demonstrate a communication efficient interconnect architecture based on the Globally Asynchronous Locally Synchronous (GALS) infrastructure. MAG3D is composed of different IP blocks which communicate via a mesh-based NoC as shown in Figure



(a) BER results ***(b)*** FER results

Figure 4.11 — FPGA prototype BER and FER results in LDPC mode for WiMAX Z=48 and WiFi Z=54 and code rate of 1/2

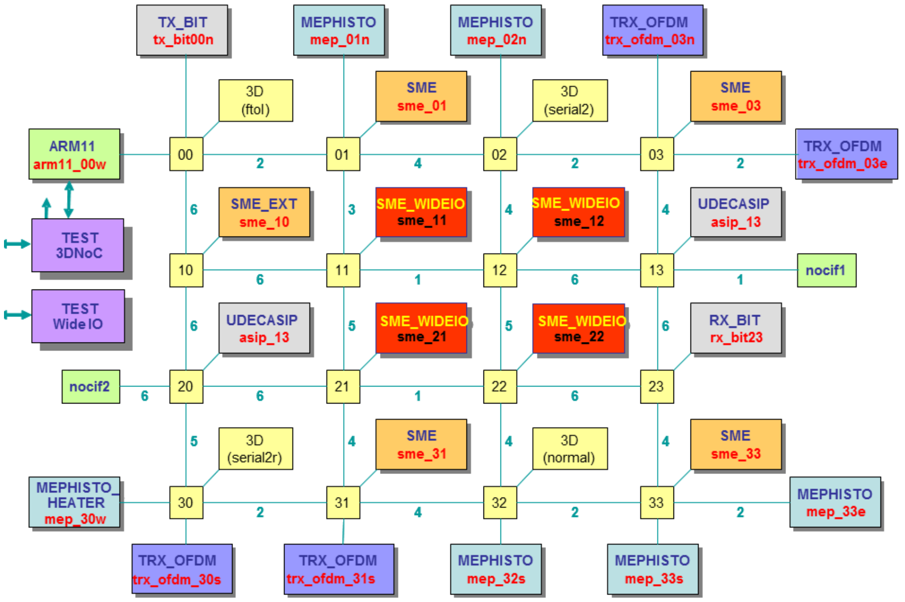


Figure 4.12 — MAG3D system architecture

4.12. The chip is semi-heterogeneous, i.e. all the IP blocks are not identical and multiple copies of the same IP are available and used depending on the application needs. The design contains two main kinds of resources:

1. Coarse-grain reconfigurable cores: these are either DSP for data processing functions (MEPHISTO) or reconfigurable memory resources for data storage and management (Smart Memory Engine - SME).

2. Specific reconfigurable IP cores: these are specialized reconfigurable IP cores that directly support dedicated functions such as OFDM (TRXOFDM), bit operations (RX_BIT, TX_BIT), channel decoding algorithms (DecASIP).

4.6.2 Integration constraints

As mentioned before, the specification of the DecASIP system is mainly defined as per the performance and cost needs of MAG3D MPSoC platform (Figure 4.13). The available space on the platform is shown in the green zone on which the channel decoder system has to be implemented. The total area available on the MAG3D plan was 1.3mm^2 + 1.2 mm^2. Hence, in order to adhere to this area constraint, 4 DecASIPs are considered taking into account the possible increase in area after the place and route. Thus two modules each containing 4 DecASIPs are implemented in MAG3D.

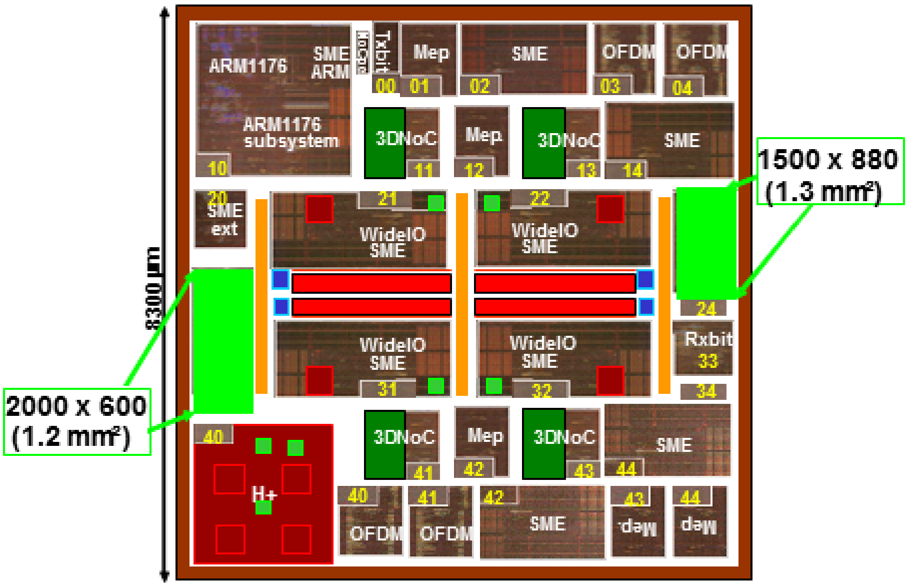


Figure 4.13 — Placement of DecASIP$_{v2}$ system on MAG3D

A network interface for the DecASIP system is developed to enable its integration with the MAG3D platform as shown in Figure 4.14. This interface includes a set of configuration, data flow and service (test mode) signals.

4.6.3 ASIC integration results

Synthesis has been done with CMOS 65nm low power technology (worst case, 1.10V, 105C). The total area of MAG3D MPSoC platform is 8500μm x 8500μm (72mm^2). The total area of the 4 DecASIP system is 1.47 mm^2 without the Globally Asynchronous Locally Synchronous (GALS) interface. Back-End area with GALS interface, placement and routing is 2.22 mm^2, where the area of GALS interface accounts for 0.02 mm^2. The placement density is 66%. The resulting chip diagram of the implemented design is as shown in Figure 4.15. Memories, combinational and sequential circuits occupy 55%, 20% and 25% of the total area dedicated for the DecASIP system respectively. The critical path for this design is 2.8 ns with asynchronous NoC working at 500 MHz. In this joint effort with the PhD student at CEA-LETI, power reduction techniques have been proposed and integrated, at different levels (algorithmic, architecture, and technology levels).

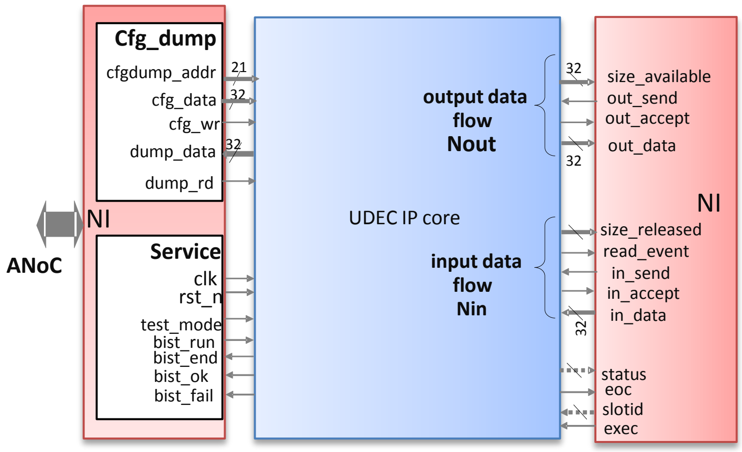


Figure 4.14 — 4 DecASIP system: Network interface to GALS on MAG3D

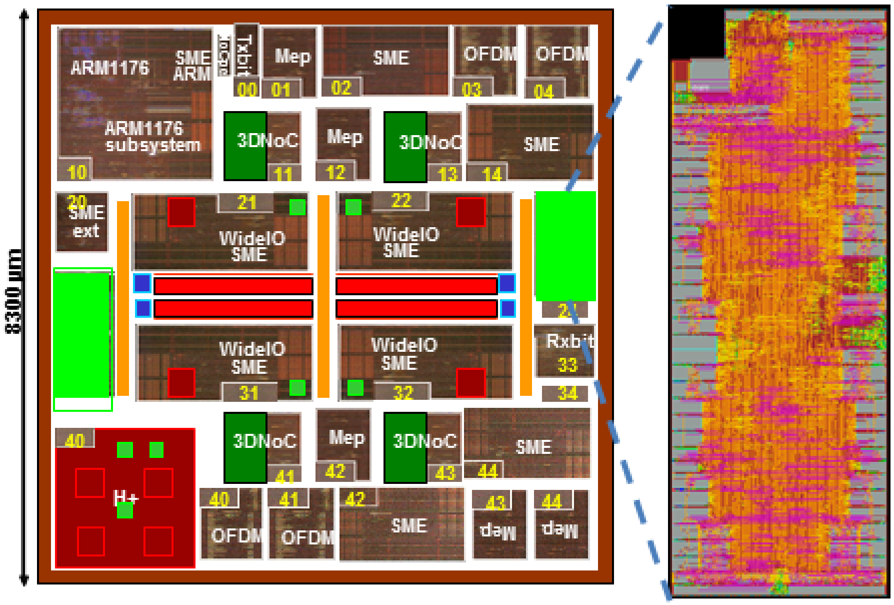


Figure 4.15 — Post-route DecASIP chip diagram on MAG3D

4.7 Summary

In this chapter, we have presented an FPGA prototype of a full communication system targeting the support of multi-standard Turbo/LDPC Encoding and Decoding. To our knowledge, this is the first demonstrated multi-ASIP NoC-based FPGA prototype of such flexible channel decoder supporting LDPC and Turbo (SBTC and DBTC) codes. The proposed functional prototype illustrates the effectiveness of ASIP concept in the implementation of scalable, flexible and optimized multi-standard platforms for wireless communications. The prototype supports all communication modes defined in LTE, WiFi, and WiMAX standards. To enable reasonable throughputs on FPGA most of the system components (including emitter, channel model, decoder, and system controller) have been implemented in hardware. A GUI enables the reconfiguration of the decoding mode on-the-fly and enables the monitoring of the communication system through a user friendly interface. Besides its wide flexibility, the proposed FPGA prototype achieves a throughput of 11.4 Mbps in Turbo modes and 36.5 Mbps in LDPC modes when operating at a clock frequency of 80 MHz.

Furthermore, as a joint effort with the PhD student at the CEA-LETI (Pallavi Reddy), an ASIC integration of the proposed flexible channel decoder has been presented. A 4-DecASIP channel decoder is integrated in the latest Telecom chip (namely MAG3D) designed by the CEA-LETI targeting 4G communication applications. Moreover, the ASIC integrated channel decoder incorporates power reduction techniques which have been proposed by the student at different levels (algorithmic, architecture and technology levels).

CHAPTER

5 TDecASIP: Parameterized Turbo Decoder

In THIS chapter we present a new optimized flexible standalone Turbo decoder, namely TDecASIP. The objective behind the conducted new design is twofold: (1) investigate the maximum attainable architecture efficiency for ASIP-based Turbo decoding, and related to this first objective (2) investigate the possibility to design application-specific parametrized cores using the available ASIP design flow. The idea of this last objective is to evaluate the benefits from removing the need of a program memory and the related instruction decoder.

Keeping in mind these goals, TDecASIP architecture is proposed and designed as a parametrized Turbo decoder. The proposed architecture exhibits a very high architecture efficiency and supports all SBTC modes of 3GPP LTE and DBTC of WiMAX standards respectively.

The chapter is organized as follows: First section presents the proposed design flow using Processor Designer tool to describe application-specific parametrized cores. The second section illustrates the motivations behind the architectural choices and describes the proposed architecture of TDecASIP. Finally, the last two sections of the chapter presents the FPGA and ASIC synthesis results highlighting the attained architecture efficiency of this new flexible Turbo decoder design.

5.1 Proposed design flow for parameterized cores

Although the proposed multi-DecASIP LDPC/Turbo decoder in Chapter 3 presents original scalability and flexibility features, the achieved overall architecture efficiency is still relatively low. The objective of this new study is to focus more on the architecture efficiency and to investigate the maximum achievable architecture efficiency when using ASIP-based LDPC decoder design. To that end, different parallelism choices and corresponding memory organization are explored.

Another related motivation of this work concerns the investigation of the possibility design parameterized cores using the available ASIP design approach. Such possibility can potentially lead to a higher architecture efficiency by simplifying the instruction decoding logic and removing the program memory. Furthermore, it should lead for an increased energy efficiency as there are no program memory accesses in this case. Finally, such an approach still keeping the benefit of the short design cycle enabled by the well established ASIP design tools.

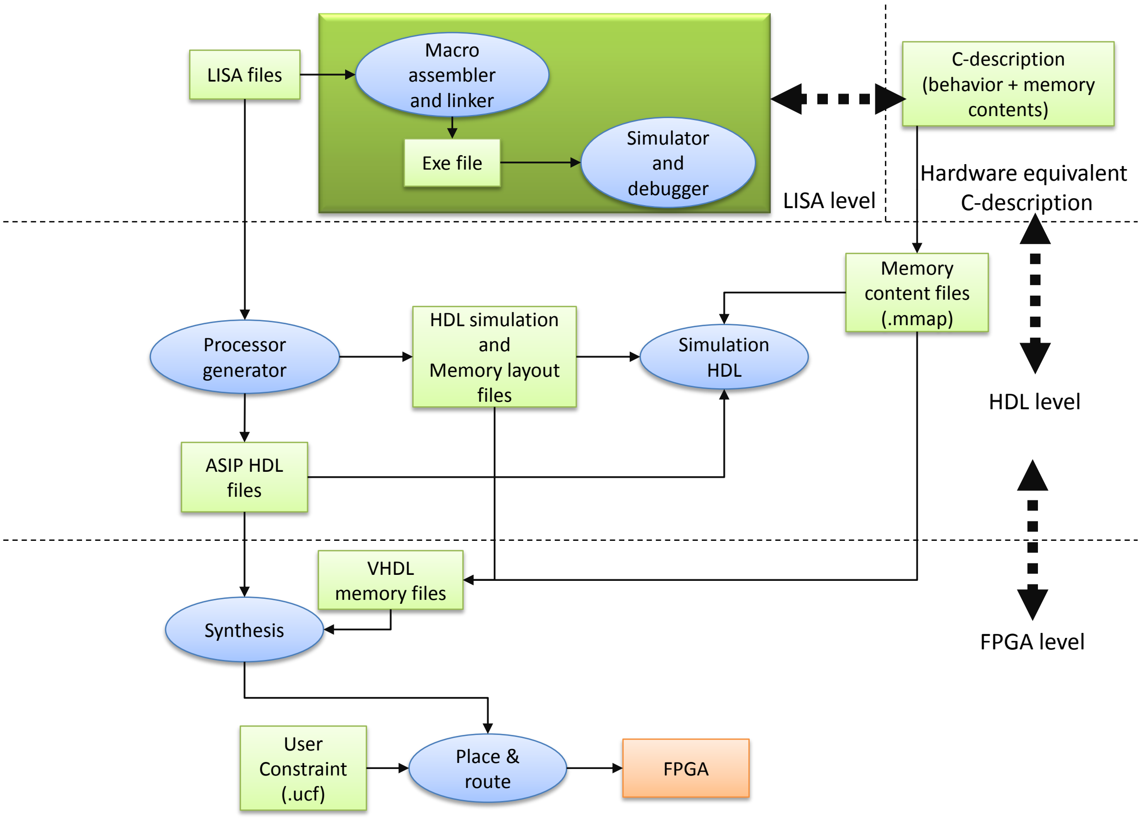


Figure 5.1 — Proposed design flow for parameterized ASIP

To that end, Figure 5.1 illustrates the proposed modified ASIP design flow for parameterized cores design. The instruction program memory is used as a configuration (config) memory, where the configuration parameters are stored. Rather than defining specialized instructions, the corresponding finite state machine (FSM) is directly described in LISA. The current state of the FSM is treated as an instruction by the Decode pipeline stage. This approach can be effective when the application exhibits a reduced number of flexible parameters and the corresponding processing presents a reduced number of control states. The target application in this study (flexible Turbo decoding) is a good example with 7 states and few flexible parameters that do not change during the decoding process (detailed in Section 5.2.2). Compared with the original ASIP design flow (Figure 2.3 in page 40), the proposed design flow presents the following features:

1. The target parameterized design does not have a program memory and hence no instructions are to be designed. The design of the specialized instructions is replaced by a state machine control, which is done as part of the LISA ADL description in the *Fetch* pipeline stage.
2. Since hardware-equivalent C model is already used for studying the quantization effects on the algorithm implemented in hardware, the same model can be used to generate the input memory files for simulation purpose, i.e. *.mmap files. Additionally, the corresponding test vectors can be generated for various check points in the design and can be used to cross verify the design functionality between LISA modeling, HDL simulation, and on-board FPGA implementation.
3. The use of the LISA simulator and debugger is still possible. A dummy assembly file with NOP (idle command) is sufficient to generate the required executable file (*.out) in order to mimic the fetch from instruction memory by the fetch stage of the pipeline. The *Fetch* stage reads the configuration parameters (e.g. decoding mode, iterations, window sizes, etc.) and passes the control to the FSM that issues the commands to the rest of the pipeline stages. Thus, the design of the FSM can be validated with the Processor designer LISA debugger.

5.2 Design choices and TDecASIP decoder architecture

Towards the above mentioned objectives, this section summarizes the main design choices and presents the proposed TDecASIP architecture along with the corresponding memory organization.

5.2.1 Design choices

The main design choices can be summarized as follows:

- In order to achieve the target throughput in the range of hundreds Mbps, a maximum sub-block parallelism degree of 4 is adopted. In fact, such parallelism degree allows for conflict-free memory accesses in both WiMAX and LTE standards.
- As in the DecASIP case, each sub-block is further divided into L windows of length W. This reduces the depth of the storage memory required for storage of previous state metrics (as required by the equation (1.31) in page 19) to W.
- Each TDecASIP uses two recursion units and employs Backward-Forward schedule for window processing. The first recursion unit (processing in the backward direction of the trellis) works on window *j* while the second recursion unit (processing in the forward direction of the trellis) works on window $j - 1$ at the same time instant (as shown in Figure 5.2). This enables to achieve the throughput equivalent to butterfly schedule (as in TurbASIP design) but by using backward-forward schedule which further enables use of hardware interleave address generators for extrinsic memory addressing.
- In backward recursion, at the end of processing of the j^{th} window, the boundary state metrics are stored in an external (*BoundaryState*) memory. These state metrics are later used as initial states for the window $(j - 1)$ in the subsequent iteration. In TDecASIP, the window size is considered to be W=64 symbols (as it was the case in TurbASIP and DecASIP).

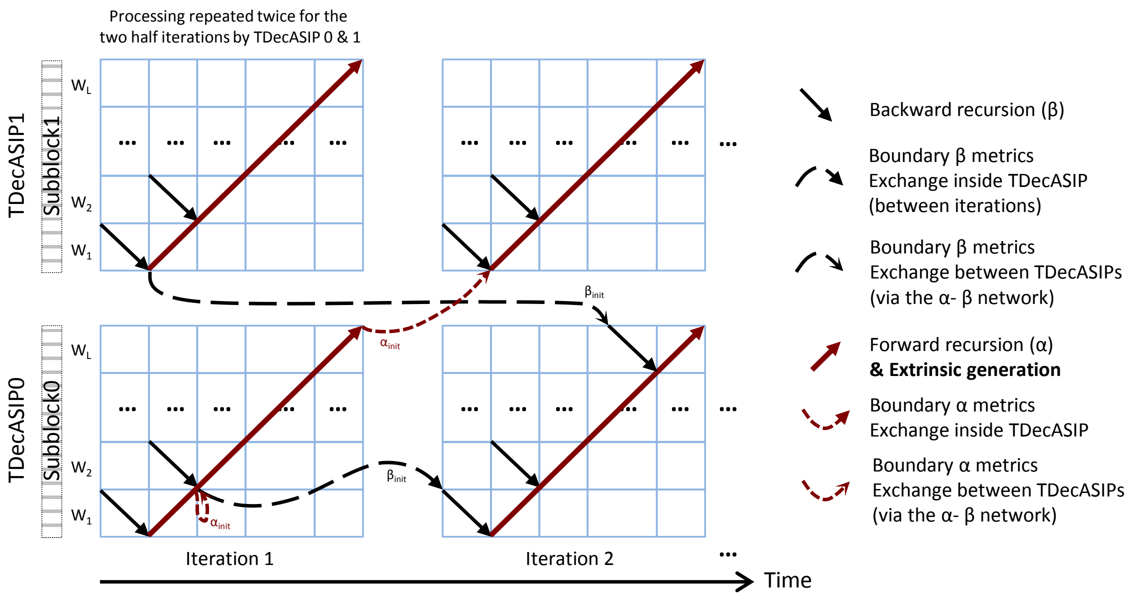

Processing repeated twice for the two half iterations by TDecASIP 0 & 1
TDecASIP1
Subblock1
TDecASIP0
Subblock0
W_L
W_2
W_1
β_{init}
α_{init}
Iteration 1
Iteration 2
Time
Backward recursion (β)
Boundary β metrics Exchange inside TDecASIP (between iterations)
Boundary β metrics Exchange between TDecASIPs (via the α- β network)
Forward recursion (α) & Extrinsic generation
Boundary α metrics Exchange inside TDecASIP
Boundary α metrics Exchange between TDecASIPs (via the α- β network)

Figure 5.2 — TDecASIP: Windowing and backward-forward schedule

- Half iterations are performed in serial order, i.e. all processing cores perform first half iteration by reading the systematic and extrinsic information sequentially from memories, followed by the second half iteration where the systematic and extrinsic memories are read in interleaved order. The generated extrinsic data are written at the same location as it was read from. In both of these half iteration cycles the parity memory is always read sequentially. This type of scheduling has the following advantages:

  1. Only one copy of systematic information bits are needed to be stored. This reduces the number of memory banks required and the configuration network complexity.
  2. Only sequential counter and interleaved address generator are needed for addressing the memories while the shuffled decoding needs in addition a deinterleaved address sequence. Given the adopted low sub-block parallelism degree of 4, this serial decoding reduces the memory access complexity as only low number of multiplexers would be sufficient (crossbar switch). WiMAX interleavers support sub-blocking of 2 and 4 while LTE interleavers support sub-blocking of at least 2 and 4 [123] (with a maximum of 64).
  3. Fewer number of memory banks also results in less address decoding logic and hence reduced total memory area, resulting in area efficient decoding core.

- The DecASIP architecture, described in the previous chapter, was validated and demonstrated with the complete communication system prototyped on FPGA. This approach is good when both validation and real time working prototypes are the design objectives. However, it is a complex task which requires significant design and validation time as many blocks other than the channel decoder itself, have to prototyped. Thus, when quick on-board validation of the design is the objective, a simpler hardware-in-the-loop strategy with only the design under test implemented onto the FPGA is sufficient.

Based on the above design choices, we propose a 2-TDecASIP decoder architecture as shown in Figure 5.3. Each TDecASIP core processes a sub-block of the input frame and interconnected by two ring 80 bit bus to enable state metric exchanges across sub-blocks.As β state metrics are quantized to 10 bits, 80-bit (for 8 states) wide bus is needed to exchange the boundary state metrics between processors. Each core has direct access to configuration, cross metric, boundary state and input parity memories. The input Systematic and Extrinsic memory banks are connected to the cores through a simple read/write exchange network as illustrated in Figure 5.3.

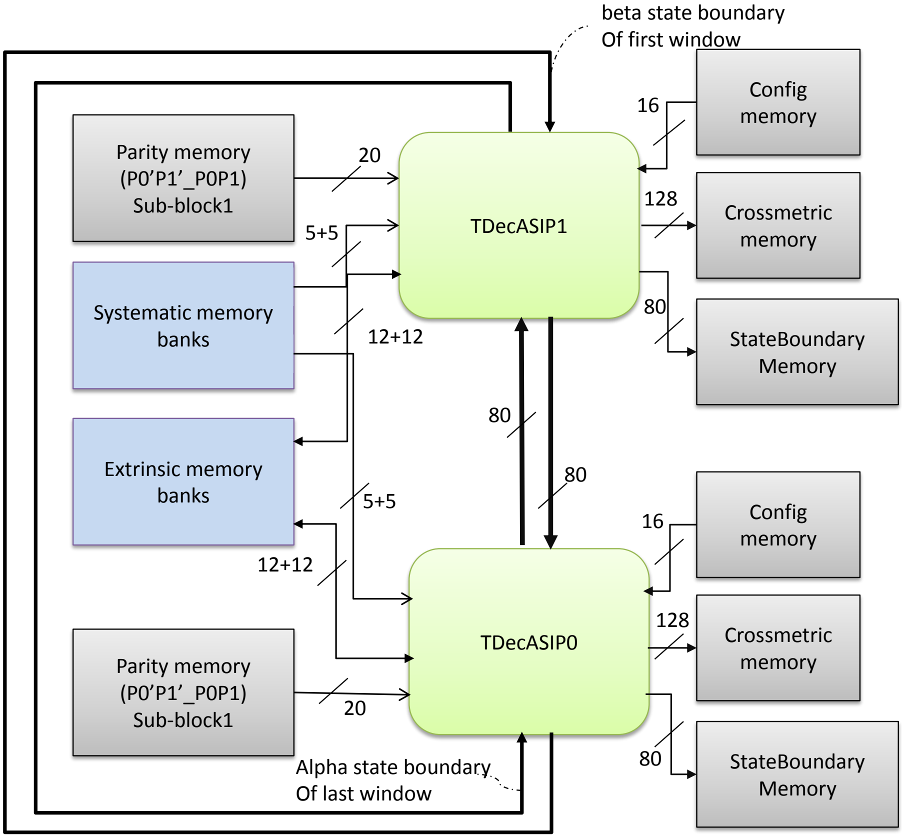


Figure 5.3 — Overview and memory organization of the proposed 2-TDecASIP Turbo decoder architecture

5.2.2 TDecASIP decoder architecture

Figure 5.4 shows the different hardware components of the proposed design. The design consists of 8 pipeline stages, of which the first 3 stages are dedicated for data fetch from the memories and for the control of the pipeline. Since the number of flexible parameters is small, these values are fetched from the configuration memory by the *ConfigFetch* stage. These parameters consist of the following:

- Mode: LTE or DVB/WiMAX.
- The number of iterations to be executed.
- Normal window size (W), size of the last window (W_L) and the number of windows (L).
- Extrinsic address generation initialization values: these values are required to configure the address generation logic in WiMAX/DVB or LTE modes [123].

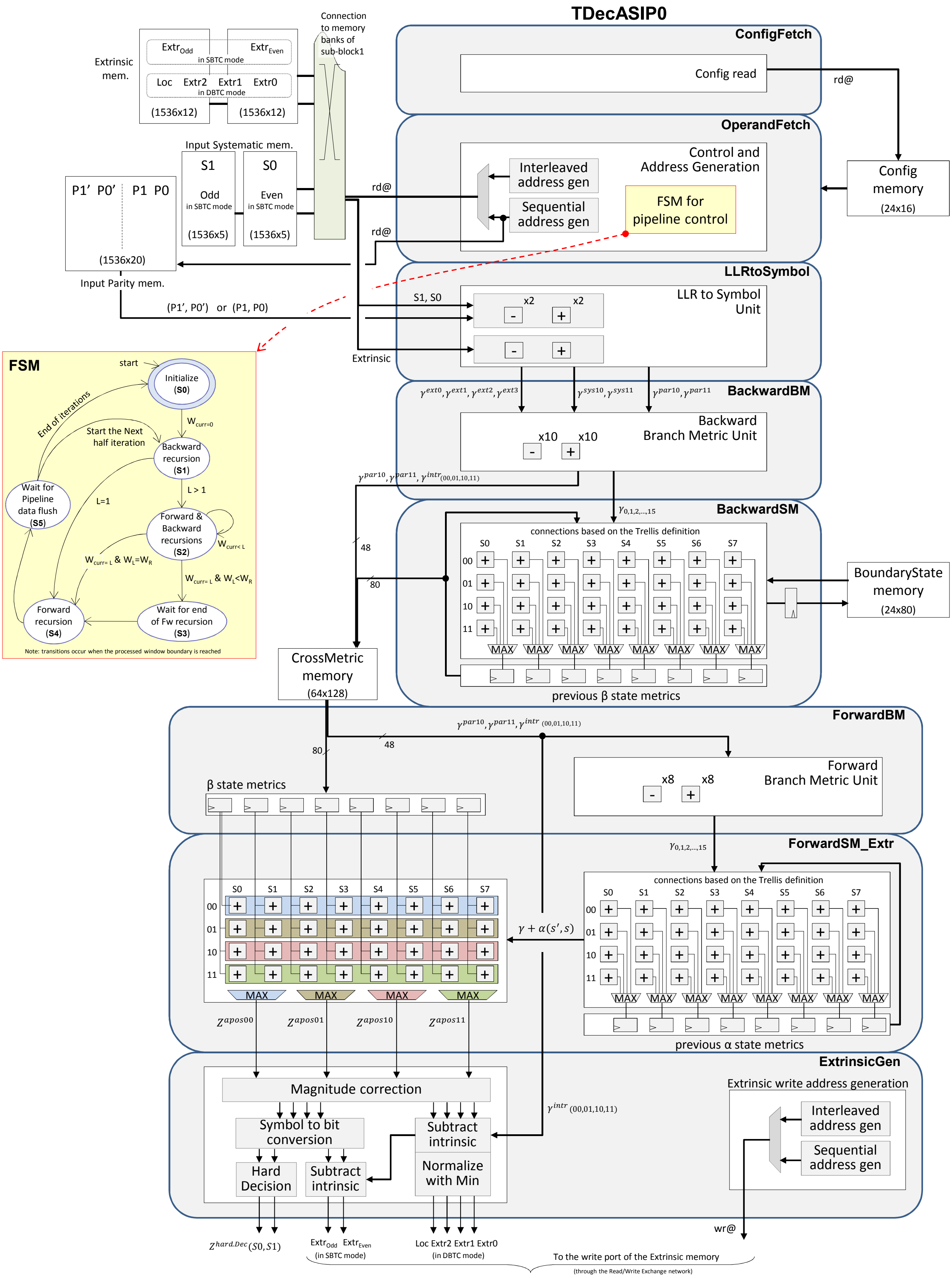


Figure 5.4 — Detailed pipeline architecture and FSM of the proposed TDecASIP parametrized core

5.2.2.1 Pipeline control finite state machine

Using the above parameters, the address generation and control unit of the *OperandFetch* stage implements a finite state machine (FSM) that generates appropriate control signals to activate or deactivate appropriate stages of the pipeline (Figure 5.5). As soon as the start signal is asserted the processor starts with the *Initialize* state, initializing the registers to the default values, reading the configuration parameters mentioned in before.

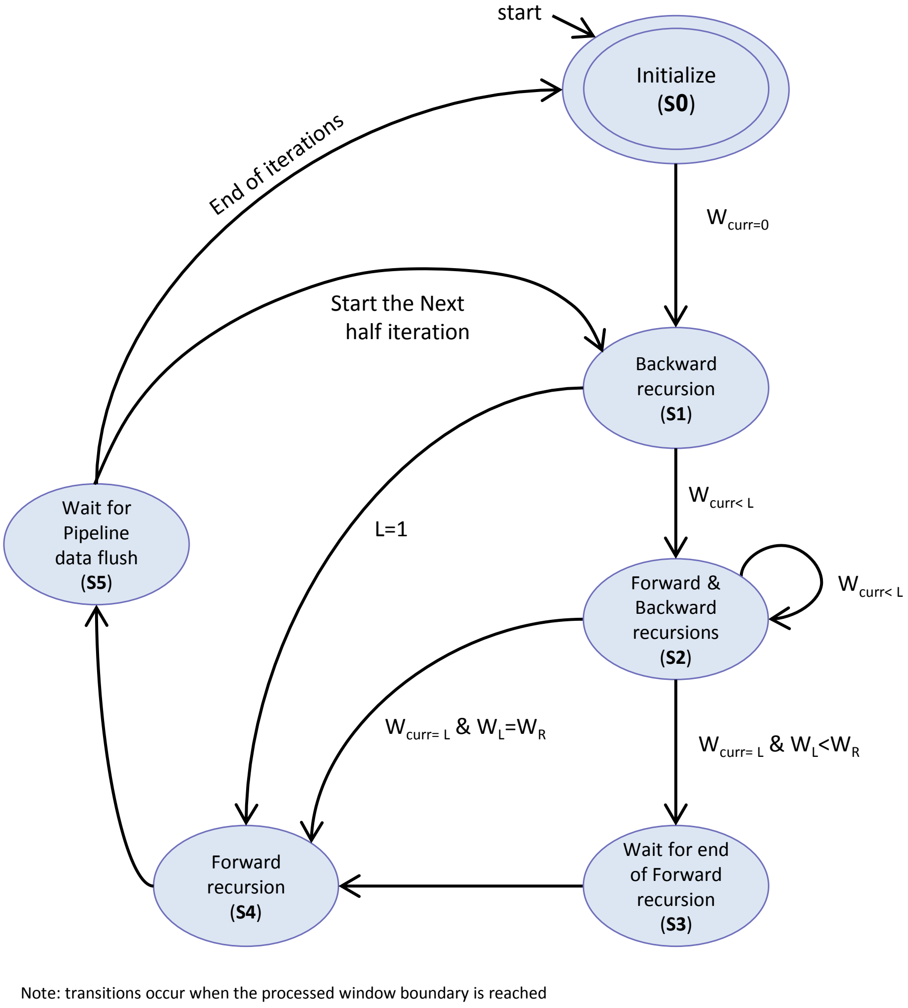


Figure 5.5 — TDecASIP: Finite state machine for decoder execution control

At the end of the initialization, the FSM reaches *S1* state generating appropriate signals for the backward recursion execution. If the processor is executing first half iteration, the generated addresses for systematic and extrinsic memories are sequential otherwise interleaved addresses are generated. The addresses for parity memories are always sequential.

All FSM transitions in Figure 5.5 occur when the window boundary is reached. At the end of the window processing, if the number of windows $L = 1$ then the forward recursion is executed for the window currently processed else the control is passed to $S2$ state. In $S2$ state both forward recursion for W_{curr-1} and backward recursion for W_{curr} window are executed in parallel by two dedicated state metrics processing units as shown in Figure 5.4. Since Cross metric memory is read and written simultaneously by two different execution units, *Crossmetric* memory bank is chosen to be dual port memory.

When the backward recursion unit completes the processing of all L windows, the forward recursion unit would still be executing W_{L-1}. In case $W_L < W_R$, i.e. the Lth window size is less than W_R, a wait state $S3$ is introduced (corresponding to W_R-W_L clock cycles) so that the forward recursion unit can complete the execution of the $(L-1)^{th}$ window before transition to state $S4$. The $S4$ state only generates signals to activate forward recursion of the last window. Once the forward recursion of the last window (W_L) is complete, 5 clock cycles wait state $S5$ is inserted to ensure all the control data are flushed out of the pipeline before starting the next half iteration. During the last half iteration, hard decisions are made on the aposteriori LLRs.

5.2.2.2 Pipeline architecture

The *LLRtoSymbol* pipeline stage converts the fetched systematic and parity bit LLRs to symbol LLRs according to equations (1.27) to (1.30). If the processor is executing the first half iteration, the least significant 10 bits (P1,P0) of the parity data fetched are used, else the most significant 10 bits (P1',P0') are used for processing. With negligible performance loss the channel LLRs can be quantized to 5 bits and normalized extrinsic information from the processor are quantized to 7 bits as it was inferred from the quantization analysis in Sub-section 3.1.2 in page 60. The *BackwardBM* pipeline stage computes the branch metrics $\gamma_i(s', s)$ for the backward recursion using extrinsic, systematic, and parity symbol LLRs as described by the equation (1.33).

Then the pipeline stage *BackwardSM* computes the 8 backward state metrics $\beta_i(s)$ corresponding to the received symbol i as defined in equation (1.31). To that end, the computations related to the 32 trellis transitions (refer Figure 1.7) are done in parallel using 32 adder nodes and 8 maximum operators as illustrated in Figure 5.4. The computed 8 backward state metrics $\beta_i(s)$ are buffered in the CrossMetric memory as they are needed to compute the aposteriori information Z_i^{apos} in the *ForwardSM_Extr* stage. In addition, the CrossMetric memory buffers the $\gamma_i^{intr}(s', s)$ and $\gamma_i^{par}(s', s)$ of the processed window so they can be used directly in the *ForwardBM* pipeline stage for the computation of the branch metrics $\gamma_i(s', s)$ in the forward recursion (avoiding external memory accesses and double-ported input memories). State metrics are quantized to 10 bits while the intrinsic LLRs γ_i^{intr} need an 8 bits quantization.

The *ForwardSM_Extr* pipeline stage includes all the required hardware units (Figure 5.4) to compute the forward state metrics $\alpha_i(s)$ as defined in equation (1.32) and to complete the computation of the 4 aposteriori LLRs Z_i^{apos} of the symbol k after fetching the $\beta_i(s)$ from the CrossMetric memory. Overflows are allowed in the state metric calculations $(\alpha(s), \beta(s))$ and the magnitude correction unit of the *ExtrinsicGen* pipeline stage implements the modulo normalization (as explained in Chapter 2). Finally, the magnitude corrected aposteriori Z_i^{apos} and the intrinsic $\gamma_i^{intr_x}$ LLRs are used to generate the normalized extrinsic (for DBTC or SBTC modes) and the hard decision.

5.2.3 Memory organization

The memory organization of the proposed architecture is illustrated in Figure 5.3. With negligible performance loss, the channel LLRs can be quantized to 5 bits and the normalized extrinsic information to 7 bits. As radix-4 is adopted in SBTC, systematic LLRs are stored in two memory banks, and similarly for extrinsic LLRs. This memory organization and the corresponding efficient address generation are allowed by the QPP (quadratic permutation polynomial) interleaver adopted in LTE standard which maps even addresses to even addresses and odd to odd. The total depth of these memories allow to store up to 6144 LLRs, which corresponds to the maximum specified LTE frame length. As the parity LLRs are always read in sequence, the consecutive parity LLRs information bits are combined and stored in one memory bank as shown in Figure 5.3.

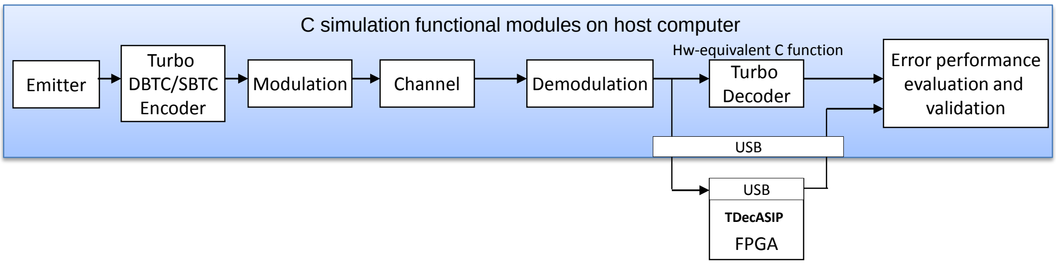


Figure 5.6 **—** TDecASIP: Prototyping environment

5.3 FPGA prototype and synthesis results

As mentioned in the design motivations (Sub-section 5.2.1) , prototyping the complete communication system is a complex task which requires significant design and validation time as many blocks other than the channel decoder itself, have to prototyped and validated towards their reference software models. This approach has been used for the validation of DecASIP.

However, for TDecASIP, we proposed to explore a simpler hardware-in-the-loop strategy with only the design under test implemented onto the FPGA for quick on-board validation (Figure 5.6). The basic design principle of the proposed prototype is to implement only the Turbo decoder (TDecASIP) along with its memories on the FPGA while keeping all the other components of the communication system running in software at the host computer. The USB interface connecting the FPGA board to the host computer is used to establish the required data and control transfers. The TDecASIP can be configured to run for one or several iterations and the memory values are read back into the host computer to validate the decoding process and to estimate the error rate performance.

This approach of FPGA prototyping enables fast validation as it requires the implementation of very few additional hardware components on the FPGA apart from the channel decoder (i.e. TDecASIP, the design unit under validation) itself. All remaining design units like data source (emitter), Turbo encoder, modulation, channel model and demodulation units are executed as C functions in the host computer connected to the FPGA platform.

The output of the demodulation function is decoded in parallel on the FPGA and on the host computer (by the hardware-equivalent C function of the Turbo decoder). Once the required number of decoding iterations is reached, the outputs of the hardware-equivalent C function are compared with that obtained from the FPGA to evaluate the error rate performance and to prove the correctness of the implemented design.

In this context, a single TDecASIP along with the associated memories are synthesized and prototyped into the FPGA hardware (Xilinx Virtex-5 (xc5vlx330-1ff1760)). The results of logic synthesis are as shown in Table 5.1.

Design unit	Slices regs.	LUTs
ConfigFetch	8	11
OperandFetch	250	381
LLRtoSymbol	0	133
BackwardBM	0	176
BackwardSM	92	1102
ForwardBM	0	300
ForwardSM_Extr	160	1875
ExtrinsicGen	27	569
RegisterFile	806	468
MemoryInterface	297	548
Total logic area	1697	5582
Total BRAMs	20	
Operating frequency	44 MHz	

Table 5.1 — TDecASIP: FPGA (Xilinx Virtex-5 (xc5vlx330-1ff1760)) resource utilization for 1 processor

5.4 ASIC synthesis results

The proposed parameterized core was modeled with Synopsys Processor Designer tool and the corresponding VHDL description was generated and synthesized targeting 65nm general purpose CMOS technology (worst case 0.9v and 125C). Table 5.2 and 5.3 and summarizes the memory partitions and the post-synthesis logic and memory area results obtained for a single core respectively. All the memories used are single port (sp) memories except for the CrossMetric and extrinsic memories which are double port (dp) memories. The total logic area, including the interleaver, is 0.065 mm^2 while the memory area for one processor is 0.15 mm^2. The total area (post-synthesis) for the two core Turbo decoder design presented in this paper is 0.437 mm^2 for a clock frequency of 510 MHz. If the frame length is N bits and the window size is W symbols, then the throughput of the proposed Turbo decoder is given by:

Memory	width (bits)	depth	#	type[a]
Systematic	5	1536	2	sp
Parity	20	1536	1	sp
Extrinsic	12	1536	2	dp
Cross Metric	128	64	1	dp
BoundaryState	80	48	1	sp
Configuration	16	24	1	sp

[a] Dual port (dp) / Single port (sp)

Table 5.2 — TDecASIP: Memory partitions

$$Throughput = \frac{Num_{procs} \times N \times f_{clk}}{((\frac{\lceil N_{sym}/W \rceil}{Num_{procs}} + 1) \times W + N_{pip}) \times (2 \times N_{iter})} \tag{5.1}$$

For the presented architecture: $Num_{procs} = 2$ processors, the maximum clock frequency is $f_{clk} = 510$MHz, considering the largest LTE frame size $N_{sym} = 3072$ symbols or $N = 6144$ bits and $N_{iter} = 6.5$ iterations, the throughput obtained is $Throughput = 150$ Mbps.

The proposed 2 processor Turbo decoder achieves an architecture efficiency of 4.37 bit/clock /iteration /mm^2. Furthermore, the proposed architecture is scalable and can be extended to 4 processing cores, since both LTE and WiMAX interleavers support sub-blocking levels of 4. In this case, the memory area of one processing core decoder 0.097mm^2 which results in total area occupancy of 0.65mm^2. The architecture efficiency in this case is $Arch_{Efficiency} = 5.88$

Design unit	Area (um^2)
ConfigFetch	191
OperandFetch	6586
LLRtoSymbol	957
BackwardBM	1905
BackwardSM	10038
ForwardBM	2480
ForwardSM_Extr	17847
ExtrinsicGen	5006
RegisterFile	13695
MemoryInterface	6683
Total logic area	65390
Total mem area	153478
Total area	218868

Table 5.3 **—** TDecASIP: Post synthesis area utilization per processor for 2 processor architecture with general purpose CMOS 65nm, (worst case 0.9v, 125C)

bit/clock/iteration/mm^2. This further illustrates the area efficiency of sub-block parallelism, wherein the throughput is doubled while the occupied area is increased only by 1.47 times (rather than doubled). This is due to the fact that Systematic, Parity, Extrinsic, and BoundaryState memory sizes remain unchanged.
The achieved results of the proposed design are summarized and compared along with some recent related works in Table 5.4. Considering this definition, the proposed 2 processor Turbo

	This work		[59]	[124]	[62]
Standard supported	LTE, WiMAX		LTE, WiMAX	LTE	LTE
LTE modes supported #	188		188	188	188
WiMAX modes supported #	17		17	-	-
Technology (nm)	65		130	90	65
Core area (mm^2)	0.438	0.65	10.7[a]	2.1	7.7[a]
$Area_{Norm}$ @65nm (mm^2)	0.438	0.65	1.335	1.1	3.85
Throughput (Mbps)	150 @6.5iter	300 @6.5iter	187 @8iter	284 @5iter	2150 @6iter
Parallel MAPs #	**2**	**4**	**8**	**16**	**32**
f_{clk} (MHz)	510		250	200	450
AE (bits/cycle/iter/mm^2)	4.37	5.88	4.48	6.49	7.45

[a] Post place&route

Table 5.4 **—** Results and comparison with with few recent related works

decoder achieves an architecture efficiency of 4.37 bits/cycle/iteration/mm^2. Furthermore, the proposed architecture is scalable and can be extended to 4 processing cores, since both LTE and WiMAX interleavers support sub-blocking level of 4 with conflict-free memory accesses. In this case, the memory area of one processing core decoder becomes 0.097mm^2 which results in a total area occupancy of 0.65mm^2. The architecture efficiency in this case is 5.88 bits/cycle/iteration/mm^2. This further illustrates the area efficiency of the sub-block parallelism, where the throughput is doubled while the occupied area is increased only by 1.47 times (rather than doubled). This is due to the fact that Systematic, Parity, Extrinsic, and BoundaryState memory requirements remain unchanged. The achieved results of the proposed design are summarized and compared along with few recent related works in Table 5.4. The cited three implementations [59] [124] [62] use a conventional parametrized design approach with almost similar internal computation, interleaving, and storage optimization techniques. However, each of them has selected a different sub-blocking parallelism level (8, 16, and 32). The increased architecture efficiency with the sub-blocking parallelism degree is coherent with the above discussed results of the proposed 2- and 4-TDecASIP architectures. The 4-TDecASIP architecture achieves even a slightly better architecture efficiency than the one presented in [59] which supports both Turbo modes (DBTC and SBTC) and uses 8 parallel MAP decoders. The LTE-dedicated implementations presented in [124] and [62] exploit the available higher sub-blocking parallelism degrees in this standard (parallel interleaving with conflict-free memory accesses). Results comparison illustrates how the proposed architecture achieves a high architecture efficiency while using such an ASIP-based parameterized core approach by selecting the appropriate parallelism and optimization techniques.

5.5 Summary

In this chapter we have presented a novel parameterized architecture for multi-standard Turbo decoding, namely TDecASIP. The architecture was designed to maximize the architecture efficiency, which was achieved through the use of high sub-block parallelism degree compared to DecASIP (presented in Chapter 3). Thus, a 2 TDecASIP system decoder reaches a throughput of 150 Mbps with an AE of 4.37 bit/cycle/iteration/mm^2 outperforming related state of the art implementations. Furthermore, the proposed architecture is scalable to the degree of sub-block parallelism allowed by the standards (4 for WiMAX and 32 after Radix-4 optimization for LTE). The attainable architecture efficiency increases with sub-block parallelism degree, thus a 4-TDecASIP Turbo decoder achieves and AE of 5.88 bit/cycle/iteration/mm^2 with a throughput of 300 Mbps.

Another main contribution of this work concerns the use of available ASIP design flow to design application-specific parametrized cores. The main idea was to evaluate the benefits from removing the program memory and the related instruction decoder. Besides its direct impact on the architecture efficiency, such an architecture model should improve energy efficiency while keeping the benefit of ASIP design tools for high level and quick design and debugging flow. Thus, the proposed TDecASIP does not perform instruction fetch from an external program memory which has been replaced by a simple and structured finite state machine implemented in the *OperandFetch* pipeline stage. Finally, a new FPGA-based prototyping environment has been proposed with a hardware-in-the-loop approach for quick and accurate on-board validation.

CHAPTER

6 LDecASIP: LDPC Decoder

In THIS chapter we present an optimized flexible standalone LDPC decoder, namely LDecASIP. The objective behind the conducted new design is twofold: (1) investigate the maximum attainable architecture efficiency for ASIP-based LDPC decoding and (2) explore the possibility of realizing a flexible decoder that allows to implement and validate new/incremental algorithm changes with fast turnaround time in design. The idea of this last objective is to enhance the ASIP-based LDPC decoder with a design-time feature enabling incremental changes for future support of other QC-LDPC codes (e.g. DVB-S2 with high expansion factor $Z = 360$).

Towards fulfilling these objectives, LDecASIP architecture is proposed and designed. The proposed architecture exhibits a very high architecture efficiency, supports all QC-LDPC codes and related parameters of WiFi and WiMAX standards (with expansion factors ranging from $Z = 24$ to $Z = 96$), and enables the support other QC-LDPC codes with structured incremental hardware changes at design time.

The chapter is organized as follows. The first section describes the design motivations along with the proposed LDPC decoder architecture. The second section presents the added design-time flexibility feature and illustrates the proposed way of upgrading the design to support other QC-LDPC codes through the example of DVB-S2 LDPC decoding. Finally, the third section presents the FPGA and ASIC synthesis results with architecture efficiency evaluation and related discussions.

6.1 Design motivations and LDecASIP decoder architecture

Although the proposed multi-DecASIP LDPC/Turbo decoder in Chapter 3 presents original scalability and flexibility features, the achieved overall architecture efficiency is still relatively low. The objective of this new study is to focus more on the architecture efficiency and to investigate the maximum achievable architecture efficiency when using ASIP-based LDPC decoder design. To that end, different parallelism degree, memory organization, and communication interconnect are proposed. Another motivation of this work concerns the investigation of the possibility to support other QC-LDPC codes than the ones specified in WiFi and WiMAX. In this context, we considered the QC-LDPC codes specified in the DVB-S2 standard which present very high expansion factor Z=360 and frame lengths (64800 and 16400 bits).

Regarding the parallelism degree, in fact the VN update and CN update steps of the NMS decoding algorithm can be parallelized and pipelined as long as there are no conflicts in simultaneous updates of the variable nodes. Supporting the maximum parallelism degree of 360 allowed by DVB-S2 standard leads to inefficient use of the hardware resources when lower parallelism degrees are permitted (e.g. Z=24 for WiMAX LDPC standards). Furthermore, considering the throughput requirement of these standards which is in the range of hundreds of Mbps and the number of iterations required for decoding which is in the range of 10 to 25 (depending on the standards and the code rate for WiMAX and WiFi and DVB-S2 [86,98,108]), a maximum parallelism degree P_{Deg}=48 is chosen. This means that a maximum of 48 check nodes are processed in parallel by the proposed new ASIP for LDPC decoding, namely LDecASIP (Figure 6.1).

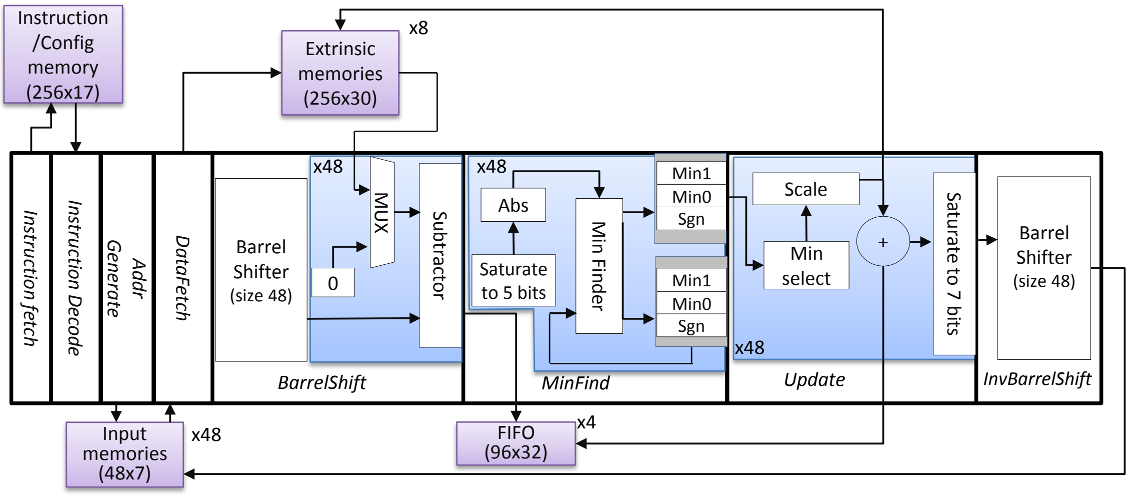
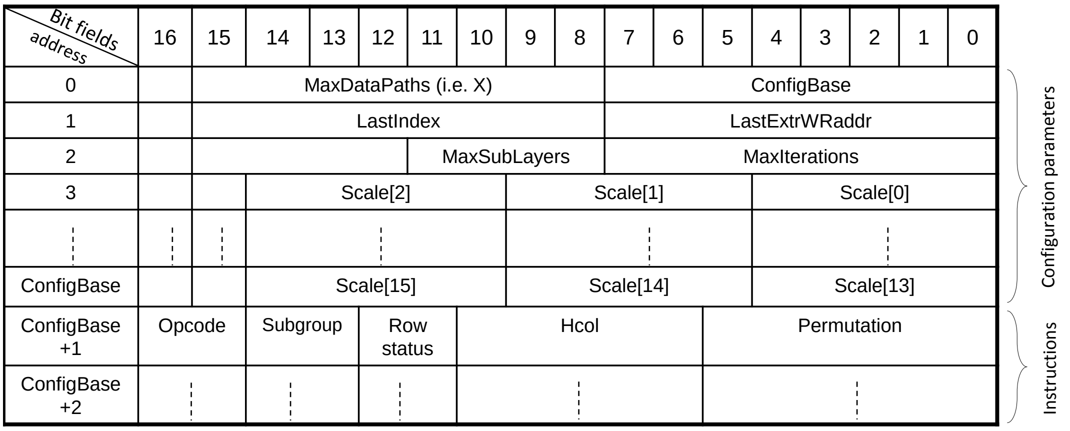


Figure 6.1 — LDecASIP pipeline architecture

Two flexible barrel shifters of size 48 are used to implement the cyclic shifts of any permutation (related to the standards): one for reading and one for writing channel LLRs in the input memory banks. With this parallelism degree, decoding LDPC frames with $Z > 48$ are supported by dividing the group of check nodes into two sub-groups (refer Figure 1.20 in page 28). Each sub-group is scheduled for processing in two phases by applying the layered decoding schedule (as explained in section 1.5.3 of page 32). In fact, all expansion factors for $Z > 48$ are multiple of ”2” except for $Z = 81$ in WiFi standard. For this particular case, the group is split into two sub-groups containing 40 and 41 check nodes. In this context, the barrel shifter flexibility covers all specified expansion factors $Z < 48$: 24, 26,28,30,32...48, besides the support of Z=27 and 41. For DVB-S2 with Z=360, each group is split into 9 sub-groups containing 40 check nodes each.

The proposed LDecASIP architecture has 8 pipeline stages. The configuration parameters regarding check matrix definition and the decoding instructions are stored in a single 17 bit wide instruction/config memory (Figure 6.2). The first part of this memory until *ConfigBase* stores the configuration parameters specifying the key characteristics of the check matrix: the number of parallelism degree (*MaxDataPaths*), the depth of the instruction memory (*LastIndex*), number of sub-groups (*MaxSubgroups*), number of iterations (*MaxIterations*) and the lookup table containing the scaled 4-bit CN to VN message $L(m,n)$ (*scale[0..15]*). As in DecASIP the quantization of 7 bits for channel LLR $L(n)$ and 5 bits for $L(m,n)$ are used with acceptable performance losses (as shown in figure 3.3 of page 61).



<table>
<tr><th>Bit fields
address</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td></td><td colspan="8">MaxDataPaths (i.e. X)</td><td colspan="8">ConfigBase</td></tr>
<tr><td>1</td><td></td><td colspan="8">LastIndex</td><td colspan="8">LastExtrWRaddr</td></tr>
<tr><td>2</td><td></td><td colspan="4"></td><td colspan="4">MaxSubLayers</td><td colspan="8">MaxIterations</td></tr>
<tr><td>3</td><td></td><td></td><td colspan="5">Scale[2]</td><td colspan="5">Scale[1]</td><td colspan="5">Scale[0]</td></tr>
<tr><td>⋮</td><td>⋮</td><td>⋮</td><td colspan="5">⋮</td><td colspan="5">⋮</td><td colspan="5">⋮</td></tr>
<tr><td>ConfigBase</td><td></td><td></td><td colspan="5">Scale[15]</td><td colspan="5">Scale[14]</td><td colspan="5">Scale[13]</td></tr>
<tr><td>ConfigBase
+1</td><td colspan="2">Opcode</td><td colspan="2">Subgroup</td><td colspan="2">Row
status</td><td colspan="5">Hcol</td><td colspan="6">Permutation</td></tr>
<tr><td>ConfigBase
+2</td><td colspan="2">⋮</td><td colspan="2">⋮</td><td colspan="2">⋮</td><td colspan="5">⋮</td><td colspan="6">⋮</td></tr>
</table>

Configuration parameters
Instructions

Figure 6.2 — LDecASIP: Instruction/config memory

The rest of the memory carries the instructions that control the execution as shown in Figure 6.2. Each instruction is composed of 5 fields. The first two fields, *Hcol* and *Permutation* specify the column number of the LDPC check matrix H_{base} and the associated cyclic permutation value respectively. The permutation value specified here is given after taking into account the available parallelism degree. The third field, *RowStatus* indicates if the *Hcol* corresponds to the beginning, middle or end of a row. The fourth field *Subgroup* is equal to 1 or 0 depending upon if the *Permutation* value fetched is above or below P_{Deg} respectively. The *Hcol* and *Subgroup* are used to determine the read addresses for the input channel memories. These read addresses are calculated as:

$$(fetch\ address)_{bank_{num}} = Hcol * 2 + (Subgroup + \lfloor\frac{(bank_{num} + Permutation)}{Z}\rfloor)\%2 \tag{6.1}$$

where $bank_{num} = 0...P_{Deg} - 1$ is the number of the bank that is being accessed. The fifth field of the instruction format is the *Opcode*. This field can have one of the four values namely: *CNupdt*, *VNupdt*, *CombinedVNnCNupt* or *NOP* corresponding to the execution of CN update, VN update, combined VN and CN updates or idle cycle respectively.

CNupdt instruction initiates the variable to check node messages calculated according to equation (1.59) by using the output of the barrel shifter and the check node to variable node message $L^{i-1}(m,n)$ of the previous iteration $(i-1)$ fetched from the extrinsic memory banks. The 8 bit variable to check node message, namely $L^{i}(n,m)$ are stored in a FIFO memory along with *Permutation*, *Hcol* and *Subgroup* values to be used to generate addresses and barrel shift values during update step. The *MinFind* pipeline stage (Figure 6.1) serially calculates the 2 least absolute minimums $min(0,1)_m$ and sgn_m of the variable to check node messages corresponding to a check node of the sub-group under processing.

Once the variable node group corresponding to the last column of the sub-group/group connected to check node group is read, the *Update* and *InvBarrelShift* stages of the pipeline is activated. This done by *CNupdt* or *CombinedVNnCNupt* instructions depending on if one or more than one sub-group is present per group. $L^i(n,m)$ is read from the FIFO and aposteriori LLRs $L^i(n)$ are calculated according to equation (1.64). Furthermore, the extrinsic LLRs $L^i(m,n)$ are also calculated as given by the equation (1.63). The *InvBarrelShift* pipeline stage (in Figure 6.1) shifts the newly calculated aposteriori LLRs to the original order and writes them back to the channel memories. The check node to variable node update messages $L^i(m,n)$ are stored in the extrinsic memory banks. When more than one sub-groups are present, the *CombinedVNnCNupt* instruction is used to execute the VN update step for the current sub-group while the *Update* and *Inverse Barrel Shifter* pipeline stages execute the VN update for the previous sub-group. This implies the use of dual-port memories for input and extrinsic, however it allows a perfect usage of all pipeline stages.

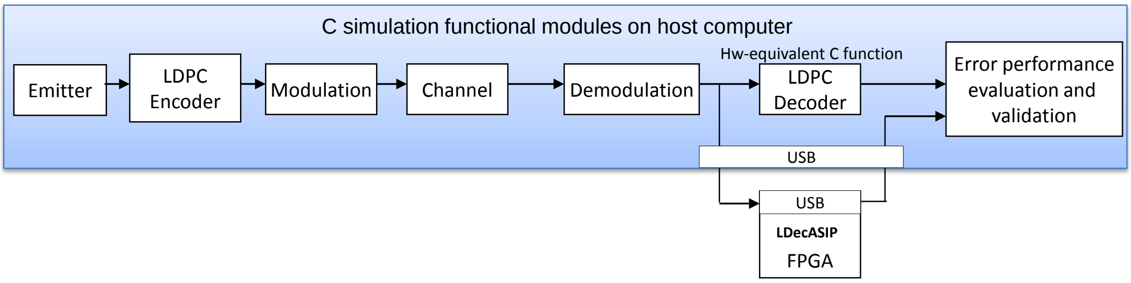


Figure 6.3 — LDecASIP: FPGA prototyping environment

6.2 Prototype and incremental feature addition

The above architecture is prototyped on an FPGA platform and controlled via graphical user interface on a host computer. As it was with the validation of TDecASIP, hardware-in-the-loop strategy is used to validate the design under test as in 5.3, with LDPC encoder and hardware equivalent decoder as shown in Figure 6.3.

Furthermore, as the barrel shifter description is also included in the LISA modelling, the generated VHDL needs only addition of memory modules and input interface to realize a complete decoder system. The input configuration parameters and the channel memories are written through the USB interface from the host computer running the C based simulation program and can be modified at run time. The LDPC hardware chosen can be modified to include 3-λ-min algorithm [99], which is shown to be more performant when the LDPC code rates are less than 1/2 as in the case of DVB-S2 standard. This is done by modifying the *MinFind* pipeline stage to find 3 least minimum absolute values of the variable to check node messages connected to a check node as shown in Figure 6.4. Similarly, the *Update* pipeline stage is also modified accordingly. Furthermore, DVB-S2 specifies frame sizes of length 64800 bits and 16400 bits. These huge codeword lengths correspond to sub-matrix size of Z=360, which can be supported by dividing into 9 sub-groups with each sub-group containing 40 check nodes. It also implies that the input memory banks and correspondingly the extrinsic memory banks need to be extended. The input memory depth for each bank is given by (Max. frame length)/P_{Deg} and the maximum number of banks remains 48. Similarly, given the width of the extrinsic memory W and the number of banks P_{ext} (in the proposed LDecASIP architecture it is chosen to be P_{ext}=8), the depth of an

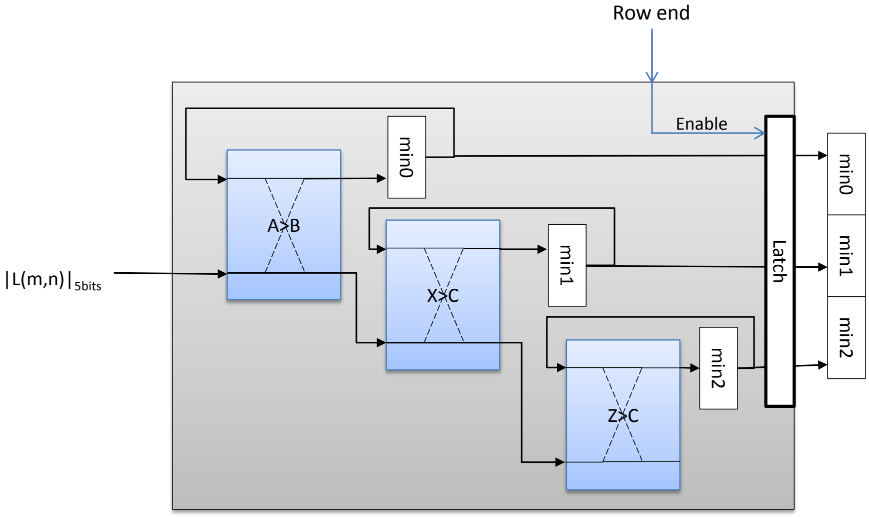


Figure 6.4 — LDecASIP: Min calculator unit for DVB-S2

extrinsic memory bank is given by:

$$Extr.\ mem.\ depth = \frac{\sum_{all\ CNs} check\ node\ degree}{W \times P_{ext}} \tag{6.2}$$

Resulting memory cuts are summarized in Table 6.1 for WiFi and WiMAX standards along with DVB-S2 support (shown in brackets). Furthermore, the DVB-S2 standard specifies check matrices that have few double diagonal permutation matrices (superposed sub-matrices) instead of the single diagonal permutation matrix as in WiFi and WiMAX standards. This implies that the current update mechanism would result in overwriting of the $L^i(n)$ (equation (1.64)) by a second update in the same sub-iteration. This problem can be resolved by differential update mechanism as proposed in [99], where the difference of the extrinsic messages from the check nodes involved in the update of the variable node is used as the final update value.

Memory	width (bits)	depth	#	type[b]
Channel	7	48 (1620[a])	48 (40[a])	dp
Extrinsic	5x6	256 (4860[a])	8	dp
FIFO	128	32	4	dp
Instruction	17	256	1	sp

[a] for DVB-S2

[b] Dual port(dp) /Single port (sp)

Table 6.1 — LDecASIP: Memory bank partition for WiFi and WiMAX standard

6.3 FPGA and ASIC synthesis results

LDecASIP was modeled with Processor Designer tool through which an RTL VHDL description has been generated. Table 6.2 summarizes the synthesis results for both ASIC and FPGA implementation on Xilinx Virtex5 (xc5vlx330-1ff1760) device. The design synthesized currently features the support of WiMAX and WiFi standards. Targeting general purpose CMOS

Hierarchical Unit	CMOS 65nm (um^2)	FPGA (Xilinx Virtex-5(xc5vlx330-1ff1760))		
		Slice regs.	LUTs	BRAMs
MemoryInterface	29836	1245	3891	0
RegisterFile	27742	1890	1890	0
InstrFetch	552	11	47	0
InstrDecode	591	20	61	0
AddrGenerate	5964	22	294	0
DataFetch	2433	18	373	0
BarrelShift	19530	5	2508	0
MinFind	23317	685	2087	0
Update	29105	95	2389	0
InvBarrelShift	15094	0	2370	0
Total logic area	154328	3991	15807	0
Total memory area	427886	0	0	60
Total Area	582215	3991	15807	60

Table 6.2 — LDecASIP: ASIC and FPGA synthesis results

Z	1/2 @20 iter		2/3(A) @17 iter		2/3(B) @17 iter		3/4(A) @15 iter		3/4(B) @15 iter		5/6 @10 iter	
	L	thru.	L	thru.	L	thru.	L	thru.	L	thru.	L	thru.
WiMAX Mode												
24 - 48	87	82 - 165	89	126 - 253	91	124 - 248	98	146 - 293	101	142 - 285	114	210 - 421
52 - 96	163	95 - 176	169	144- 267	173	141 - 261	183	170 - 314	189	165 - 304	202	257- 475
WiFi Mode												
27	110	73			100	127			110	147	104	259
54	192	84			196	129			193	167	185	291
81	200	121			190	200			185	262	172	470

Table 6.3 — LDecASIP: Throughput (in Mbps) and latency L (in clock cycles per iteration) for WiMAX and WiFi mode @500MHz

65nm technology (worst case 0.9V, 125C), the logic area occupies around 0.15 mm^2 while the memory occupies 0.43 mm^2 with maximum operating clock frequency being 500 MHz. The total area of the design post-synthesis is given 0.58 mm^2. Thus resulting best case AE is 13.6 bits/cycle/iter/mm^2.

The achieved results of the proposed design are summarized and compared along with some recent related works in Table 6.5. As it can be seen from the table, the AE of LDecASIP is comparable to the other cited architectures. However, there still room for further optimizations in order to reach the maximum achievable AE of 36.3 obtained in [127]. In fact, such high AE can be explained by the lack of run-time flexibility to support different frame sizes [127]. The architectures presented in [125] and [126] support only WiMAX LDPC codes which have better organization for parallelism than those specified in WiFi standard.

Similarly, synthesis on Virtex-5 (xc5vlx330-1ff1760) FPGA resulted in a design utilization of 3991 slice registers, 15807 LUTs and 60 BRAMs and occupying about 10% of the FPGA

Code rates	1/4	1/3	2/5	1/2	3/5	2/3	3/4	4/5	5/6	8/9	9/10
Latency	2430	3240	3888	4050	5832	4320	4860	5184	5346	4500	4536
Throughput	133	133	133	160	133	200	200	200	202	256	257

Table 6.4 — LDecASIP: Expected DVB-S2 latency L (in clock cycles per iteration) and throughputs (in Mbps) at 25 iterations @500MHz

	LDecASIP	[125]	[126]	[127]
Standard supported	WiMAX/WiFi	WiMAX	WiMAX	WiMAX/Wifi
Tech (nm)	65	180	65	130
Core area(mm^2)	0.58	3.39[a]	3.36[a]	2.744
$Area_{Norm}$ @65nm(mm2)	0.58	0.44[b]	1.68[b]	0.3430
Throughput (Mbps)	73- 475	68	1056	141-249
f_{clk} (MHz)	500	100	110	300
AE (bit/cycle/iter/mm^2)	5-13.6 (LDPC)	15	14.29	20-36.3

[a] (post-layout area)
[b] (scaled down by factor 2 to get an estimated post synthesis area)

Table 6.5 **—** LDecASIP: Comparison with the state of the art

logic resources with maximum clock frequency of 90MHz. The core latency in terms of numbers of clock cycles per iteration required during the decoding process is as shown in Table 6.3. We can notice that the decoding latency is approximately twice for frames with $Z > 48$ than when compared to $Z < 48$ due to the sub-group decoding strategy. Yet the maximum throughput that can be achieved is more than 470Mbps for both WiFi and WiMAX standards.

Similarly, expected throughput for DVB-S2 standard with 25 iterations is given in Table 6.4 with a maximum reaching 257 Mbps. As only slight incremental changes are expected during design time in the logic structure of the hardware, the memories are the major contributor for area increase.

6.4 Summary

In this chapter we have presented a flexible and optimized ASIP-based architecture for multi-standard LDPC decoding, namely LDecASIP. This new architecture was designed to maximize the architecture efficiency, which was achieved by increasing the parallelism degree and the memory bandwidth with flexible barrel shifters. LDecASIP exploits a maximum parallelism degree of 48, i.e. 48 check nodes are processed in parallel, achieving a maximum throughput of 475 Mbps and an architecture efficiency of 13.6 bit/cycle/iteration/mm^2. It supports all LDPC codes and related parameters specified in WiFi and WiMAX standards.

Another main contribution of this work concerns the design time flexibility feature of LDecASIP which allows to implement and validate new/incremental algorithm changes with fast turnaround time in design. The proposed architecture for LDecASIP LDPC decoder enables through incremental changes the future support of other QC-LDPC codes, such as those specified in DVB-S2 with high expansion factor $Z = 360$. This illustrates how the ASIP-based design approach can be used as a way to achieve both design-time and run-time flexibility features. Finally, the proposed architecture has been fully validated through an FPGA prototype based on a hardware-in-the-loop approach.

Conclusions and Perspectives

This thesis work has investigated multi-ASIP architecture model towards the target of unifying flexibility-oriented and optimization-oriented approaches in the design of flexible channel decoders. By considering mainly the challenging Turbo and LDPC decoding applications, multi-ASIP channel decoder architectures are proposed targeting high flexibility combined with high architecture efficiency in terms of bits/cycle/iteration/mm^2. Different architecture alternatives and design approaches are explored.

In this context, three original contributions have been proposed. The first one concerns the design of a scalable and flexible high throughput multi-ASIP LDPC/Turbo decoder. Several design objectives have been attained in this work in terms of scalability, resource sharing, and configuration speed between the different supported decoding modes. The proposed DecASIP supports the decoding of LDPC and Turbo codes specified in WiFi, WiMAX, and LTE standards. An 8-DecASIP system decoder achieves an architecture efficiency AE of 0.7 in Turbo mode and a maximum AE of 3.12 in LDPC mode. Achieved throughputs are 156 Mbps and 437 Mbps in Turbo and LDPC modes respectively. The second contribution concerns the design of a parameterized ASIP for Turbo decoding (TDecASIP). Here the objective was to investigate the maximum attainable architecture efficiency for ASIP-based Turbo decoding when maximising the usage of sub-block parallelism. The proposed 4-TDecASIP system decoder achieves an architecture efficiency AE of 5.3 and a throughput of 300 Mbps in both SBTC and DBTC modes. Furthermore, with this architecture we demonstrated the possibility to design application-specific parametrized cores using the available ASIP design flow. The third contribution corresponds to the design of an optimized ASIP for LDPC decoding (LDecASIP). As for LDecASIP, the objective was to investigate the maximum attainable architecture efficiency for ASIP-based LDPC decoding by increasing the parallelism degree and the memory bandwidth with flexible barrel shifters. The proposed LDecASIP decoder achieves a maximum architecture efficiency AE of 13.6 and a maximum throughput of 475 Mbps.

A fourth main contribution of this thesis work concerns the proposed flexible and fully functional multi-ASIP channel decoder. A complete communication system platform has been emulated along with Turbo and LDPC encoder, channel, proposed 4-DecASIP channel decoder and error counter. The decoding mode and parameters can be configured on-the-fly through a dedicated GUI from a host computer. To our knowledge, this is the first demonstrated multi-ASIP NoC-based FPGA prototype that is capable of decoding LDPC and Turbo (SBTC and DBTC) codes. Furthermore, an ASIC integration of the 4-DecASIP system decoder has been accomplished on the MAG3D Telecom chip designed by the CEA-LETI which targets 4G communication applications.

These results allowed the proposal of original flexible and optimized implementations for channel decoding. They further demonstrate the effectiveness of ASIP based design in this application domain to fine tune design trade-offs w.r.t. diverse design objectives.

In this thesis manuscript we firstly provided the basic background on Turbo and LDPC codes along with their construction and decoding algorithms. Max-Log MAP and Normalized Min-Sum algorithms were illustrated to be the hardware efficient versions of the MAP and Sum-Product algorithms respectively. The different parallelism levels which can be exploited in the implementation of a Turbo decoder were also explained. For LDPC decoding, a brief presentation on existing computation scheduling techniques was given. Finally, the modified NMS formulation adopted in this thesis work was presented.

The concept of ASIP-based design and the associated design methodology and tool which are considered in this thesis work have been then presented in Chapter 2. Furthermore, an overview on state-of-the-art efforts in channel decoder design was addressed. The proposed overview presents a selection of recent works related to the thesis scope in terms of flexibility support of Turbo and LDPC decoding in order to clarify the position of the proposed contributions in this thesis. The chapter has also presented the architecture of an initial ASIP for flexible Turbo decoding. This ASIP has been developed in a previous thesis study at the Electronic department of Telecom Bretagne. In this initial architecture, the main target was to explore the effectiveness of the newly proposed ASIP-design tools in terms of quality of the generated HDL code and flexibility limitations when targeting this class of applications. To that end, the target flexibility was set very high to investigate the support of any convolutional code trellis of Turbo codes. Although not supporting LDPC decoding, this architecture has investigated the exploitation of the various parallelism techniques available for Turbo decoding, particularly for DBTC. This initial effort constitutes the starting point of this thesis work.

Chapter 3 presented our first contributions in the design of scalable, flexible and optimized channel decoder supporting Turbo and LDPC codes using a multi-ASIP NoC based architecture model. Several design objectives that were targeted at the starting of this work have been achieved. Resource sharing between the LDPC and the Turbo decoding modes is achieved through efficient sharing of memories and communication network resources. In fact, the computational logic required for the low complexity NMS algorithm (LDPC mode) has relatively low contribution to the overall decoder area and has no direct commonalities with that required for the Max-Log-MAP algorithm (Turbo mode). Scalability is obtained through multi-ASIP architecture connected through an application-specific NoC interconnect. Two NoC architectures were explored: the first one is based on the binary de-Bruijn direct topology which was later replaced by a more area efficient NoC based on the Butterfly indirect topology. Thus, the final system decoder with 4 DecASIP$_{v2}$ can achieve a maximum throughput of 78 Mbps and 235 Mbps for Turbo (DBTC, SBTC) and LDPC modes respectively. When scaled to an 8 DecASIP based system decoder, the proposed architecture allows throughputs in Turbo and LDPC modes of 156 Mbps and 437 Mbps respectively. New LDPC decoding schedule adapted to the base TurbASIP architecture and to the target scalable multi-ASIP channel decoder has been proposed. Compatible with the above architectural choices, possible parallelism techniques have been explored through the use of shuffled decoding and sub-blocking in Turbo mode. In LDPC mode, partial parallelism and layered decoding have been explored where the proposed multi-DecASIP decoder processes disjoint set of check nodes within a check node group. Rapid reconfigurability between the different supported decoding modes is provided by regrouping all the parameters in a well structured configuration memory and by unifying the program memory for both SBTC and DBTC modes.

The proposed scalable multi-ASIP LDPC/Turbo decoder fairs reasonably well when compared to the recent related state of the art implementations. It achieves a high throughput in

LDPC mode with an architecture efficiency of 3.12 bit/cycle/iteration/mm^2. The Turbo mode architecture efficiency is lower achieving a 0.72 bit/cycle/iteration/mm^2, yet the throughputs target of 150 Mbps is met.

In chapter 4 we presented an FPGA prototype of a full communication system targeting the support of multi-standard Turbo/LDPC Encoding and Decoding. To our knowledge, this is the first demonstrated multi-ASIP NoC-based FPGA prototype of such flexible channel decoder supporting LDPC and Turbo (SBTC and DBTC) codes. The proposed functional prototype illustrates the effectiveness of ASIP concept in the implementation of scalable, flexible and optimized multi-standard platforms for wireless communications. The prototype supports all communication modes defined in LTE, WiFi, and WiMAX standards. To enable reasonable throughputs on FPGA most of the system components (including emitter, channel model, decoder, and system controller) have been implemented in hardware. A GUI enables the reconfiguration of the decoding mode on-the-fly and enables the monitoring of the communication system through a user friendly interface. Besides its wide flexibility, the proposed FPGA prototype achieves a throughput of 11.4 Mbps in Turbo modes and 36.5 Mbps in LDPC modes when operating at a clock frequency of 80 MHz. Furthermore, as a joint effort with another PhD student at the CEA-LETI (Pallavi Reddy), an ASIC integration of the proposed flexible channel decoder has been elaborated. A 4-DecASIP channel decoder is integrated in the latest Telecom chip (namely MAG3D) designed by the CEA-LETI targeting 4G communication applications.

Chapter 5 presented a novel parameterized architecture for multi-standard Turbo decoding, namely TDecASIP. The architecture was designed to maximize the architecture efficiency, which was achieved through the use of high sub-block parallelism degree compared to DecASIP (presented in Chapter 3). Thus, a 2 TDecASIP system decoder reaches a throughput of 150 Mbps with an AE of 4.37 bit/cycle/iteration/mm^2 outperforming related state of the art implementations. Furthermore, the proposed architecture is scalable to the degree of sub-block parallelism allowed by the standards (4 for WiMAX and 32 after Radix-4 optimization for LTE). The attainable architecture efficiency increases with sub-block parallelism degree, thus a 4-TDecASIP Turbo decoder achieves and AE of 5.88 bit/cycle/iteration/mm^2 with a throughput of 300 Mbps. Another main contribution of this work concerns the use of available ASIP design flow to design application-specific parametrized cores. The main idea was to evaluate the benefits from removing the need of a program memory and the related instruction decoder. Besides its direct impact on the architecture efficiency, such an architecture model should improve energy efficiency while keeping the benefit of ASIP design tools for high level and quick design and debugging flow. Thus, the proposed TDecASIP does not perform instruction fetch from an external program memory which has been replaced by a simple and structured finite state machine implemented in the *OperandFetch* pipeline stage. Finally, a new FPGA-based prototyping environment has been proposed with a hardware-in-the-loop approach for quick and accurate on-board validation.

Finally, the last chapter presented a flexible and optimized ASIP-based architecture for multi-standard LDPC decoding, namely LDecASIP. This new architecture was designed to maximize the architecture efficiency, which was achieved by increasing the parallelism degree and the memory bandwidth with flexible barrel shifters. LDecASIP exploits a maximum parallelism degree of 48, i.e. 48 check nodes are processed in parallel, leading to an architecture efficiency of 13.3 bit/cycle/iteration/mm^2 with a maximum throughput of 475 Mbps. It supports all LDPC codes and related parameters specified in WiFi and WiMAX standards. Another main contribution of this work concerns the design time flexibility feature of LDecASIP which allows to implement and validate new/incremental algorithm changes with fast turnaround time in design. The proposed architecture for LDecASIP LDPC decoder enables through incremental changes the future support of other QC-LDPC codes, such as those specified in DVB-S2 with high expansion factor $Z = 360$. This illustrates how the ASIP-based design approach can be used as a way to achieve both design-time and run-time flexibility features. Finally, the proposed architecture has been fully validated through an FPGA prototype based on a hardware-in-the-loop approach.

Perspectives

Regarding work perspectives, several ideas can be investigated:

- The last architecture presented (LDecASIP) for LDPC decoding was designed to serve as the base for a future work on the common architecture integrating TDecASIP for Turbo decoding. The resulting architecture should lead to a high architecture efficiency in both modes.
- Increasing the architecture efficiency in terms of bits/cycle/iteration/mm^2 leads to optimal usage of available hardware resources which should optimizes energy consumption. It would be interesting to evaluate the impact of the explored diverse design choices on energy efficiency and to investigate the integration of low power design techniques.
- 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

De nombreuses techniques de codage de canal sont spécifiées dans les nouvelles normes de communications numériques, chacune adaptée à des besoins applicatifs spécifiques (taille de trame, type de canal de transmission, rapport signal-à-bruit, bande-passante, etc.). Si l'on considère les applications naissantes multi-mode et multi-standard, ainsi que l'intérêt croissant pour la radio logicielle et la radio cognitive, la combinaison de plusieurs techniques de correction d'erreur devient incontournable. Néanmoins, 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. Le tableau (Table 1) donne un panel représentatif de normes mobiles sans fil afin de mettre en évidence leurs différences en termes de taux de transfert et de mode de codage de canal. Les codes correcteurs d'erreurs les plus communément utilisés dans ces normes sont des codes convolutifs (CC), les turbocodes (TCSB: Turbo Codes Simples Binaires, et TCDB: Turbo Codes Doubles Binaires) et les codes LDPC (Low-Density Parity-Check codes).

Problèmes

Dans le contexte décrit ci-dessus, le décodage canal représente un des composants les plus exigeants en terme de capacités de calcul, de communications, de mémoire, et donc de consommation d'énergie. La conception de décodeurs de canal a été largement étudiée au cours des dernières années et plusieurs implémentations ont été proposées. Certaines de ces implémentations ont réussi à atteindre de très hauts débits pour des normes spécifiques grâce à l'adoption d'architectures dédiées qui fonctionnent comme des accélérateurs hardwares. Toutefois, ces réalisations ne prennent pas en compte la flexibilité d'application et l'évolutivité. En effet, cette approche implique l'attribution de plusieurs accélérateurs hardwares distincts pour réaliser des systèmes multistandards, ce qui se traduit souvent, par une mauvaise optimisation de l'utilisation des ressources hardware. De plus, ceci implique un temps de conception long, incompatible avec les contraintes de délais de mise sur le marché et l'avènement continu de nouvelles normes et d'applications.

Plus récemment, plusieurs contributions ont été proposées pour l'implémentation de décodeurs canal, les rendant flexibles tout en conservant un haut débit. La flexibilité varie de la prise en charge de plusieurs modes de communication monostandard, à la prise en charge d'applications multistandards et multimodes. D'autres implémentations ont proposé d'augmenter la flexibilité cible afin de prendre en charge différentes techniques de codage canal. En réalité, un fossé s'est rapidement creusé ces dernières années entre le besoin de flexibilité dans le domaine du traitement numérique en bande de base des systèmes de communication modernes, et la disponibilité réelle d'implémentations flexibles et efficaces avec la prise en charge de la reconfigurabilité. Les principales raisons de cet écart sont, d'une part, la faible efficacité en terme de surface et de consommation des solutions flexibles proposées jusqu'à maintenant et d'autre part l'augmentation considérable des coûts d'ingénierie non-récurrente (NRE) dans la production de circuits intégrés dédiés à des applications spécifiques (ASIC) avec des nouvelles technologies utilisant des semi-conducteurs.

Objectifs et portée de la thèse

Ce travail de thèse vise à définir et à développer un modèle d'architecture de décodage canal flexible et haut débit pour les systèmes de communications numériques émergents et futurs. Il est nécessaire d'optimiser les solutions en termes de performances, de surface et de consommation d'énergie. Ceci ne peut être négligé au profit d'une plus grande flexibilité. L'objectif de ce travail est d'allier les approches augmentant la flexibilité et l'optimisation. L'objectif principal est de fournir des outils se présentant comme un jeu de construction pouvant être assemblé à la guise afin de trouver le meilleur compromis entre très grande flexibilité et très grande spécificité/optimisation pour une application donnée. Pour atteindre cet objectif, le travail de thèse étudie le multitraitement et les jeux d'instructions de processeur à application spécifique (ASIP) qui permettent au concepteur d'ajuster librement la flexibilité par rapport à la performance conformément aux exigences de l'application considérée.

De nouvelles contributions sur le sujet ont été publiées récemment, elles visent l'amélioration de l'efficacité de l'architecture résultante, en termes de compromis performances/surface et de flexibilité. En se concentrant sur les techniques Turbo et LDPC, des architectures de décodage canal multi-ASIP ciblant une grande flexibilité alliée à une grande efficacité architecturale (AE) en terme de bits/cycle/itération/mm^2 ont été proposées. Dans ces contributions, différentes solutions architecturales et approches de conception sont explorées. Afin d'être pertinent par rapport aux normes existantes et émergentes, nous limitons la flexibilité à la prise en charge des codes LDPC et Turbo spécifiées dans le WiFi, WiMAX et LTE. Cela permet également de les comparer aux implémentations actuelles.

Contributions

Dans ce contexte, trois contributions originales ont été proposées.

La première concerne la conception d'un décodeur multi-ASIP LDPC/Turbo à haut débit, évolutif et flexible. Plusieurs objectifs de conception ont été atteints dans ce travail en termes d'évolutivité, de partage des ressources, et de vitesse de configuration entre les différents modes de décodage supportés. Le DecASIP proposé prend en charge le décodage des codes LDPC et Turbo spécifiés dans les normes WiFi, WiMAX et LTE. Un décodeur 8-DecASIP atteint une efficacité d'architecture (AE) de $0,7$ en mode Turbo et $3,12$ en mode LDPC. Les débits obtenus sont de 156 Mbps et 437 Mbps respectivement en mode Turbo et LDPC.

La deuxième contribution porte sur la conception d'un ASIP paramétrable pour turbo-décodage (TDecASIP). Ici, l'objectif était de déterminer l'efficacité d'architecture maximale pour un turbo-décodage ASIP tout en maximisant les parallélismes de sous-blocs. Le décodeur 4-TDecASIP proposé, atteint une efficacité d'architecture (AE) de $5,3$ et un débit de 300 Mbps dans les deux modes, SBTC et DBTC. De plus, nous avons démontré qu'avec cette architecture, il était possible de paramétrer les cœurs de façon spécifique aux applications en utilisant la méthodologie ASIP.

La troisième contribution correspond à la conception d'un ASIP optimisé pour le décodage LDPC (LDecASIP). En ce qui concerne le LDecASIP, l'objectif était d'étudier l'efficacité d'architecture maximale pour un décodage LDPC ASIP, en augmentant le degré de parallélisme et la bande passante de la mémoire grâce à des registres à décalage flexibles. Le décodeur LDecASIP réalisé a atteint un maximum d'efficacité d'architecture (AE) de $13,6$ et un débit maximal de 475 Mbps.

La quatrième contribution principale de ce travail de thèse porte sur le projet de décodeur-canal multi-ASIP flexible et totalement fonctionnel. Une plate-forme simulant un système de communication complet a été développé sur cible FPGA. Cette plate-forme embarque un encodeur Turbo, un encodeur LDPC, un modèle de canal, un décodeur canal 4-DecASIP (proposé dans ce projet) et un compteur d'erreurs. Le mode de décodage et les paramètres peuvent être configurés à la volée à l'aide d'une interface graphique dédiée sur ordinateur. À notre connaissance, c'est la première fois qu'un tel prototype FPGA multi-ASIP à base de NoC est capable de décoder les codes LDPC et Turbo (SBTC et DBTC). En outre, une intégration ASIC du décodeur système 4-DecASIP a été réalisée sur la puce MAG3D Telecom conçu par le CEA-LETI qui cible les applications de communication 4G.

Structure du manuscrit

Dans ce manuscrit de thèse, nous avons tout d'abord fourni les connaissances de base sur les codes LDPC et les turbocodes avec leur construction et leurs algorithmes de décodage. Les algorithmes Max-Log MAP et Min-Sum normalisé ont été utilisés comme les versions matérielles efficaces des algorithmes MAP et Sum-produit. Les niveaux de parallélisme différents qui peuvent être exploitées dans l'implémentation d'un décodeur Turbo a aussi été présenté. Pour le décodage LDPC, une brève présentation sur les techniques existantes d'ordonnancement a été donnée. Enfin, la formulation modifiée NMS adoptée dans ce travail de thèse a été présentée.

Ensuite, le concept du développement basé sur les ASIP, la méthodologie associée ainsi que les outils considérés dans ce travail de thèse ont été présentés dans le chapitre deux. En outre, un état de l'art sur les travaux dans le domaine de de la conception de décodage canal a été abordé. Cet état le l'art présente une sélection de contributions récentes relatives aux travaux de cette thèse en termes de support et de flexibilité de décodage Turbo et LDPC afin de mettre en évidence la position des contributions proposées dans cette thèse. Dans ce chapitre à aussi été présenté l'architecture d'un ASIP initiale de turbo-décodage flexibles. Cet ASIP a été développé dans une précédente thèse au département électronique de Télécom Bretagne. Pour cette architecture initiale, l'objectif principal était d'étudier l'efficacité de l'approche basée sur les ASIP et des outils associé en termes de qualité du code HDL généré et les limites de flexibilité lors du ciblage de cette classe d'applications. À cette fin, la flexibilité ciblée était très haute afin d'étudier le support de tous les treillis possibles en code Turbo et codes convolutifs. Bien que ne prenant pas en compte le décodage LDPC, ce travail s'est porté sur l'exploitation des techniques de parallélisme disponibles pour les turbo-décodage, en particulier pour TCDB. Cet effort initial constitue le point de départ de ce travail de thèse.

Dans le chapitre 3 nous avons présenté nos premières contributions à la conception de décodeurs canal évolutifs, flexibles et optimisé supportant des codes LDPC et Turbo à l'aide d'un modèle multi-ASIP basée sur une architecture autour d'un NoC (Figure 3.4). Plusieurs objectifs de conception qui ont été ciblées au départ de ce travail ont été atteints. Le partage des ressources entre les LDPC et les modes Turbo décodage est réalisé par un partage efficace des mémoires et des ressources réseau de communication. En revanche, la ressource en calcul nécessaire pour l'algorithme de faible complexité NMS (mode LDPC) prend une très faible part de la surface globale du décodeur n'a pas de partie commune direct avec celle requise pour l'algorithme Max-Log-MAP (en mode Turbo) (Figure 3.25). Évolutivité est obtenue par l'architecture multi-ASIP reliés par une interconnexion sous forme de réseau sur puce (NoC). Deux architectures NoC ont été explorées: la première est basée sur la topologie directe binaire de-Bruijn qui a ensuite été remplacé par un NoC plus efficace basé sur la topologie butterfly indirecte. Ainsi, le décodeur avec 4-DecASIP$_{v2}$ peut atteindre un débit maximal de 78 Mbps et 235 Mbps respectivement pour les modes Turbo (DBTC, SBTC) et LDPC. Lorsque l'on extrapole pour un système utilisant 8 DecASIP basé sur l'architecture proposée les débits en modes Turbo et LDPC montent respectivement à de 156 Mbps et 437 Mbps. Un nouvel ordonnancement pour le LDPC adapté à l'architecture de base TurbASIP et à celle du décodeur multi-ASIP à été proposé.

Compatible avec les choix architecturaux ci-dessus, différentes techniques de parallélisme potentielles ont été explorées notamment l'ordonnacementy shuffle et le découpage de trame en sous bloc en mode Turbo. En mode LDPC, le parallélisme partielle et le décodage layered ont été explorées, où le décodeur multi-DecASIP proposé traite des ensemble disjoint de check-node d'un groupe de check-node. La reconfigurabilité rapide entre les différents modes de décodage pris en charge est assurée par le regroupement de tous les paramètres de configuration dans une mémoire bien structuré et en unifiant la mémoire de programme pour les deux modes TCSB et TCDB. Le décodeur évolutif LDPC/Turbo multi-ASIP, ici présenté soutient la comparaison par rapport à ceux trouvé dans les contributions de l'état de l'art. Il atteint un haut débit en mode LDPC avec une efficacité d'architecture de $3,12$ bits/cycle/itération/mm^2. L'efficacité d'architecture en mode Turbo est inférieure, elle atteint 0,72/cycle/itération/mm^2, néanmoins l'objectif d'un débit de 150 Mbps est atteint.

Dans le chapitre 4, nous avons présenté le prototype FPGA d'un système complet de communication supportant le codage et le décodage Turbo/LDPC multi-standard. À notre connaissance, c'est le premier prototype de décodeur de canal FPGA multi-ASIP basé sur des NoC, flexible prenant en charge les codes LDPC et Turbo (SBTC et DBTC). Le prototype proposé (Figure 4.1) parfaitement fonctionnel, illustre l'efficacité du concept de l'ASIP dans l'implémentation de plateformes multi-standards évolutives, flexibles et optimisés pour la communication sans fil. Le prototype prend en charge tous les modes de communication définis dans les normes LTE, WiFi et WiMAX. Afin de permettre des débits raisonnables sur FPGA, la plupart des composants du système (y compris l'émetteur, le modèle de canal, le décodeur, et le contrôleur système) ont été déportés sur le FPGA. Une interface utilisateur graphique permet une configuration du mode de décodage et un contrôle du système de communication aisée. En plus de sa grande flexibilité, le prototype FPGA proposée permet d'atteindre un débit de $11,4$ Mbps en mode Turbo et $36,5$ Mbps en mode LDPC en fonctionnant à une fréquence d'horloge de 80 MHz. Par ailleurs, en collaboration avec une doctorante au CEA-LETI (Pallavi Reddy), une intégration ASIC d'un tel décodeur flexible à été réalisée. Un décodeur canal 4-DecASIP est intégré dans la dernière puce Telecom (à savoir MAG3D) conçu par le CEA-LETI ciblant les applications de communication 4G. De plus, ce décodeur canal intégré sur ASIC, embarque des techniques de réduction de puissance/énergie qui ont été proposées par Pallavi Reddy à différents niveaux (niveaux algorithmique, de l'architecture et de la technologie).

Le chapitre 5 détaille une nouvelle architecture paramétrée pour le décodage turbo multi-standard, à savoir le TDecASIP. Par rapport à DecASIP (présentée dans le chapitre 3), l'architecture a été conçue pour maximiser l'efficacité architectural, grâce à l'utilisation d'un degré élevé de parallélisme sous-bloc. Ainsi, un décodeur système de 2 TDecASIP atteint un débit de 150Mbps avec une AE de $4,37$ bits/cycle/itération/mm^2, dépassant aisément les performances des décodeurs actuels trouvés dans la littérature. De plus, l'architecture proposée peut s'adapter au degré de parallélisme sous-bloc autorisé par les normes (4 pour le WiMAX et 32 pour le LTE avec l'optimisation Radix-4). L'efficacité maximal de l'architecture augmente avec le degré de parallélisme sous bloc, donc un 4-TDecASIP Turbo décodeur réalise un AE de $5,88$ bits/cycle/itération/mm^2, avec un débit de 300 Mbps. Une autre contribution principale de ce travail concerne l'utilisation de la méthodologie de conception ASIP pour concevoir des coeurs de calcul paramétrés spécifique à une application. L'idée principale était d'évaluer les avantages de la suppression de la mémoire programme et des instructions décodeur associées. Outre son impact direct sur l'efficacité architectural, un tel modèle d'architecture devrait permettre d'améliorer l'économie d'énergie tout en conservant le bénéfice des outils ASIP pour leur rapidité et leur qualité de conception et débogage. Ainsi, ce TDecASIP n'éxécute pas d'instruction récupérée à partir d'une mémoire de programme externe, ceci a été remplacé par un simple automate à états finis structuré intégré au pipeline *OperandFetch*. Enfin, un nouvel environnement de prototypage basé sur FPGA a été proposé avec une approche hardware-in-the-loop pour une utilisation rapide et précise de validation embarquée.

Enfin, le dernier chapitre présente le LDecASIP (Figure 6.1), une architecture flexible et optimisée basée sur le principe des ASIP, pour le décodage LDPC multi-standard. Cette nouvelle architecture a été conçue pour maximiser l'efficacité architectural obtenue en augmentant le degré de parallélisme et la bande passante de la mémoire grâce à des registres à décalage (barrel shifters) flexibles. LDecASIP exploite un degré de parallélisme maximal de 48, soit 48 check-nodes traités en parallèle, avec un efficacité architectural de $13,3$ bits/cycle/itération/mm^2 avec un débit maximal de 475 Mbps. Il prend en charge tous les codes LDPC et les paramètres associés spécifiés dans les normes WiFi et WiMAX. Une autre contribution principale de ce travail concerne la flexibilité temporelle de conception de LDecASIP, qui permet d'implèmenter et de valider les changements successifs d'algorithme avec une réacivité rapide lors de la conception. L'architecture proposée pour le décodeur LDPC LDecASIP, permet par des changements progressifs, la prise en charge future d'autres codes QC-LDPC, tels que ceux spécifiés en DVB-S2 avec des facteurs d'expansion élevés $Z = 360$. Cet exemple illustre le fait que l'approche de conception basée sur l'ASIP peut être utilisé comme un moyen d'atteindre à la fois des temps de conception, et des temps d'éxécution modulables. Enfin, l'architecture proposée a été entièrement validée par un prototype FPGA basé sur une approche hardware-in-the-loop.

Perspectives proposées

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

- La dernière architecture présentée (LDecASIP) pour décodage LDPC a été conçu pour servir de base à un futur travail sur une architecture commune intégrant TDecASIP pour le décodage Turbo. L'architecture résultante devrait conduire à une haute efficacité d'architecture dans les deux modes.

- L'augmentation de l'efficacité d'architecture en termes de bits / cycle / itération / mm^2 conduit à une utilisation optimale des ressources matérielles disponibles, ce qui devraient optimiser la consommation énergétique. Il serait intéressant d'évaluer l'impact des différents choix de conception étudiés sur l'économie d'énergie et d'étudier l'intégration des techniques de conception à faible puissance.

- Explorer des techniques à d'autre niveau d'architecture telle que l'utilisation du concept de reconfiguration dynamique associée à l'approche de conception ASIP. Avoir un réseau dynamiquement reconfigurable attaché au pipeline ASIP peut permettre un meilleur partage des ressources et d'utilisation des différents paramètres et variables de l'algorithme.

- Explorer d'autres technologies émergentes clés telles que l'intégration 3D qui peut permettre de nouvelles améliorations en matière d'énergie, de vitesse de reconfiguration et d'efficacité architecture.

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
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
FER	Frame Error Rate
FEC	Forward Error Correction
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FS	Flooding Schedule
FSM	Finite State Machine
GALS	Globally Asynchronous Locally Synchronous
GSM	Global System for Mobile Communications
GUI	Graphical User Interface

HDL	Hardware Description Language
HSS	Horizontal Shuffle Scheduling
IP	Intellectual Property
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
MPSoC	Multiple Processor System on Chip
NI	Network Interface
NMS	Normalized Min-Sum algorithm
NoC	Network on Chip
OFDM	Orthogonal Frequency Division Multiplexing
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
SDR	Software Defined Radio
SIMD	Single Instruction Multiple Data
SISO	Soft In Soft Out
SNR	Signal to Noise Ratio
SoC	System on Chip
SPC	Single Parity Check Algorithm
SOVA	Soft Output Viterbi Algorithm
TPMP	Two-Phase Message Passing
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
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
ZOL	Zero Overhead Loop

Notations

Channel Coding:

X_i	Coded symbol of sequence i
Y_i	Modulated symbol of sequence 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:

d_i	Information symbol
$\alpha_i(s)$	Decoder forward recursion metrics for i^{th} bit and s in Max-log-MAP algorithm
$\alpha\_int(w_{(n)}^{iter})(i)$	Initial forward recursion metrics of i^{th} symbol of window n in iteration $iter$
$\beta_i(s)$	Decoder backward recursion metrics (Max-log-MAP algorithm)
$\beta\_int(w_{(n)}^{iter})(i)$	Initial backward recursion metrics of i^{th} symbol of window n in iteration $iter$
$\gamma(s', s)$	Decoder branch metrics in Max-log-MAP algorithm
γ^{ext}	Turbo decoder extrinsic information LLR
γ^{intr}	Turbo decoder intrinsic information LLR
γ^{apos}	Turbo decoder *a posteriori* information LLR
$\gamma^{n.app}$	Turbo decoder normalized *a priori* information LLR
$\gamma^{Hard.dec}$	Turbo decoder hard decision
S_c	Extrinsic scaling factor

LDPC Decoding:

M	Number of check nodes
N	Number of variable nodes
M_b	Number of block rows in a QC-LDPC check matrix
N_b	Number of block columns in a QC-LDPC check matrix
m	Check node m
n	Variable node n
Z	Permutation matrix size or expansion matrix size
H_{base}	Base parity check matrix described in terms of permutation values
H	Parity check matrix
$L^i(n,m)$	Variable node message from n to m, for sub-iteration i

$L^i(m,n)$	Check node message from m to n, for sub-iteration i
$L^i_{ext}(m,n)$	Extrinsic check node message from m to n, for sub-iteration i
$L^i(n)$	a posteriori LLR for sub-iteration i
Δ_n	Channel value for variable node n
$min0^{k1}_{T0}(m)$	Overall minimum of the received $\vert L(n,m)\vert$ for ASIP k1 at time T0
$min1^{k1}_{T0}(m)$	Second minimum of the received $\vert L(n,m)\vert$ for ASIP k1 at time T0
$\mathbb{P}_{x,y}$	Permutation value of the sub-matrix at block row x and block column y
$\mathbb{P}^{VNG_y}_x(m)$	Permutation value which results in the variable node of VNG$_y$ connected to the check node m of the x^{th} check node group
P	Number of check nodes processed in parallel
$LOC(m)$	The index of the connected VN$_m$ providing $min0$
P_{Deg}	Parallelism degree in LDPC mode in LDecASIP
$RV^k_T(m)$	Running Vector group $[min0, min1, ind, sgn](m)$, for ASIP k at time T
$sgn(m)$	Product of the signs of the received $L(n,m)$
$UV^k(m)$	Update Vector of variable node m containing from ASIP k $[min0, min1, ind, sgn]$

Others:

$Bits_{sym}$	Bits per symbol
C_{rate}	Coding rate
f_{clk}	Operational frequency
N_{Iter}	Number of iteration
N_{Instr}	Number of instructions
f	technology Feature size
NA	Normalized Area
N_A	Number of ASIPs
AE	Architecture Efficiency

Bibliography

[1] C. E. Shannon, “A mathematical theory of communication,” *Bell system technical journal*, vol. 27, 1948.

[2] C. Wang, D. Sklar, and D. Johnson, “Forward Error-Correction Coding,” *Crosslink, the Aerospace Corportaion magazine of advances in aerospace technology, accessible online: http://aerospace.wpengine.netdna-cdn.com/wp-content/uploads/crosslink/V3N1.pdf*, vol. 3, no. 1, Winter 2001/2002, pp. 26–29.

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

[4] “3GPP TS 36.212: Multiplexing and channel coding, version 8.4.0, Sept. 2008 .”

[5] *802.16 IEEE Standard for Local and metropolitan area networks, Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems*, Std., 2005.

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

[7] C. Douillard, M. Jezequel, C. Berrou, J. Tousch, N. Pham, and N. Brengarth, “The Turbo Code Standard for DVB-RCS,” in *Proc. of the 2nd International Symposium on Turbo Codes & Related Topics, Brest, France*, 2000, pp. 535 – 538.

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

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

[10] O. Takeshita and J. Daniel, “New deterministic interleaver designs for Turbo codes,” *IEEE Transactions on Information Theory*, vol. 46, no. 6, pp. 1988–2006, 2000.

[11] 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.

[12] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error-correcting coding and decoding: Turbo-codes,” in *Proc. of the IEEE International Conference on Communications (ICC 93)*, vol. 2, May 1993, pp. 1064 –1070.

[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)*, Nov. 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*, pp. 284 – 287, Mar. 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 IEEE International Conf. on Communications*, vol. 2, jun 1995, pp. 1009 –1013.

[16] O. Muller, “Architectures multiprocesseurs monopuces génériques pour turbo-communications haut-débit,” Ph.D. dissertation, Institut Mines-Télécom-Télécom Bretagne-UEB, 2007.

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

[18] Y. Zhang and K. Parhi, “Parallel Turbo decoding,” in *Proc. of the International Symposium on Circuits and Systems (ISCAS)*, vol. 2, 2004.

[19] Z. Wang, “High-Speed Recursion Architectures for MAP-Based Turbo Decoders,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 4, pp. 470 –474, Apr. 2007.

[20] H. Moussa, “Architectures de réseaux sur puce pour décodeurs canal multiprocesseurs,” Ph.D. dissertation, Institut Mines-Télécom-Télécom Bretagne-UEB, 2009.

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

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

[23] D. MacKay, “Good error-correcting codes based on very sparse matrices,” in *Proc. of the IEEE International Symposium on Information Theory*, jun 1997, p. 113.

[24] T. Richardson and R. Urbanke, “The capacity of low-density parity-check codes under message-passing decoding,” *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 599 –618, feb 2001.

[25] S.-Y. Chung, J. Forney, G.D., T. Richardson, and R. Urbanke, “On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit,” *IEEE Communications Letters*, vol. 5, no. 2, pp. 58 –60, feb 2001.

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

[27] M. Mansour and N. Shanbhag, “High-throughput LDPC decoders,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 976 –996, Dec. 2003.

[28] *802.11n Local and metropolitan area networks,Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, Std., 2009.

[29] 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.

[30] J. Hagenauer, E. Offer, and L. Papke, “Iterative decoding of binary block and convolutional codes,” *IEEE Transactions on Information Theory*, vol. 42, pp. 429–445, 1996.

[31] P. Ienne and R. Leupers, *”Customizable Embedded Processors–Design Technologies and Applications”*. Morgan Kaufmann, 2006.

[32] “CoWare Processor Designer Homepage,” *http://www.synopsys.com/Systems/BlockDesign/ProcessorDev*.

[33] "ARC Configurable Cores Homepage," *http://www.synopsys.com/IP/ProcessorIP/ARCProcessors*.

[34] "Tensilica Xtensa 7 Homepage," *http://www.tensilica.com/products/x7_processor_generator.htm*.

[35] "Stretch Software-Configurable Processors Homepage," *http://www.stretchinc.com/technology/*.

[36] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *IEEE Transactions on Design Test of Computers*, vol. 22, no. 2, pp. 90 – 101, Mar. 2005.

[37] "Target IP Designer Homepage," *http://www.retarget.com/resources.php*.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] NEWCOM++ (NoE FP7), "Report on the state for the art on hardware architectures for flexible radio and intensive signal processing," *http://www.newcom-project.eu:8080/Plone/public-deliverables/research/DR.C.1_final.pdf*.

[43] 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 International Conference Computer Aided Design (ICCADICCAD)*, 2001, pp. 625–630.

[44] J.-M. Hsu and C.-L. Wang, "A parallel decoding scheme for turbo codes," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, Jun. 1998, pp. 445 –448.

[45] Z. Wang, H. Suzuki, and K. Parhi, "VLSI implementation issues of TURBO decoder design for wireless applications," in *Proc. of the IEEE Workshop on Signal Processing Systems (SiPS)*, 1999, pp. 503 –512.

[46] T. W. Kwon, D. W. Kim, W. T. Kim, E. K. Joo, J. R. Choi, P. Choi, J. J. Kong, S. H. Choi, W. H. Chung, and K. W. Lee, "A modified two-step SOVA-based turbo decoder for low power and high performance," in *Proc. of the IEEE Region 10 Conference (TENCON)*, vol. 1, 1999, pp. 297 –300.

[47] C. Chaikalis and J. Noras, "Implementation of an improved reconfigurable sova/log-map turbo decoder in 3gpp," in *Proc. of the Third International Conference on 3G Mobile Communication Technologies*, May 2002, pp. 146 – 150.

[48] E. Boutillon, C. Douillard, and G. Montorsi, "Iterative Decoding of Concatenated Convolutional Codes: Implementation Issues," *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1201 –1227, Jun. 2007.

[49] D. Gnaedig, "Optimisation des architectures de décodage des turbo-codes," Ph.D. dissertation, Université de Bretagne-Sud, Institut Mines-Télécom-Télécom Bretagne-UEB, 2005.

[50] F. Viglione, G. Masera, G. Piccinini, R. Ruo Roch, and M. Zamboni, "A 50 Mbit/s iterative turbo-decoder," in *Proc. of the ACM/IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2000, pp. 176 –180.

[51] Z. Wang, H. Suzuki, and K. Parhi, "Efficient approaches to improving performance of VLSI SOVA-based turbo decoders," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2000, pp. 287 –290.

[52] "TMS320C64x DSP Turbo-Decoder Coprocessor," *http://www.ti.com/lit/ug/spru534b/spru534b.pdf*.

[53] K. Loo, T. Alukaidey, and S. Jimaa, "High performance parallelised 3gpp turbo decoder," in *Proc. of the 5th European Personal Mobile Communications Conference (2003)*, Apr. 2003, pp. 337 – 342.

[54] Z. Zhong, T. Peng, Z. Zhong, W. Wang, and Z. Liu, "Hardware implementation of turbo coder in lte system based on picochip pc203," in *Proc. of the 12th IEEE International Conference on Communication Technology (ICCT)*, Nov. 2010, pp. 995 –998.

[55] F. Gilbert, M. Thul, and N. Wehn, "Communication centric architectures for turbo-decoding on embedded multiprocessors," in *Proc. of the ACM/IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2003, pp. 356 – 361.

[56] C.-C. Wong, Y.-Y. Lee, and H.-C. Chang, "A 188-size 2.1mm2 reconfigurable turbo decoder chip with parallel architecture for 3GPP LTE system," in *Proc. of the Symposium on VLSI Circuits*, Jun. 2009, pp. 288 –289.

[57] D.-S. Cho, H.-J. Park, and H.-C. Park, "Implementation of an efficient UE decoder for 3G LTE system," in *Proc. of the International Conference on Telecommunications (ICT)*, Jun. 2008, pp. 1 –5.

[58] D. Wu, R. Asghar, Y. Huang, and D. Liu, "Implementation of a high-speed parallel Turbo decoder for 3GPP LTE terminals," in *Proc. of the IEEE 8th International Conference on ASIC (ASICON)*, Oct. 2009, pp. 481 –484.

[59] J.-H. Kim and I.-C. Park, "A unified parallel radix-4 turbo decoder for mobile WiMAX and 3GPP-LTE," in *Proc. of the IEEE Custom Integrated Circuits Conference (CICC)*, 2009, pp. 487 –490.

[60] 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.

[61] M. May, T. Ilnseher, N. Wehn, and W. Raab, “A 150Mbit/s 3GPP LTE Turbo code decoder,” in *Proc. of the Design, Automation Test in Europe Conference Exhibition (DATE)*, 2010, pp. 1420 –1425.

[62] T. Ilnseher, F. Kienle, C. Weis, and N. Wehn, “A 2.15GBit/s turbo code decoder for LTE advanced base station applications,” in *Proc. of the International Symposium on Turbo Codes and Iterative Information Processing (ISTC’12)*, Aug. 2012, pp. 21 –25.

[63] O. Muller, A. Baghdadi, and M. Jézéquel, “ASIP-Based Multiprocessor SoC Design for Simple and Double Binary Turbo Decoding,” in *Proc. Design, Automation and Test in Europe (DATE), Munich, Germany*, Mar. 2006, pp. 1330–1335.

[64] H. Moussa, O. Muller, A. Baghdadi, and M. Jezequel, “Butterfly and Benes-Based on-Chip Communication Networks for Multiprocessor Turbo Decoding,” in *Proc. of the ACM/IEEE Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, Apr. 2007, pp. 1–6.

[65] M. J. Thul, F. Gilbert, and N. Wehn, “Optimized concurrent interleaving architecture for high-throughput Turbo decoding,” in *Proc. of the IEEE International Conference on Electronics, Circuits and Systems*, 2002, pp. 1099–1102.

[66] H. Moussa, O. Muller, A. Baghdadi, and M. Jezequel, “Butterfly and Benes based on chip communication networks for multiprocessor Turbo decoding,” in *Proc. of the ACM/IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2007, pp. 654–659.

[67] A. Blanksby and C. Howland, “A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder,” *IEEE Journal of Solid-State Circuits*, pp. 404 –412, Mar. 2002.

[68] D. Hocevar, “A reduced complexity decoder architecture via layered decoding of LDPC codes,” in *Proc. of the IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 2004, pp. 107 – 112.

[69] B. Levine, R. Reed Taylor, and H. Schmit, “Implementation of near Shannon limit error-correcting codes using reconfigurable hardware,” in *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 217 –226.

[70] F. Verdier and D. Declercq, “A low-cost parallel scalable FPGA architecture for regular and irregular LDPC decoding,” *IEEE Transactions on Communications*, pp. 1215 –1223, Jul. 2006.

[71] S. Kim, G. Sobelman, and J. Moon, “Parallel VLSI architectures for a class of LDPC codes,” in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 2, 2002, pp. 93 – 96.

[72] T. Zhang and K. Parhi, “A 54 Mbps (3,6)-regular FPGA LDPC decoder,” in *Proc. of the IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 2002, pp. 127 – 132.

[73] M. Mansour and N. Shanbhag, “A 640-Mb/s 2048-bit programmable LDPC decoder chip,” *IEEE Journal of Solid-State Circuits*, pp. 684 – 698, Mar. 2006.

[74] Z. Wang and Z. Cui, “Low-Complexity High-Speed Decoder Design for Quasi-Cyclic LDPC Codes,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 104 –114, Jan. 2007.

[75] Y. Dai, Z. Yan, and N. Chen, "High-Throughput Turbo-Sum-Product Decoding of QC-LDPC Codes," in *Proc. of the 40th Annual Conference on Information Sciences and Systems*, Mar. 2006, pp. 839 –844.

[76] M. Karkooti, P. Radosavljevic, and J. Cavallaro, "Configurable, High Throughput, Irregular LDPC Decoder Architecture: Tradeoff Analysis and Implementation," in *Proc. of the International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Sept. 2006, pp. 360 –367.

[77] S.-H. Kang and I.-C. Park, "Loosely coupled memory-based decoding architecture for low density parity check codes," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1045 – 1056, May 2006.

[78] "Encyclopedia of sparse graph codes, D. Mackay." *http://www.inference.phy.cam.ac.uk/mackay/codes/data.html.*

[79] "Software for low density parity check (LDPC) codes, R. M. Neal." *http://www.cs.utoronto.ca/~radford/ldpc.software.html.*

[80] T. Brack, M. Alles, F. Kienle, and N. Wehn, "A Synthesizable IP Core for WIMAX 802.16E LDPC Code Decoding," in *Proc. of the IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications*, Sept. 2006, pp. 1 –5.

[81] K. Gunnam, G. Choi, M. Yeary, and M. Atiquzzaman, "VLSI Architectures for Layered Decoding for Irregular LDPC Codes of WiMax," in *Proc. of the IEEE International Conference on Communications (ICC'07)*, Jun. 2007, pp. 4542 –4547.

[82] X.-Y. Shih, C.-Z. Zhan, C.-H. Lin, and A.-Y. Wu, "An 8.29 mm^2 52 mW Multi-Mode LDPC Decoder Design for Mobile WiMAX System in 0.13μm CMOS Process," *IEEE Journal of Solid-State Circuits*, pp. 672 –683, Mar. 2008.

[83] K. Zhang, X. Huang, and Z. Wang, "High-throughput layered decoder implementation for quasi-cyclic LDPC codes," *IEEE Journal on Selected Areas in Communications*, pp. 985 –994, Aug. 2009.

[84] M. Rovini, G. Gentile, F. Rossi, and L. Fanucci, "A Scalable Decoder Architecture for IEEE 802.11n LDPC Codes," in *Proc. of the IEEE Global Telecommunications Conference (GLOBECOM '07)*, Nov. 2007, pp. 3270 –3274.

[85] J. Jin and C.-Y. Tsui, "A low power layered decoding architecture for LDPC decoder implementation for IEEE 802.11n LDPC codes," in *Proc. of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2008, pp. 253 –258.

[86] T. Brack, M. Alles, T. Lehnigk-Emden, F. Kienle, N. Wehn, N. L'Insalata, F. Rossi, M. Rovini, and L. Fanucci, "Low Complexity LDPC Code Decoders for Next Generation Standards," in *Proc. of the ACM/IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, Apr. 2007, pp. 1 –6.

[87] Y. Sun, M. Karkooti, and J. R. Cavallaro, "High Throughput, Parallel, Scalable LDPC Encoder/Decoder Architecture for OFDM Systems," in *Proc. of the IEEE Dallas/CAS Workshop on Design, Applications, Integration and Software*, Oct. 2006, pp. 39 –42.

[88] C.-H. Liu, C.-C. Lin, S.-W. Yen, C.-L. Chen, H.-C. Chang, C.-Y. Lee, Y.-S. Hsu, and S.-J. Jou, "Design of a Multimode QC-LDPC Decoder Based on Shift-Routing Network," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 734 –738, Sept. 2009.

[89] C.-H. Liu, C.-C. Lin, H.-C. Chang, C.-Y. Lee, and Y. Hsua, "Multi-mode message passing switch networks applied for QC-LDPC decoder," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2008, pp. 752 –755.

[90] Y. Sun and J. Cavallaro, "A low-power 1-Gbps reconfigurable LDPC decoder design for multiple 4G wireless standards," in *Proc. of the IEEE International SOC Conference*, Sept. 2008, pp. 367 –370.

[91] X. Zhang, Y. Tian, J. Cui, Y. Xu, and Z. Lai, "An multi-rate LDPC decoder based on ASIP for DMB-TH," in *Proc. of the IEEE 8th International Conference on ASIC (ASICON)*, Oct. 2009, pp. 995 –998.

[92] L. Dinoi, R. Martini, G. Masera, F. Quaglio, and F. Vacca, "ASIP design for partially structured LDPC codes," *Electronics Letters*, pp. 1048 –1049, 31 2006.

[93] T. Theocharides, G. Link, N. Vijaykrishnan, and M. Irwin, "Implementing LDPC decoding on network-on-chip," in *Proc. of the 18th International Conference on VLSI Design*, Jan. 2005, pp. 134 – 137.

[94] F. Quaglio, F. Vacca, C. Castellano, A. Tarable, and G. Masera, "Interconnection framework for high-throughput, flexible LDPC decoders," in *Proc. of the ACM/IEEE Design, Automation and Test in Europe (DATE)*, Mar. 2006, p. 6.

[95] F. Vacca, H. Moussa, A. Baghdadi, and G. Masera, "Flexible architectures for LDPC decoders based on network on chip paradigm," in *Proc. of the 12th Euromicro Conference on Digital System Design (DSD)*, 2009.

[96] H. Moussa, A. Baghdadi, and M. Jézéquel, "Binary de Bruijn on-chip network for a flexible multiprocessor LDPC decoder," in *Proc. of the 45th Design Automation Conference (DAC)*, Jun. 2008, pp. 429–434.

[97] A. Segard, F. Verdier, D. Declercq, and P. Urard, "A DVB-S2 compliant LDPC decoder integrating the Horizontal Shuffle Scheduling," in *Proc. of the International Symposium on Intelligent Signal Processing and Communications (ISPACS'06)*, Dec. 2006, pp. 1013 –1016.

[98] P. Urard, E. Yeo, L. Paumier, P. Georgelin, T. Michel, V. Lebars, E. Lantreibecq, and B. Gupta, "A 135Mb/s DVB-S2 compliant codec based on 64800b LDPC and BCH codes," in *Proc. of the Digest of Technical Papers IEEE International Solid-State Circuits Conference, (ISSCC)*, vol. 1, Feb. 2005, pp. 446 –609.

[99] S. Muller, M. Schreger, M. Kabutz, M. Alles, F. Kienle, and N. Wehn, "A novel LDPC decoder for DVB-S2 IP," in *Proc. of the ACM/IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, 2009, pp. 1308 –1313.

[100] J. Wang, M. Ghosh, and K. Challapali, "Emerging cognitive radio applications: A survey," *IEEE Communications Magazine*, vol. 49, no. 3, pp. 74 –81, Mar. 2011.

[101] S. Haykin, "Cognitive radio: brain-empowered wireless communications," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 2, pp. 201 –220, feb. 2005.

[102] M. A. Bickerstaff, D. Garrett, T. Prokop, C. Thomas, B. Widdup, G. Zhou, and L. M. Davis, "A Unified Turbo/Viterbi Channel Decoder for 3GPP Mobile Wireless in 0.18-mm CMOS," *IEEE Journal of Solid-State Circuits*, pp. 1555–1564, Nov. 2002.

[103] C. Thomas, M. Bickerstaff, L. Davis, T. Prokop, B. Widdup, G. Zhou, D. Garrett, and C. Nicol, "Integrated Circuits for Channel Coding in 3G Cellular Mobile Wireless Systems," *IEEE Communications Magazine*, pp. 150–159, Apr. 2003.

[104] G. Kreiselmaier, T. Vogt, and N. Wehn, "Combined Turbo and Convolutional Decoder Architecture for UMTS Wireless Applications," in *Proc. of the ACM/IEEE Design, Automation and Test in Europe (DATE)*, Feb. 2004, pp. 192–197.

[105] J. R. Cavallaro and M. Vaya, "VITURBO: A Reconfigurable Architecture for Viterbi and Turbo Decoding," in *Proc. of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)* , Apr. 2003, pp. 497–500.

[106] Y. Sun and J. R. Cavallaro, "Unified decoder architecture for LDPC/Turbo codes," in *Proc. of the IEEE Workshop on Signal Processing Systems, (SiPS)*, Oct. 2008, pp. 13–18.

[107] Y. Sun and J. Cavallaro, "A Flexible LDPC/Turbo Decoder Architecture," *Journal of Signal Processing Systems*, pp. 1–16, 2010.

[108] F. Naessens, V. Derudder, H. Cappelle, L. Hollevoet, P. Raghavan, M. Desmet, A. AbdelHamid, I. Vos, L. Folens, S. O'Loughlin, S. Singirikonda, S. Dupont, J.-W. Weijers, A. Dejonghe, and L. Van der Perre, "A 10.37 mm2 675 mW reconfigurable LDPC and Turbo encoder and decoder for 802.11n, 802.16e and 3GPP-LTE," in *Proc. of the IEEE Symposium on VLSI Circuits (VLSIC)*, Jun. 2010, pp. 213 –214.

[109] 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.

[110] M. R. Giuseppe Gentile and L. Fanucci, "A Multi-Standard Flexible Turbo/LDPC Decoder via ASIC Design," in *Proc. of the 6th International Symposium on Turbo Codes and iterative information processing.*, Sept. 2010.

[111] J. Dielissen, N. Engin, S. Sawitzki, and K. van Berkel, "Multistandard FEC Decoders for Wireless Devices," *IEEE Transactions on Circuits and Systems II: Express Breifs*, pp. 284–288, Mar. 2008.

[112] T. Vogt and N. Wehn, "A Reconfigurable Application Specific Instruction Set Processor for Convolutional and Turbo Decoding in a SDR Environment," in *Proc. of the ACM/IEEE Design, Automation and Test in Europe (DATE)*, Mar. 2008.

[113] M. Alles, T. Vogt, and N. Wehn, "FlexiChaP: A reconfigurable ASIP for convolutional, turbo, and LDPC code decoding," in *Proc. of the 5th International Symposium on Turbo Codes and Related Topics*, 2008, pp. 84 –89.

[114] A. Tarable, S. Benedetto, and G. Montorsi, "Mapping interleaver laws to parallel Turbo and LDPC decoders architectures," *IEEE Transactions on Information Theory*, pp. 2002–2009, Sept. 2004.

[115] C. Condo, M. Martina, and G. Masera, "A Network-on-Chip-based turbo/LDPC decoder architecture," in *Proc. of the Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2012, pp. 1525 –1530.

[116] M. Imase and M. Itoh, "A Design for Directed Graphs with Minimum Diameter," *IEEE Transactions on Computers*, vol. C-32, no. 8, pp. 782 –784, Aug. 1983.

[117] F. Kienle, N. Wehn, and H. Meyr, "On Complexity, Energy- and Implementation-Efficiency of Channel Decoders," *IEEE Transactions on Communications*, vol. 59, no. 12, pp. 3301 –3310, Dec. 2011.

[118] A. Hekstra, "An alternative to metric rescaling in Viterbi decoders," *IEEE Transactions on Communications*, vol. 37, no. 11, pp. 1220 –1222, Nov. 1989.

[119] ITRS: International Technology Roadmap for Semiconductors, "System Drivers," 2011 Edition, [online] Available: http://www.itrs.net/Links/2011ITRS/Home2011.htm.

[120] J. Perez and V. Fernandez, "Low-cost encoding of IEEE 802.11n," *Electronics Letters*, vol. 44, no. 4, pp. 307–308, 2008.

[121] E. Boutillon, Y. Tang, C. Marchand, and P. Bomel, "Hardware Discrete Channel Emulator," in *Proc. of the International Conference on High Performance Computing and Simulation*, 2010, pp. 452 –458.

[122] The Dini Group, Inc., " DN9000K10PCI: Xilinx Virtex-5 Based ASIC Prototyping Engine," http://www.dinigroup.com/new/DN9000k10PCI.php.

[123] Y. Sun, Y. Zhu, M. Goel, and J. Cavallaro, "Configurable and scalable high throughput turbo decoder architecture for multiple 4G wireless standards," in *Proc. of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2008, pp. 209 –214.

[124] A. Ahmed, M. Awais, A. Rehman, M. Maurizio, and G. Masera, "A High Throughput Turbo Decoder VLSI Architecture for 3GPP LTE Standard," in *Proc. of the IEEE 14th International Multitopic Conference (INMIC)*, 2011, pp. 340–346.

[125] T.-C. Kuo and A. Willson, "A flexible decoder IC for WiMAX QC-LDPC codes," in *Proc. of the IEEE Custom Integrated Circuits Conference (CICC)* , Sept. 2008, pp. 527 –530.

[126] X. Peng, Z. Chen, X. Zhao, D. Zhou, and S. Goto, "A 115mW 1Gbps QC-LDPC decoder ASIC for WiMAX in 65nm CMOS," in *Proc. of the IEEE Asian Solid State Circuits Conference (A-SSCC)*, Nov. 2011, pp. 317 –320.

[127] M. Awais, A. Singh, E. Boutillon, and G. Masera, "A Novel Architecture for Scalable, High Throughput, Multi-standard LDPC Decoder," in *Proc. of the 14th Euromicro Conference on Digital System Design (DSD)*, Sept. 2011, pp. 340 –347.

List of publications

International Conferences

[1] P. Murugappa, R. Al-Khayat, A. Baghdadi, and M. Jézéquel. A Flexible High Throughput Multi-ASIP Architecture for LDPC and Turbo Decoding. In *Proc. of the ACM/IEEE Design, Automation and Test in Europe Conference & Exhibition (**DATE**)*, Grenoble, France, 13-17 March, 2011.

[2] R. Al-Khayat, P. Murugappa, A. Baghdadi, and M. Jézéquel. Area and Throughput Optimized ASIP for Multi-Standard Turbo Decoding. In *Proc. of the 22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, Karlsruhe, Germany, 24-27 May, 2011.

[3] P. Murugappa, J-N. Bazin, A. Baghdadi, and M. Jézéquel. FPGA Prototyping and Performance Evaluation of Multi-standard Turbo/LDPC Encoding and Decoding". In *Proc. of the 23nd IEEE International Symposium on Rapid System Prototyping (RSP)*, Tampere, Finland, 11-12 Oct, 2012.

[4] P. Murugappa, A. Baghdadi, and M. Jézéquel. Parameterized Area-Efficient Multi-standard Turbo Decoder. *Accepted in the ACM/IEEE Design, Automation and Test in Europe Conference & Exhibition (**DATE**)*, Grenoble, France, 18-22 March, 2013.

National Conferences

[5] P. Murugappa, P. Reddy, R. Al-Khayat, J-N. Bazin, A. Baghdadi, F. Clermidy, and M. Jézéquel. Flexible Multi-ASIP SoC for Turbo/LDPC Decoder. *Colloque National du GDR SoC-SiP : Groupe de Recherche System on Chip - System in Package* Paris, France, 13-15 June, 2012.

Ready for Submission

[6] P. Murugappa, A. Baghdadi, and M. Jézéquel. Reconfigurable Decoder Architecture for QC-LDPC Codes. *Ready for submission to the ACM/IEEE Design Automation Conference (DAC)*, 2013.

[7] P. Murugappa, A. Baghdadi, and M. Jézéquel. Multi-ASIP Platform for Multi-standard Turbo/LDPC Decoding. *Ready for submission to the University Booth of the Design, Automation and Test in Europe Conference & Exhibition (DATE)*, 2013.

[8] P. Murugappa, A. Baghdadi, and M. Jézéquel. ASIP-based Flexible and Optimized Channel Decoders for Turbo and LDPC Codes. *In preparation for submission to IEEE Transactions on Circuits and Systems Part I.*

[9] P. Murugappa, A. Baghdadi, and M. Jézéquel. Unifying Flexibility and Optimization Techniques in the Design of Multi-ASIP LDPC/Turbo Decoders. *In preparation for submission to IEEE Transactions on Communications.*