Toward dynamically reconfigurable high throughput multiprocessor Turbo decoder in a multimode and multi-standard context

Vianney Lapotre

THÈSE / UNIVERSITÉ DE BRETAGNE SUD
sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD
Mention : STIC
École Doctorale SICMA

présentée par
Vianney Lapôtre
Préparée au Laboratoire Lab-STICC, Lorient, France

Toward dynamically reconfigurable high throughput multiprocessor Turbo decoder in a multi-mode and multi-standard context

Thèse soutenue le 20 Novembre 2013
devant le jury composé de :

Christophe JEGO
Professeur, IMS, IPB / *Rapporteur*

Guido MASERA
Professeur, Politecnico di Turino / *Rapporteur*

Jean-Didier LEGAT
Professeur, Université Catholique de Louvain / *Rapporteur*

Édith BEIGNÉ
Docteur, ingénieur de recherche, CEA - LETI / *Examinateur*

Michael HÜBNER
Professeur, Ruhr-Universität Bochum / *Examinateur*

Amer BAGHDADI
Professeur, Telecom Bretagne / *Co-directeur de thèse*

Jean-Philippe DIGUET
Directeur de Recherche, CNRS / Co-*directeur de thèse*

Guy GOGNIAT
Professeur, Université de Bretagne Suf / *Directeur de thèse*

Plus on connaît, plus on aime.
Léonard de Vinci

Remerciements

Je n'imagine pas pouvoir rédiger ces remerciements sans commencer par remercier vivement Guy GOGNIAT qui m'a accompagné et conseillé tout au long de ces trois années de thèse. Nos nombreux échanges allant bien souvent au-delà des questions purement scientifiques entourant ma thèse, m'ont permis de découvrir et d'apprécier aussi bien le Professeur que l'homme.

Je remercie Amer BAGHDADI pour son aide et ses conseils précieux qui m'ont permis d'avancer tout au long de mes travaux de thèse. Je tiens également à le remercier pour l'accueil chaleureux lors de mes différents séjours à Brest. Je remercie également Jean-Philippe DIGUET pour son soutien et nombreux apports lors de la rédaction de ce manuscrit. Je remercie Michael HÜBNER pour son investissement dans mes travaux de thèse ainsi que pour le superbe accueil qui m'a été réservé lors de mon séjour à Bochum.

Je remercie également le Professeur Christophe JEGO d'avoir accepté la charge de rapporteur ainsi que le rôle de président du jury de soutenance de cette thèse. Je remercie également les Professeurs Guido MASERA et Jean-Didier LEGAT d'avoir accepté la charge de rapporteur de cette thèse. Je tiens également à remercier dith BEIGNE d'avoir accepté de participer au jury de la soutenance de thèse.

Ces remerciements sont également le bon moment pour faire une pause, se retourner, regarder ces trois années qui ont défilé si vite, et prendre le temps de vous remercier. Cédric, six années ensemble à l'UBS, emplies de superbes souvenirs, ça ne s'oublie pas. Florence, Virginie, après trois ans de pauses, de missions, et de formulaires administratifs en tout genre, je ne peux que vous dire un grand merci pour votre bonne humeur et votre soutien sans faille. J'ai également à cœur de remercier tous les autres membres du Lab-STICC avec qui nous avons partagé de très bons moments et que je ne cite pas explicitement ici car la liste est bien trop longue. Enfin, j'aimerais remercier les différentes personnes que j'ai rencontrées à Brest et à Bochum pour m'avoir accueillies chaleureusement lors de mes différentes visites.

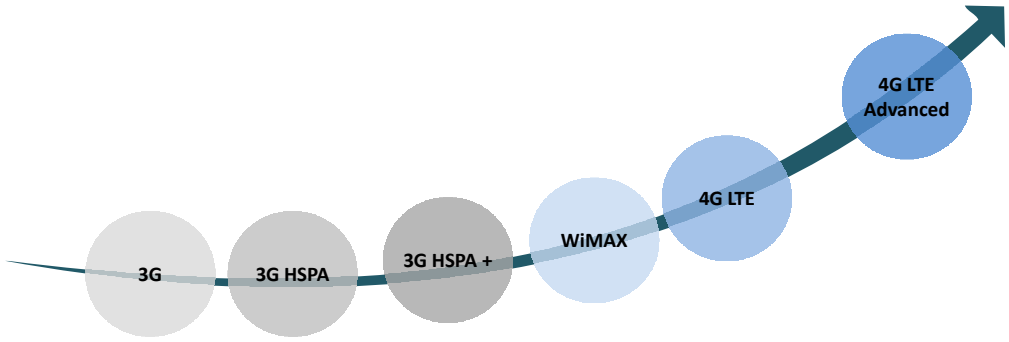
Contents

List of Figures

List of Tables

Introduction

The need of mobile connectivity has hugely increased in the first decade of the 21st century. Homes, schools, businesses and people are now connected together for sharing information as soon as that information is produced. This permanent connectivity has lead to a growing number of connected mobile devices such as laptops, tablets, mobile phones, watches and plenty of other portable devices. This multiplication of connected devices goes along with a large variety of applications and traffic types needing diverse requirements.



Market impact	2003	2006	2009	2010	2016
Peak rate	384 kbps	7 Mbps	42 Mbps	≈150 Mbps	≈ 1 Gbps
Typical user rate downlink	~ 200 kbps	1-2 Mbps	2-10 Mbps	10-20 Mbps	≈ 30-100 Mbps
Typical user rate uplink	64 kbps	64-884 kbps	0.5-4.5 Mbps	5-10 Mbps	≈ 10-60 Mbps

Source: Ericsson – TeliaSonera joint technology study, October 2008

Figure 1: *Communication standards evolution in mobile telephone networks*

Accompanying this mobile connectivity evolution, the last years have seen considerable evolutions of wireless communication standards in the domain of mobile telephone networks, local/wide wireless area networks, and Digital Video Broadcasting (DVB). Figure 1 shows the evolution of standards for mobile since 2003 in terms of throughput requirements. Besides the increasing requirements in terms of throughput and robustness against destructive channel effects, the convergence of services in single smart terminal becomes a crucial and challenging feature. As an example, the fourth generation (4G) of cellular wireless standards aims at providing mobile broadband solution to laptop computer wireless modems, smartphones, and other mobile devices. Diverse features such as ultra-broadband Internet access, IP telephony, gaming services, and streamed multimedia are provided. In order to enable such advanced services at the algorithmic level, new state of the art data processing techniques have been developed and adopted in the emerging wireless communication standards.

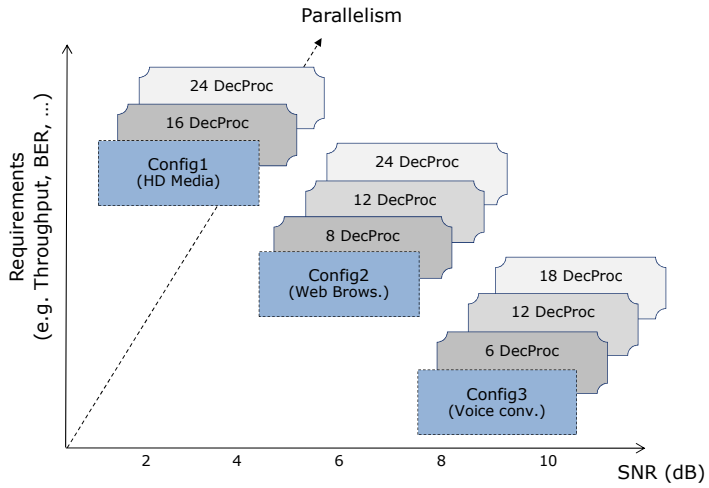


Figure 2: *Usage scenario example of a considered multiprocessor platform*

Channel decoding is a key feature of a wireless communication standard. It allows reliable data transfer targeting high throughput over unreliable communication channels. However, a channel coding technique is typically associated to a variety of parameters and configuration options (frame size, communication channel, signal-to-noise ratio, etc). Among channel decoding techniques, Turbo codes are frequently adopted in the recent wireless standards to reach a very low bit error rate (BER). Furthermore, the high throughput requirement of emerging services imposes the efficient exploitation of different parallelism levels of the underlying algorithms. In this context, multiprocessor architecture is a promising approach to reach high flexibility and high throughput. In fact, flexible multiprocessor architectures are generally designed to support a set of communication standards which correspond to some specific application needs and usage scenarios. Each usage scenario corresponds to particular requirements for example in terms of throughput, latency, error rates, and/or others. Figure 2 gives an example of such usage scenario which corresponds to a mobile terminal supporting different services (High Definition Multimedia, Web Browsing, Voice Conversation) at different channel conditions. At design-time, the multiprocessor architecture must be dimensioned to support the highest requirements while, at run-time, the number of processors can be chosen depending of the current level of requirements. Considering the emerging multi-mode and multi-standard applications, as well as the increasing interest for Software Defined Radio and Cognitive Radio applications, Turbo decoder architectures have to be able to be dynamically adapted to face emerging requirements.

Problematic and contributions

In this context, intensive research has been conducted to provide flexible Turbo decoder targeting high throughput, multi-mode, multi-standard and power consumption efficiency. However, flexible Turbo decoder implementations are not often designed regarding dynamic reconfiguration issues in the context of high throughput, multi-mode and multi-standard scenario requiring high speed configuration switching. In fact, most of the existing related works have proposed flexible hardware platforms while trying to optimize their efficiency in terms of area, throughput, and energy consumption. Very few contributions have considered the crucial requirement of rapid dynamic configuration and the related implementations and costs. Starting from this assessment, this thesis work aims to propose novel contributions in order to reach efficient and high speed configuration of a flexible multiprocessor Turbo decoder. As a base architecture, we consider an ASIP[1]-based flexible Turbo decoder developed at the Electronics Department of Telecom Bretagne in Brest. The considered ASIP, namely DecASIP, supports several wireless communication standards and is integrated in a scalable and flexible multiprocessor platform, namely UDec[2].

Toward the above mentioned objective, the following contributions are detailed in this thesis:

- **Configuration optimization of the flexible DecASIP processor**
  - Proposal of an efficient configuration parameters storage.
  - Optimization of the configuration memory organization in order to provide a low latency configuration information transfer.
  - Proposal of the support of multi-configuration storage and high speed reinitialization of the ASIP.
  - Proposal of a generic program in order to reduce the configuration load.

  These contributions have been presented as a poster at GRETSI national conference and as regular presentations at ISCAS'13 and ISVLSI'13 international conferences.

- **Design of a configuration infrastructure for the UDec multi-ASIP architecture**
  - Optimizations of the platform controller and the interconnection structure of the UDec architecture in order to increase its flexibility.
  - Implementation of a complete configuration infrastructure for high speed

[1]Application Specic Instruction-set Processor
[2]Universal channel Decoder

configuration of the multi-ASIP UDec architecture.

These contributions have been realized during a collaboration with Pr. Michael Hübner in the context of a researcher mobility of six months at the University of Bochum, Germany and have been presented as regular presentations at ReCoSoC'13 and DSD'13 international conferences.

- **Configuration management of the UDec architecture**
  - Definition of a configuration management where configuration information is stored in a global configuration memory.
  - Proposal of two configuration management techniques where configuration information is generated at run-time.
  These last contributions have not been yet published. Several papers are currently under revision and will be submitted soon.

Thesis outline

This thesis manuscript is composed of four chapters as described below:

Chapter 1 firstly introduces the basic concepts related to Turbo codes and Turbo decoding techniques. An overview of the fundamental concepts of channel coding and the basics for error-correcting codes are introduced. Then, the Turbo codes and their basic components are presented. Next, The Maximum Aposteriori Probability algorithm for Turbo decoding and its different levels of parallelism are described. The second section introduces the dynamic configuration problematic for multi-mode and multi-standard Turbo decoders. This is followed by the State of the Art in flexible channel Turbo decoder design. The final part of this chapter presents the initial UDec architecture which constitutes the starting point of this thesis work.

Chapter 2 proposes to tackle the optimization of the initial DecASIP processor for Turbo decoding in order to offer an efficient dynamic configuration of the multi-ASIP UDec architecture for Turbo decoding. An analysis of the configuration lacks of the initial DecASIP architecture is proposed. Based on these observations, optimizations to reach an efficient dynamic reconfiguration are described. These optimizations lead to the implementation of a new processor called RDecASIP. The final part of this chapter provides implementations results in terms of area overhead and configuration load. Finally, the architectural choices are discussed.

Chapter 3 addresses the dynamic configuration of the UDec multi-ASIP architecture for Turbo decoding. The chapter proposes to study the lacks of flexibility of the UDec architecture and points out needs to support the dynamic configuration of the platform. A complete reconfigurable platform implementing eight RDecASIPs is presented and detailed. In order to transfer the configuration to each component of the architecture, the second part of this chapter addresses the definition and the implementation of a dedicated configuration infrastructure providing an efficient and low complexity solution for configuration data transfer to each configuration memory of the implemented RDecASIPs. The proposed configuration infrastructure is evaluated and validated through a SystemC/VHDL mixed simulation model. Finally, implementation results and configuration timing performance are discussed targeting both FPGA and ASIC implementation.

Chapter 4 studies the configuration management of the UDec architecture in order to offer high throughput and high decoding performance. An analysis of the dynamic evolution of the number of decoding iterations regarding the level of sub-block parallelism is provided in order to be integrated in the configuration management of the UDec architecture. Then, this chapter presents two configuration managements. The first one proposes to store the configuration information for all possible configurations in a global memory while the second one proposes the run-time configuration generation respecting the hard constraints in terms of throughput and error rate in a multi-mode and multi-standard scenario.

1

Turbo codes and state of the art in channel decoder design

This first chapter starts with an overview of a typical communication system and an introduction of the main concepts of error correcting codes. As this thesis work targets the dynamic configuration of Turbo decoders, the Turbo coding and the Turbo decoding principles are introduced. Afterwards, the state of the art in flexible channel Turbo decoder design is presented. One of these contributions has been developed at the Electronics department of Telecom Bretagne in Brest using a flexible multi-ASIP approach called UDec. The final part of this chapter presents the initial UDec architecture which constitutes the starting point of this thesis work.

1.1 Context of channel coding

In the context of digital wireless communication systems, information is transmitted over a noisy channel that may cause errors on the received message. Channel coding techniques are used in order to reduce the noise disturbances effects by introducing redundant information to the original message. These coding techniques seek to increase as much as possible the correction capabilities of the communication system to reach the theoretical limits defined by Shannon [1].

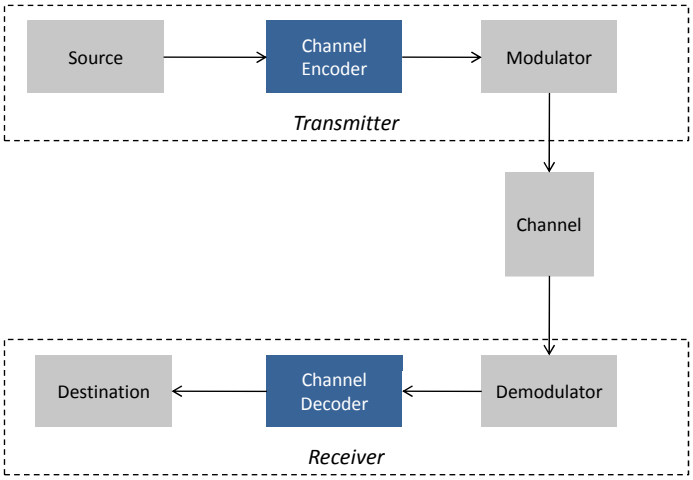


Figure 1.1: *Elements of a digital communication system*

1.1.1 Communication system

A simplified block diagram of a communication system is presented in Figure 1.1. It consists of a *source* that generates a flow of bits representing a particular digital message to be transmitted. This message can be related for example to a video or audio signal, to digital data or be the samples of an analog signal. At the receiver, an estimated message is provided to the *destination*. In an ideal case, the estimated message is identical to the original message generated by the source. Due to noise disturbances introduced by the channel, a *channel encoder* has to be used at the transmitter such as the additional redundant bits can be used by the *channel decoder* for error correction at the receiver. Then, the modulator maps the encoded message into signal waveforms to be transmitted over the channel. Modulation is performed by varying the amplitude, the phase, the frequency or a combination of the three signal parameters of a sinusoidal waveform called a *carrier*. The *channel* reflects the communication medium over which the

message is transmitted (e.g. air, wire-line, optic fiber, etc.). At the receiver, the *demodulator* extracts the information-bearing signal from the modulated carrier. Finally, the *channel decoder* estimates the most probable transmitted message based on the coding rules and the characteristics of the channel.

When a set of data has to be encoded, it is generally admitted that a step of segmentation has to be performed before the encoding process. Indeed, this step prevents excessive complexity and memory requirement of the decoding algorithm at the receiver. The segmentation process consists in dividing the set of data in several parts called *frame* in this document.This process is driven by the application requirements (i.e. throughput, latency, etc...). The maximal size of a frame and the segmentation rules are defined by the targeted standard. For instance, the LTE standard [2] supports frame size from 40 to 6144 bits for Turbo codes. Following the segmentation step, each frame is then encoded and mapped to be transmitted.

1.1.2 Channel code performance

The channel code performance represents the ability of a code to correct transmissions errors. It depends on the Signal-to-noise ratio (SNR) value and the decoding technique used. It is presented in terms of the Frame Error Rate (FER) or Bit Error Rate (BER) values. The benefit that a code associated to a specific decoding technique provides is quantized in terms of the coding gain which is defined as the SNR difference between the coded and uncoded curves for a given error rate value. The coding gain is usually expressed in decibels (dB).

Past years have seen the emergence of two main coding techniques providing excellent error corrections properties: LDPC codes and Turbo codes. The next section introduces the Turbo coding concept which is the channel coding technique focused of this thesis work.

1.1.3 Turbo encoding

The advent of Turbo codes [3] marks a major turning point for digital telecommunication. Indeed, Turbo decoding technique was the first practical solution to closely approach the Shannon's theoretical limits. For the recent and emerging wireless communication standards supporting Turbo codes, a Turbo encoder is usually built from the parallel concatenation of two Recursive Systematic Convolutional (RSC) encoders separated by an interleaver as shown in Figure 1.2. The first RSC encoder receives the input data bits stream D_i in a natural order while the second RSC encoder receives the data in an interleaved one. Three output streams are generated: the systematic S_i, which is identical to the input stream and two parities $P1_i$ and $P2_i$ generated by the encoders in natural and

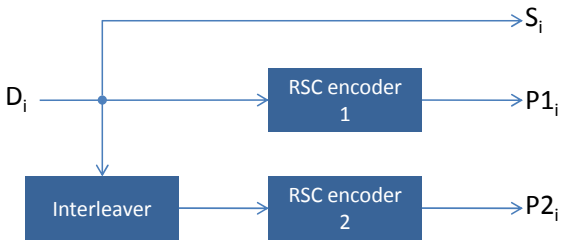
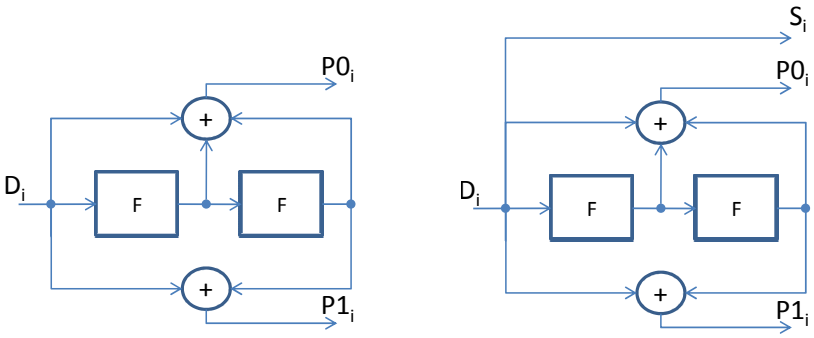


Figure 1.2: *Turbo encoder: Parallel concatenation of two RSC encoders*



(a) Non-systematic encoder of code rate = 1/2

(b) Systematic encoder of code rate = 1/3

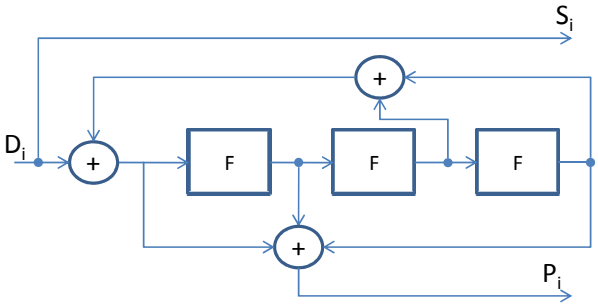
Figure 1.3: *Non-recursive non-systematic and Non-recursive systematic encoders*

interleaved domain respectively. In recent standards, we observe two types of RSC encoders: the Double Binary Turbo Code (DBTC) encoder and the Single Binary Turbo Code (SBTC) encoder. The DBTC encoder generates double binary symbols by encoding bit pairs of the incoming data bits stream while the SBTC encoder encodes bitwise the incoming data bits stream.

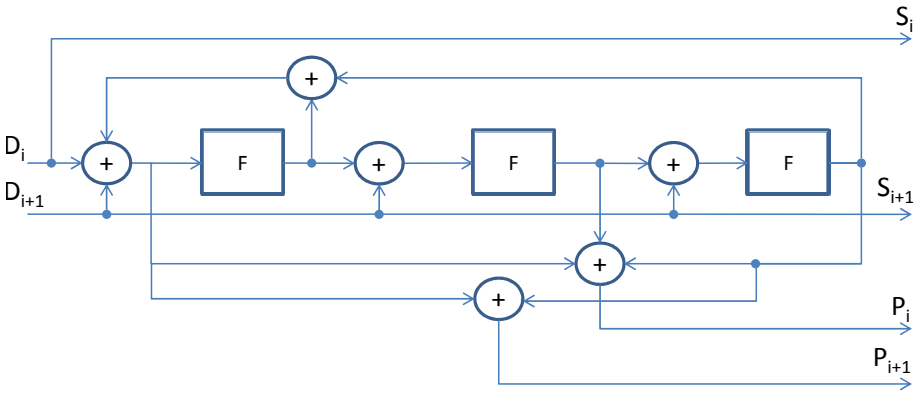
The rest of this section shows the concepts of the RSC encoders and interleaving rules introduced in recent standards such as WiMax, DVB-RCS and LTE.

1.1.3.1 Recursive Systematic Convolutional encoders

Convolutional codes have been widely used in wireless telecommunication standards due to their low complexity. The most common form of convolutional encoder is the non-recursive and non-systematic convolutional encoder presented in Figure 1.3(a). This type of encoder can not be used for Turbo encoding since it is not systematic. A second form of non-recursive encoder (Figure 1.3(b)) introduces a systematic output but it is not suitable for Turbo decoding because of



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



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

Figure 1.4: *Recursive systematic convolutional encoders*

the poor distance properties of the resulting code. Finally, RSC encoders shown in Figure 1.4 introduce the feedback of one of the output. The encoders shown in Figure 1.4(a) presents a single binary RSC encoder. It encodes, at each instant i, one bit of the input data stream. Figure 1.4(b) presents a double binary RSC encoder in which two bits of the input stream are encoded at each instant i.

These encoders have very simple structure that can be implemented with a set of flip-flops and XOR operators. The number of states of the encoder is 2^p when p flip flops are implemented. Moreover, the value $p + 1$ is known as the constraint length of the code. The code rate of a convolutional code is defined by the ratio n/l where n is the number of bits that composes the input symbol D_i and l represents the number of bits of the coded symbol with $l > n$. In the example of Figure 1.3(a), the code rate is 1/2 since at each instant i, the input bit D_i is encoded to a two bits coded symbol that consists of $P0_i$ and $P1_i$. In the example of Figure 1.3(b), the code rate is 1/3 since at each instant i, the input bit D_i is encoded to a three bits coded symbol that consists of $P0_i$ and

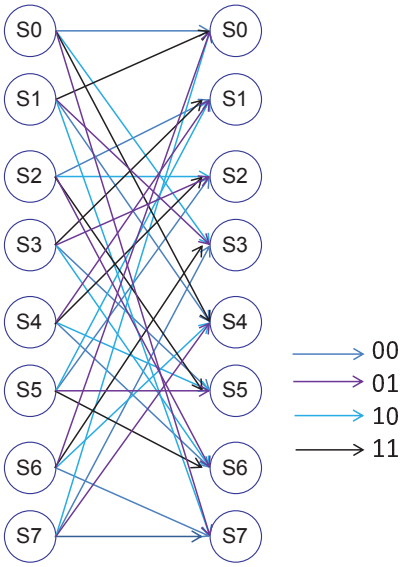


Figure 1.5: *Trellis diagram of the RSC encoder of Figure 1.4(b)*

$P1_i$ and S_i. The code rate of both RCS encoders presented in Figure 1.4 is 1/2. In order to adapt the code rate, the puncturing technique [2, 4] can be used. It consist in removing some of the parity bits after encoding in order to increase the code rate. Another commonly used representation of convolutional encoding is the trellis diagram [5] which consists of nodes and branches. A node represents the state S of the code while a branch represents a transition from one state to another state due to an input bit or bit pair in case of double binary convolutional code. An example of a trellis diagram corresponding to the Double binary RSC encoder presented in Figure 1.4(b) is given in Figure 1.5. In this example, the constraint length of the code is 4, i.e. $p = 3$. Thus, the number of states of the encoder is $2^p = 8$. It can be noted that each state has $2^b = 4$ possible transitions, where $b = 2$ is the number of bits per symbol at the input of the encoder.

For the first frame which has to be encoded, the initial state of the encoder is typically the all-zero state. In order to reinitialize the encoder at the end of the encoding process, tail bits are encoded in order to force the encoder back to the all-zero state and then start to encode the next frame. This technique called *zero padding* in the literature ensures that the encoder starts and finishes in the same state. However, extra parity bits are generated and added to the encoded message that leads to a minor loss of transmission bandwidth. Standards as WiMAX and DVB-RCS adopt the *Tail biting* scheme. It uses a circular RSC encoder, which allows the initialization with a particular state for each frame. This state, called circulation state, ensures that the encoder returns to the same state at the end

of the encoding process. The existence of such a state is guaranteed when the size of the encoded frame is not a multiple of the period of the encoding recursive generator [6]. The value of the circulation state depends on the frame to encode and is determined by a pre-ending step. Following the initialization of the encoder to the all-zero state, the frame is encoded once. From the final state reached at the end of the encoding process, the circulation state is computed using simple combinational operators or a lookup table as described in [6].

1.1.3.2 Turbo codes interleavers

Interleavers provide an efficient solution to enhance the protection of data against destructive channel effects. For that purpose, the data is temporally dispersed. In the context of Turbo codes, the parallel concatenation of two RSC encoders provides two copies of the same symbol at different intervals of time thanks to the interleaver that separates the two encoders. This solution allows retrieving at least one copy of the symbol if the second one has been distorted during the transmission. An interleaver (Π) satisfying this property can be verified by studying the dispersion factor S given by the minimum distance between two symbols i and j in natural order and interleaved order:

$$S = \min_{i,j}(|i-j| + |\Pi(i) - \Pi(j)|) \tag{1.1}$$

The design of interleavers respecting a dispersion factor can be reasonably achieved through the S-random algorithm proposed in [7]. 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 interleaving laws for each supported frame length.

In following paragraphs, the interleaving laws associated to Turbo codes for WiMAX and LTE standards are described.

WiMAX: For this standard, using double binary Turbo code, two levels of interleaving are proposed.

1. The first one is the bit swapping in the alternate couple i.e. $(a_j,b_j) = (b_j,a_j)$ if $j \ mod \ 2 = 0$ where $j = 0, 1, ...N\text{-}1$ and N is the number of couples in the frame.

2. The second one is given by the following expression:

$$\Pi(j) \quad = \quad (P_0 \times j + P + 1)\ mod N \tag{1.2}$$

where

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

where the values of parameters P_0, P_1, P_2 and P_3 depend on the frame size and are defined in the corresponding standard specification [4].

LTE: For this standard, using single binary Turbo code, the interleaver is called quadratic polynomial permutation (QPP). It is given by the following expression:

$$\Pi(j) \quad = \quad (f_1 j + f_2 j)\ mod N \tag{1.3}$$

where f_1 and f_2 are integers parameters defined in the standard [2] and depend of the frame size N.

The previous sections have introduced the basic concepts of Turbo coding. The next section presents the Turbo decoding principle which is the considered application of this thesis work.

1.1.4 Turbo decoding

Turbo decoding principle is based on an exchange of probabilistic information, called *extrinsic information* between two (or more) component decoders dealing with the same received set of data. As shown in Figure 1.6, a typical Turbo decoder consists of two decoders operating iteratively on the received frame. The first component (SISO decoder 0 in Figure 1.6) works in natural domain while the second (SISO decoder 1 in Figure 1.6) works in interleaved domain. The Soft-Input Soft-Output (SISO) decoders operate on soft information to improve the decoding performance. Thus, besides its own channel input data, each SISO decoder deals with the extrinsic information generated by the other SISO decoder in order to improve its estimation over the iterations. Usually, but not necessary, the computations are done in the logarithmic domain. Each decoder calculates the Log-Likelihood Ratio (LLR) for the i^{th} data bit d_i as

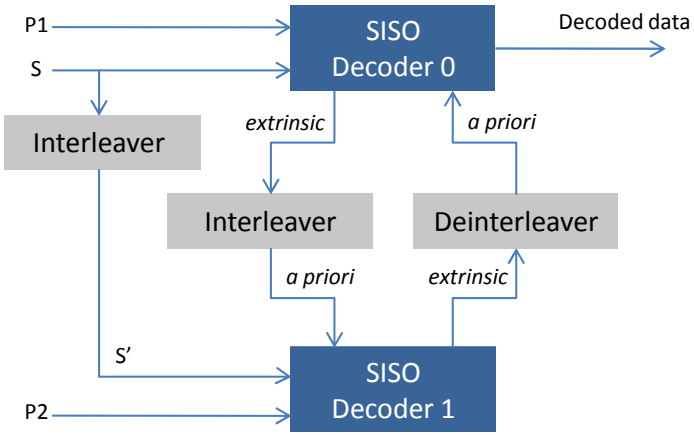


Figure 1.6: *Typical Turbo decoder structure*

$$L(d_i) = ln\frac{Pr(d_i = 1|y)}{Pr(d_i = 0|y)} \tag{1.4}$$

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

$$L(d_i) = L^{ap}(d_i) + L^{sys}(d_i) + L^{par}(d_i) \tag{1.5}$$

where $L^{ap}(d_i)$ is the a-priori information of d_i, $L^{sys}(d_i)$ and $L^{par}(d_i)$ are the channel measurement of the systematic and parity parts respectively. Each SISO decoder generates extrinsic information that is sent to the other decoder. Extrinsic information becomes the a-priori information $L^{ap}(d_i)$ for the other decoder as shown in Figure 1.6.

Several algorithms for this SISO decoding have been proposed in the literature. The Soft Output Viterbi Algorithm (SOVA) and the Maximum Aposteriori Probability (MAP) algorithms are the most frequently used. The SOVA algorithm [8] is a soft output variant of the Viterbi algorithm targeting the minimization of the FER while the MAP algorithm [9] targets to minimize the BER. This last algorithm has been simplified in [10] to propose the Max-Log-MAP algorithm that is most often adopted because of the efficient hardware implementation possibility. For a better understanding of the architectural and configuration issues highlighted in the rest of this thesis work, the next section provides a short introduction to the MAP decoding.

1.1.4.1 The MAP Algorithm

A MAP decoder provides, for each coded symbol d_i^{symb} consisting in m bits of a frame of N coded symbols, 2^m *a posteriori* probabilities given the channel output y received by the decoder. The hard decision on the corresponding value j, i.e. $d_i^{symb} = j$, that maximizes the *a posteriori* probability is expressed in terms of joint probabilities as:

$$Pr(d_i^{symb} = j|y) = \frac{P(d_i^{symb} = j, y)}{\sum_{k=0}^{2^m-1} P(d_i^{symb} = k, y)} \quad (1.6)$$

The trellis structure of the code enables us to decompose the calculation of joint probabilities between past and future observations. This decomposition defined by Equation (1.7) uses a *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. It also uses a *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 and a *Branch metric* $\gamma(S', S)$, which gives the state transition probability from state S' to state S of the trellis at instant i.

$$Pr(d_i^{symb} = j|y) = \sum_{(S',S)/d_i^{symb}=j} \alpha_i(S')\gamma_i(S', S)\beta_{i+1}(S) \quad (1.7)$$

The Forward recursion metric and Backward recursion metric are expressed by Equation (1.8) and Equation (1.9) respectively.

$$\alpha_{i+1}(S) = \sum_{S'=0}^{2^m-1} \alpha_i(S')\gamma_i(S', S), i \in 0...N-1 \quad (1.8)$$

$$\beta_i(S) = \sum_{S'=0}^{2^m-1} \beta_{i+1}(S')\gamma_i(S', S), i \in N-1...0 \quad (1.9)$$

The initialization of these metrics depends of the initial and final state of the trellis. The Branch metric is given by Equation (1.10).

$$\gamma_i(S', S) = p(y_i|x_i).Pr^a(d_i^{symb} = d_i^{symb}(S', S)) \quad (1.10)$$

where $p(y_i|x_i)$ is the channel transition probability. x_i and y_i are the i^{th} transmitted modulated symbol and received symbol respectively. Assuming an equiprobable source, the *apriori* probability $Pr^a(d_i^{symb} = d_i^{symb}(S', S)) = \frac{1}{2^m}$. The

generated extrinsic information corresponds to the aposteriori probability (Equation (1.6)) in which the branch metric is modified in order to remove the symbol channel input since the other decoder knows this information.

The next section introduces the different levels of parallelism that can be exploited considering a MAP-SISO decoder. It particularly highlights the SISO decoder level parallelism.

1.1.4.2 Parallelism in Turbo decoding

Turbo decoding provides an efficient solution to reach very low error rate performance at the cost of high processing time for data retrieval. Researches targeting the exploitation of parallelism have been conduced in order to achieve high throughput. These parallelism levels that can be categorized in three groups: Metric level, SISO decoder level, Turbo decoder level.

The Metric level parallelism concerns the processing of all metrics involved in the decoding of each received symbol inside a MAP-SISO decoder. For that purpose, the inherent parallelism of the trellis structure [11, 12] and the parallelism of the MAP computation can be exploited [11, 12, 13]. The MAP-SISO decoder level parallelism consists in duplication of the SISO decoders in natural and interleaved domain, each executing the MAP algorithm on a sub-block of the frame to decode. Finally, the Turbo decoder level parallelism proposes to duplicate whole Turbo decoders to process iterations and/or frames in parallel. However, this level of parallelism is not relevant due to the huge area overhead of such an approach (all memories and computation resources are duplicated). Moreover, this solution presents no gain in frame decoding latency.

The SISO decoder level parallelism hugely impacts the configuration process of a multiprocessor Turbo decoder. Indeed, the number of SISO-decoders that have to be configured and the configuration parameters associated with each SISO-decoder are both dependent of this parallelism level. At this level, three techniques are available: Frame sub-blocking, Windowing, and Shuffled decoding.

Frame sub-blocking: In sub-block parallelism, each frame is divided into M sub-blocks and then each sub-block is processed on a MAP-SISO decoder (Figure 1.7) using adequate initializations. Besides duplication of MAP-SISO decoders, this parallelism imposes two other constraints. On the one hand, interleaving has to be parallelized in order to scale proportionally the communication bandwidth. Due to the scramble property of interleaving, this parallelism can induce communication conflicts except for interleavers of emerging standards that are conflict-free for certain parallelism degrees. In case of conflicts an appropriate

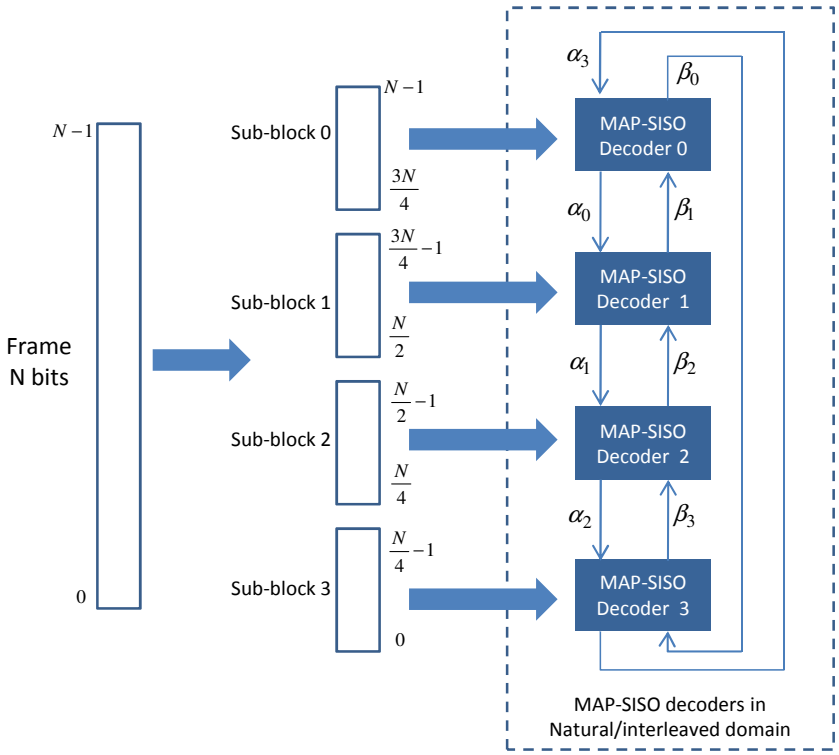


Figure 1.7: *Sub-block parallelism with message passing for metric initialization*

communication structure, e.g. Network on Chip (NoC), should be implemented for conflict management [14]. On the other hand, MAP-SISO decoders have to be initialized adequately either by acquisition or by message passing. In [15] a detailed analysis of the parallelism efficiency of these two methods is presented which gives favor to the use of message passing technique. The message passing, which initializes a sub-block with recursion metrics (α and β) computed during the previous iteration in the neighboring sub-blocks (Figure 1.7), needs not to store the recursion metric and time overhead is negligible compared to the acquisition method.

Sliding window: In addition to the sub-block parallelism, the sliding window technique is commonly used at the SISO decoder level. It allows to reduce the memory size required to store the state metric values by splitting each sub-block into a number of small windows, where the MAP decoding is applied to each window independently [16, 17]. To illustrate this technique, let us assume that a backward-forward schedule is adopted for the state metric recursions of the MAP-SISO decoder (Figure 1.8). In this schedule, β recursion metrics are computed

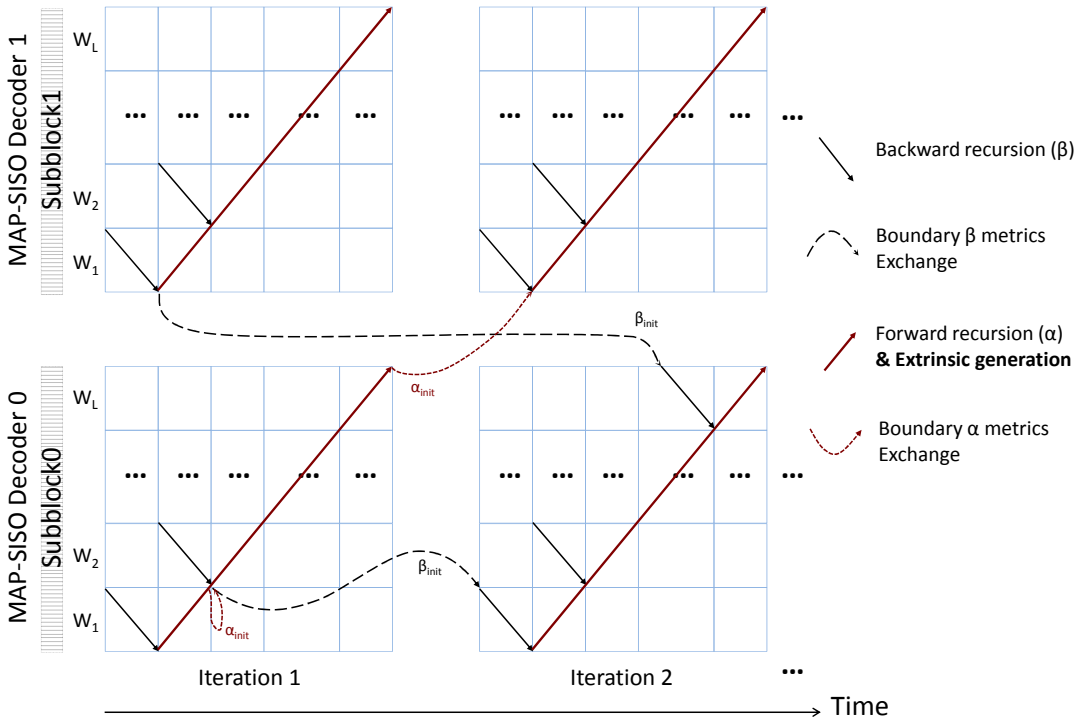


Figure 1.8: *Sub-blocking and windowing with message passing for metric initialization*

before α recursion metrics. Thus, using only sub-block parallelism would require storing the intermediate recursion metrics calculated in the backward recursion in an internal memory, called *cross metric memory*. Therefore, this 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 [16, 17]. This implies cross metric memory depth reduction to the size of a window (with an additional requirement of storage state metric boundaries values of all windows to be used in the next iteration). Consequently, each MAP-SISO decoder processes the sub-block, window by window as shown in Figure 1.8. In the example of Figure 1.8, the level of sub-block parallelism is 2. Each MAP-SISO decoder uses two recursion units and employs backward-forward schedule for window processing. The first recursion unit (processing in the backward direction of the trellis) executes on window j while the second recursion unit (processing in the forward direction of the trellis) executes on window $j-1$ at the same time (as shown in Figure 1.8). Boundary state metrics are then exchanged between windows either immediately within the same iterations or stored and used in the subsequent iteration as illustrated in Figure 1.8.

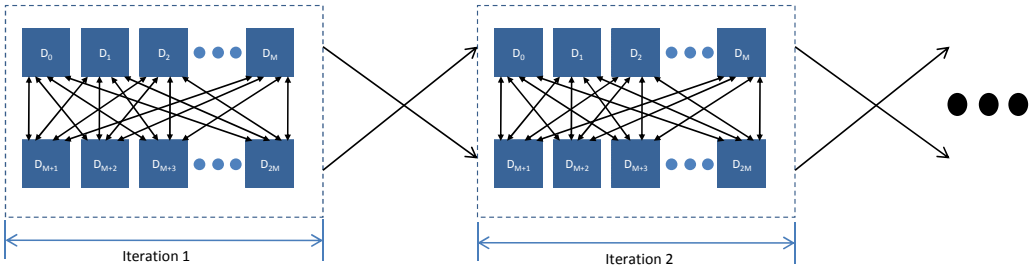


Figure 1.9: *Shuffled decoding scheme, where D_x = MAP-SISO decoder x=1,2,...*

Shuffled Turbo decoding: The principle of the shuffled decoding technique has been introduced in [18]. In this mode, all component decoders work in parallel and exchange extrinsic information as soon as it is created. Thus the shuffled decoding technique performs decoding (computation time) and interleaving (communication time) fully concurrently while serial decoding implies waiting for the update of all extrinsic information before starting the next half iteration (Figure 1.9). 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, to preserve error-rate performance with shuffled Turbo decoding, an overhead of iteration between 5 and 50 percent is required depending on the MAP computation scheme, on the degree of sub-block parallelism, on propagation time, and on interleaving rules [15].

The previous sections provided the basic background on Turbo codes and on the different levels of parallelism which can be exploited in order to reach high throughput requirement imposes by emerging communication standards. The next section introduces configuration methods in embedded system and tackles the specific dynamic configuration scenario of a Turbo decoder in a multi-mode, multi-standard and mobility context.

1.2 Dynamic configuration of flexible Turbo Decoders

1.2.1 Dynamic configuration in embedded systems

In the context of telecommunication, the multiplication of wireless standards is introducing the need of flexible and dynamically reconfigurable multi-mode and multi-standard baseband receivers. The methods to reconfigure an architecture are multiple and can be organized in three main categories.

The first one corresponds to architectures that are configured through a stream of configuration bits that are spread over the architecture components to configure the data and control path. For instance, the configured components can be multiplexers, Lookup Tables (LUTs), Arithmetic Logic Unit (ALU), etc. This configuration method is typically applied to Field-Programmable Gate Array (FPGA) in which LUTs, multiplexers and programmable routing switches are configured through a bitstream load at power-up. Recent FPGA technology also proposes dynamic configuration techniques allowing hardware reconfiguration at run-time. The configuration load of recent FPGAs represents several Mega Bytes of information that can be loaded from various sources as an external memory, a host PC, a microcontroller, etc. Figure 1.10 shows bitstream chain in a FPGA which is sent from outside and is then spread inside the component. The configuration of the SRAM points of the FPGA can be seen as a huge shift register (in practice, the configuration chain is divided into frames and latches are used).

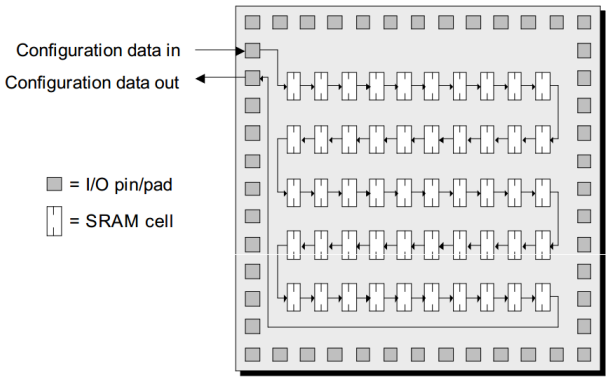
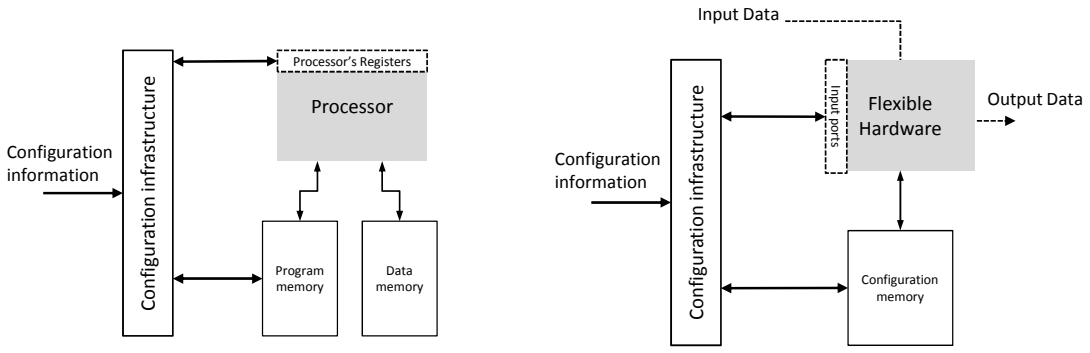


Figure 1.10: *FPGA reconfiguration chain*

The second category corresponds to instruction-set based processors that provide inherently high flexibility in terms of control logic design. The reconfiguration of these architectures is done by context switching. A context switch is the switching of a processor from one task to another. For that purpose, the program instructions/counter and processor's registers have to be initialized for the new task. As shown in Figure 1.11(a), depending on the system architecture, a configuration infrastructure could be needed to load new instructions in the program memory of the processor and modify processor's registers values before the execution of a new task.

The third category corresponds to traditional parametrized hardware architectures. The flexibility of these designs is incorporated by the designer through the use of initialization parameters loaded from a configuration memory or input ports of the architecture. Figure 1.11(b) shows a parametrized hardware



(a) Instruction-set based processor (b) Flexible Hardware

Figure 1.11: *Reconfiguration of Instruction-set based processor and flexible hardware*

Category	Configuration data	Configuration load	Configuration granularity
Stream of configuration bits	bitstream	From $\approx$ 10 to $\approx$ 100 of Mbytes	Low (LUTs, switches)
Instruction-set based processor	Registers and Program	from $\approx$ 10 to $\approx$ 1000 Mbytes	High (Software)
Parametrized hardware	Ports and config. memory	≤ 1 Kbytes	Medium (Control/data paths, operands)

Table 1.1: *Dynamic configuration methods*

design which is configured through a configuration memory and configuration information from input ports. The configuration infrastructure associated to the architecture allows the configuration memory loading with configuration parameters and drives the configuration input ports. This solution provides the lowest configuration load since the number and size of each configuration parameter are optimized at design time.

Table 1.1 summaries the main features of the three configuration methods described in previous paragraphs. It can be notice than the ASIP technology can be classed in both second and third categories. Indeed, an ASIP is an instruction-set based processor that can incorporate configuration memories and configuration pins if it is necessary.

The next section highlights the configuration issue of a Turbo decoder in a multi-mode and multi-standard context.

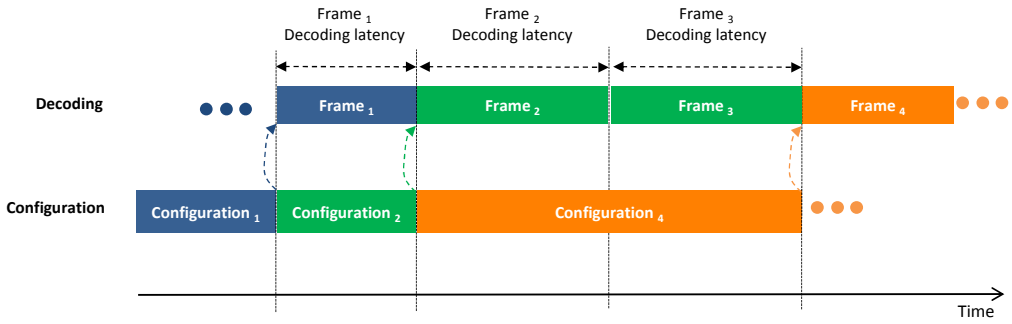


Figure 1.12: *Worst case configuration scenario*

1.2.2 Dynamic configuration in multi-mode and multi-standard scenario

When a Turbo decoder is designed to support several communication standards, the decoder behavior has to be adapted in order to respect the application requirements and to take into account the communication channel quality. In this thesis work, the scenario presented in Figure 1.12 is considered as the worst case configuration scenario that should be met by a multi-mode and multi-standard Turbo decoder in mobility.

In this scenario, the Turbo decoder deals with input frames that have to be decoded for multiple applications that use different communication standards or modes. Each application is associated with throughput and BER objectives. Moreover, considering a mobile terminal, the configuration associated to an application has to be adapted temporally depending on the communication channel quality evolution. Consequently, as shown in Figure 1.12, each frame received by the Turbo decoder is associated to a specific configuration which takes into account the application requirements and the channel quality. In order to avoid extra delays between two frames associated with different configurations, the configuration process for a frame (i.e. computing and loading the new configuration) can be performed during the processing on the current frame. Thus, the Maximum Configuration Latency (MCL) for a frame k ensuring a null extra delay between two frames is evaluated using Equation (1.11).

$$MCL(k) \quad = \quad N_{PrevFrame}(k).\frac{FrameSize(k-1)}{Throughput(k-1)} \tag{1.11}$$

where k is the k^{th} received frame, $N_{PrevFrame}$ is the number of consecutive frames decoded with the same configuration that precede the frame k, $FrameSize(k-1)$ is the $k-1^{th}$ frame size in bits and $Throughput(k-1)$ is the throughput requirement associated with the $k-1^{th}$ data frame. In the example presented in Figure 1.12, $N_{PrevFrame}(2) = 1$ since one frame is decoded with the $Configuration_1$

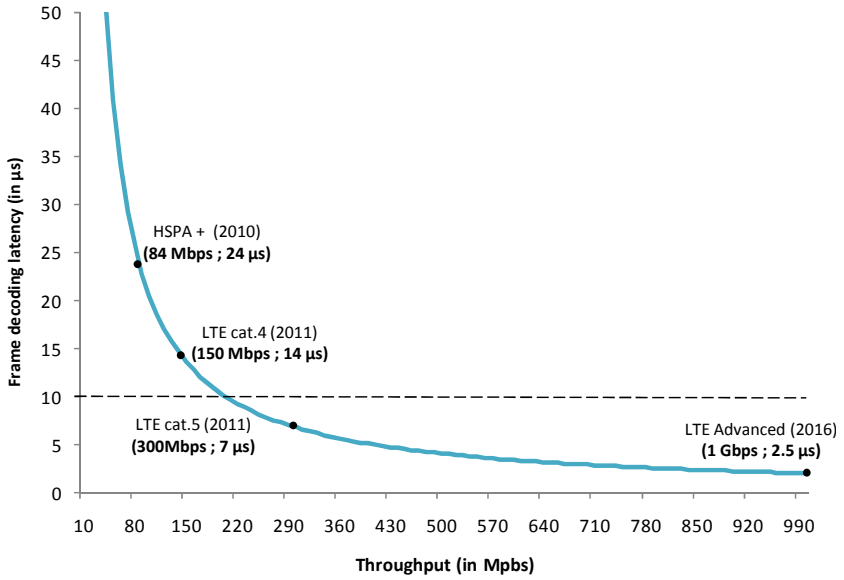


Figure 1.13: *Decoding latency of a 2048 bits frame*

while $N_{PrevFrame}(3) = 2$ since two frames are decoded with the $Configuration_2$. *MCL*, *FrameSize* and *Throughput* are expressed in seconds, bits and bits/s respectively. Assuming the worst case when $N_{PrevFrame}(k) = 1$, the maximum configuration latency critically decreases with high throughput targeted by emerging and future wireless communication standards as shown in Figure 1.13. This figure presents the decoding latency of a 2048-bit data frame for different current and emerging wireless communication standards. Regarding the throughput requirement evolution, the decoding latency of a frame decreases and will reach latencies around few microseconds in LTE-advanced standard. Thus, considering the dynamic configuration scenario presented in this section, emerging and future high throughput multi-mode and multi-standard architectures would have to deal with configuration latencies lower that 10 μs. That is why, in order to face this challenge, this thesis work aims to bring contributions providing an efficient and high speed dynamic configuration of a multi-mode and multi-standard Turbo decoder.

1.3 State of the art in flexible Turbo decoding architectures

Since the invention of Turbo codes in 1993 [3], a considerable amount of contributions targeting the VLSI implementation of Turbo decoders have been proposed.

These implementations target diverse design objectives in terms of area efficiency, energy efficiency, scalability, flexibility and high throughput. Among the initial efforts in this context, the work described in [19] has investigated sub-block parallelism in order to increase the throughput. In [13], the authors have explored different computational schemes for the MAP algorithm. Quantization optimizations for input and extrinsic information have been studied in [20]. Algorithmic and architecture joint optimizations have been explored and proposed in [21]. [22] presents one of the first ASIC implementation achieving a throughput of 50 Mbps with 10 decoding iterations and an operating frequency of 1 GHz.

Turbo codes have been widely adopted in wireless communication standards like CDMA2000, UMTS, LTE, WiMAX, DVB-RCS, etc. Several architecture approaches have been investigated in order to build high throughput and flexible Turbo decoders. Many implementations have succeeded to meet the low throughput requirements of the early standards (e.g. CDMA2000 and UMTS) using advanced DSP architectures [23, 24, 25], customizable processors [26]. However, the scalability of such implementations is limited by the block interleavers specified in these standards which cause memory access contentions when targeting higher sub-block parallelism degree. In [27], the authors present an implementation of a turbo decoder in the Coarse-Grained Montium Architecture. The implementation offers a low reconfiguration latency of 6.36 μs but the architecture reaching 100 MHz supports the low UMTS throughput requirement (1.92 Mbps) only. FPGA implementations have been investigated but this technology suffers from a prohibitive reconfiguration latency. In [28], results show that the reconfiguration process of the FPGA is 35 ms.

The introduction of contention-free interleavers, like ARP in WiMAX and QPP in LTE, alleviated this limitation enabling high throughput implementations [29, 30, 31, 32, 33, 34]. The work presented in [29] targeting LTE, allows multiple SISO decoders (1, 2, 4, or 8) to concurrently process frame subblocks 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 mm2 while exhibiting a power consumption of 219 mW and supporting the maximum specified frame size of 6144 bits. Another example of flexible architecture is the parameterized architecture of [30] which supports both Turbo modes (DBTC and SBTC) and achieves a high throughput of 187 Mbps with 8 parallel MAP decoders. Targeting Gbps throughputs, a recent work [35] has proposed an LTE-Advanced 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 mm2 using 65nm CMOS technology. The architectures previously presented propose to use multiple SISO decoders to reach high throughput of emerging and future

standards. However, even if these turbo decoders offer certain degrees of flexibility to adapt for instance the number of SISO decoders, the turbo code mode (i.e. SBTC or DBTC), or the frame size, the authors do not present any configuration infrastructures associated to their architecture in order to support dynamic configuration switches.

In order to support dynamic configuration, the authors of [36] present an FPGA implementation of a High Speed MAP Decoder Architecture for Turbo Decoding achieving 346 Mbps. However, the configuration latency cost of such an implementation is not evaluated. The recent FPGA dynamic reconfiguration mechanisms provide very high flexibility. The configuration latency of Xilinx FPGA [37] depends on the targeted FPGA technology, the bitstream size and the medium (ICAP, JTAG, etc.) used to transfer the configuration bitstream. However, the configuration latency overhead is still important (from around 100 μs to 100 ms). Recent works investigated General Purpose Processor (GPP) implementations using high performance multi-cores architectures taking advantage of the Intel SSE (Streaming SIMD Extensions) instructions. In [38], a 418 Mbps turbo decoder for LTE is implemented on an Intel Xeon processor X5670 with a 12 threads level of parallelism. The 150 Mbps LTE throughput requirement is reached with a 4 threads implementation. In [39], an adaptive turbo decoder implementation on an Intel I7-960 core is investigated. The authors propose to adapt the decoding algorithm depending on the communication channel quality. However, for both [39] and [38] works, no discussion is provided about the context switching cost when the turbo decoder configuration has to be changed. Moreover, these GPP implementations have been initially developed for base station. Thus, they are not suitable for mobile terminals due to the high power consumption of such processors.

Recently, Application Specific Instruction-set Processor (ASIP) designs solutions have been investigated in order to offer architectures providing good compromises in terms of flexibility, throughput and power dissipation. In [40] 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 is iteratively and concurrently exchanged between multiple component decoders via an on-chip communication network presented in [41]. Afterwards, optimizations on the proposed ASIP have been added in [42]. Moreover, LDPC decoding has been integrated. Efficient resource sharing between LDPC and Turbo decoding modes is proposed as well as new LDPC decoding schedule adapted to a multi-ASIP architecture. In [43], the authors introduce the FlexiTreP ASIP presented in [44] in a multi-ASIP architecture for turbo decoding consisting of two ASIPs and dedicated accelerator to reach the 150 Mbps throughput requirement of LTE. This ASIP, whose general architecture is illus-

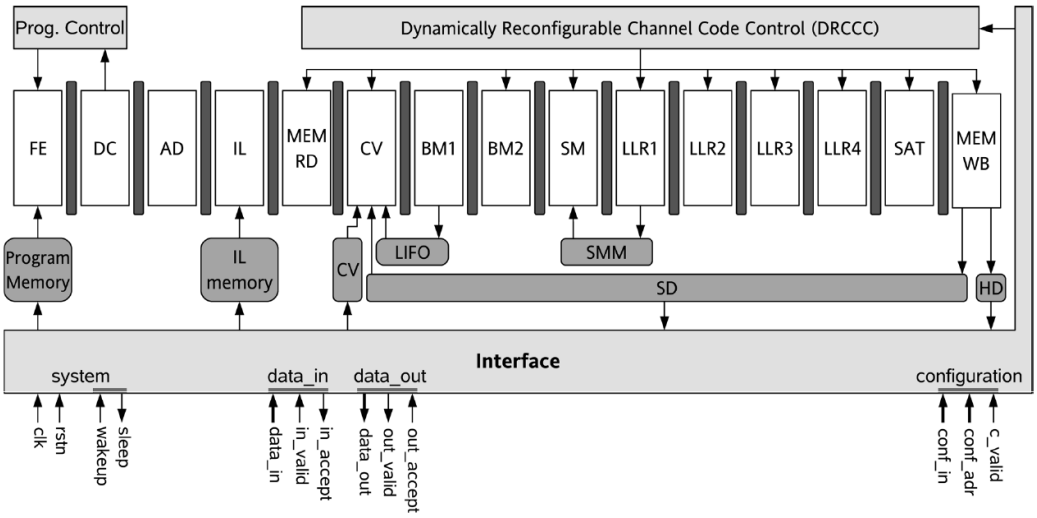


Figure 1.14: *FlexiTreP general architecture*

trated in Figure 1.14, supports both SBTC and DBTC for various standards and it is configured through an interleaver memory (IL), a program memory and the Dynamically Reconfigurable Channel Code Control (DRCCC). The DRCCC is a look-up table based unit which allows the configuration of the structure of the convolutional code, the internal data-path, and the configuration memory. Two configurations are stored in this unit, a working and a shadow configuration. The working configuration holds the parameters that are actually used while the shadow configuration is used to prepare the next configuration. One cycle switching can be performed between these two configurations thanks to a special instruction. However, using a specific instruction in the program to switch between two configurations limits the flexibility because the reconfiguration scenario is defined statically. Moreover, even if the DRCCC allows to prepare the next configuration parameters in parallel of the current decoding process, the program and interleaver memories can not be modified during the processing. A configuration delay, which is not detailed in the presented works, between two consecutive frames is necessary to load these two memories for each ASIP implemented on a platform.

Previous works provide an efficient way to reach the high performance requirement of emerging standards. However, the dynamic reconfiguration aspect of these platforms is superficially addressed. All these platforms can be reconfigured through program and configuration memories of each core, but the configuration mechanisms are not optimized for an efficient implementation in a multi-core system. Furthermore, these platforms are not coupled with a configuration infrastructure that allows configuration broadcasting to the cores. As explained in

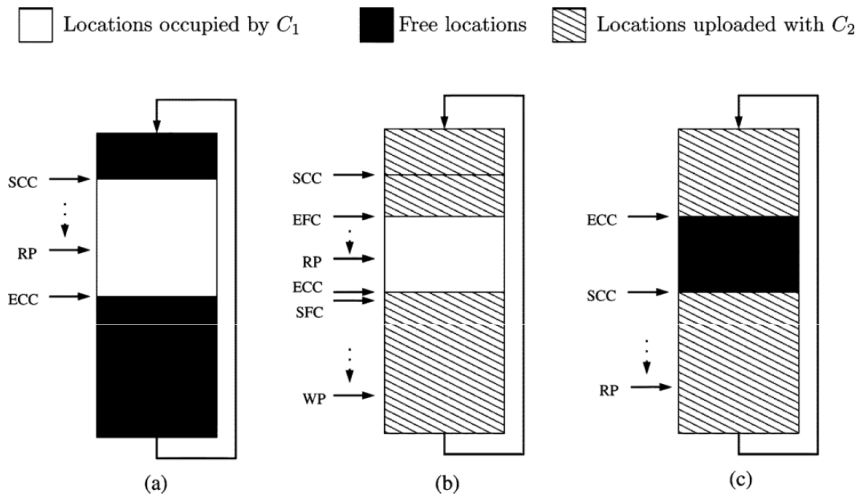


Figure 1.15: *Memory reconfiguration process presented in [44]*

Section 1.2.2, the perpetual increase of throughput of wireless standards reduces the reconfiguration time available between two data frames while the number of cores increases to reach high throughput. This point is particularly challenging as in many standards decoding parameters can be changed as early as one data frame ahead [45]. Among the few works which considered this issue, we can cite the recent architecture presented in [46] where the authors propose solutions for the reconfiguration management of the NoC-based multiprocessor Turbo/LDPC decoder architecture presented in [47]. Up to 35 processing elements (PEs) and up to 8 configuration buses have been implemented. Each PE is configured through a configuration memory which is organized as a circular buffer. The set of pointers used to manage reading and writing operations are shown in Figure 1.15. The start of current configuration (SCC) pointer and the end of current configuration (ECC) pointer delimit the memory blocks that are currently being used. A read pointer (RP) is used to retrieve the data during the decoding process, as shown in Figure 1.15(a). The start of future configuration (SFC) and end of future configuration (EFC) pointers, shown in Figure 1.15(b) are instead used concurrently with the write pointer (WP) to delimit the locations that are going to be used to store the new configuration data. The reconfiguration process to switch from a configuration C_1 to a configuration C_2 can be masked by the current decoding task (Figure 1.15(c)) if the configuration memory provides enough free space and if a high speed configuration infrastructure is provided.

For that purpose, groups of four PEs are connected to a dedicated configuration bus. The remaining PEs are shared among the buses when the number of PEs is not divisible by four. Dynamic reconfiguration during one frame duration is possible when the current configuration is small enough to load a new config-

uration in the memory. If not, authors provide management solutions to deal with this issue, such as erasing the current configuration during the last decoding iteration and continue the reconfiguration process during the first iteration of the new configuration, but it is not always sufficient. Then, stopping the current processing to configure the new configuration is unavoidable and leads to a decoding quality loss in terms of BER. Moreover, the cost of the proposed multi-bus configuration infrastructure becomes too high with the increasing number of PEs and leads to a complex configuration transfer management.

To leverage these issues, it becomes essential to propose original solutions for a low complexity and stopping-free configuration of multiprocessor Turbo decoders. Previous works targeting reconfigurable Turbo decoder architectures tackled one or two specific aspects of the reconfiguration process leading to inefficient global reconfiguration mechanisms. Indeed, as summarized in Table 1.2, [44, 43] proposes an efficient dynamically reconfigurable channel code control (DRCCC) unit allowing a one cycle reconfiguration of the processor's pipeline but the rest of the configuration information (program and interleaver configuration) is not optimized for a high speed configuration. In [46], the configuration memory of each processing element has been designed to support configuration overlapping but the configuration load is not optimized so a complex multi-bus configuration has to be implemented to support high speed configuration. In this context, this thesis work, starting from the works presented in [40, 41, 42], aims to further investigate and optimize the dynamic reconfiguration process in order to support stopping-free dynamic reconfiguration for high throughput requirements and high level of parallelism in the context of multi-mode and multi-standard scenarios presented in Section 1.2.2. Configuration latency below 10 μs, including run-time configuration generation which is not implemented in previous work, must be proposed to reach this objective for emerging and future communication standards.

Regarding a complete communication chain, reaching a low configuration latency is even more important because the Turbo decoder is not the only component to be reconfigured. Indeed, a turbo decoder is usually integrated as an accelerator into an heterogeneous platform such as the MAGALI [48], the Sandbrige [49], the Tomahawk [50], the IMEC SDR platform architectures [51] among many other platforms [52, 53, 54]. Consequently, the configuration process performed to reconfigure one component can have an impact on the entire receiver if the required configuration latency introduces an extra delay leading to stop the processing along the communication chain. The integration of a Turbo decoder in a complete platform for telecommunication is out of the scope of this thesis work. However, this thesis work aims to explore and propose a high speed reconfigurable Turbo decoder to be implemented in such a platform in future works.

	[44, 43]	[46]	[40, 41, 42]
Config. support	Program and Interleaver memories, DRCC	Configuration memory	Program and configuration memories
Config. infrastructure	Not detailed	multi-bus (up to 8 buses for 35 processors)	NO
Config. management	Not dynamic (decoding is stopped for configuration)	YES (the current configuration is erased during the processing or decoding is stopped)	NO
Stooping-free	NO	Not guaranteed	NO
Run-time config. generation	NO	NO	NO

Table 1.2: *Configuration overview of the most relevant SoA works*

1.4 Initial multi-ASIP architecture for turbo decoding

The design of an ASIP-based multi-mode and multi-standard architecture for turbo decoding has been initiated at the Electronic Department of Telecom Bretagne in Brest in previous thesis works [55, 42]. In [55], the main objective was to evaluate the abilities of newly ASIP-design tools in terms of quality of the generated HDL code and flexibility limitation when targeting turbo decoding application. The thesis work presented in [42] proposes to improve the initial ASIP architecture mainly in terms of area and throughput and to introduce the support of LDPC decoding. In this section, an overview of the multi-mode and multi-standard turbo decoder which is the starting point of this thesis work is given.

The *UDec* turbo decoder architecture is shown in Figure 1.16. It consists of two rows of DecASIPs interconnected via a Butterfly Network on Chip. Each row corresponds to a component decoder. In the example of Figure 1.16, four ASIPs are organized in 2 component decoders respectively built with 2 ASIPs. Each DecASIP is associated with 3 memory banks of size 24x256 used to store the input channel LLR values ①. There are also another 3 banks of size 30x256 used for extrinsic information storing ②. Each ASIP is further equipped with two 88x32

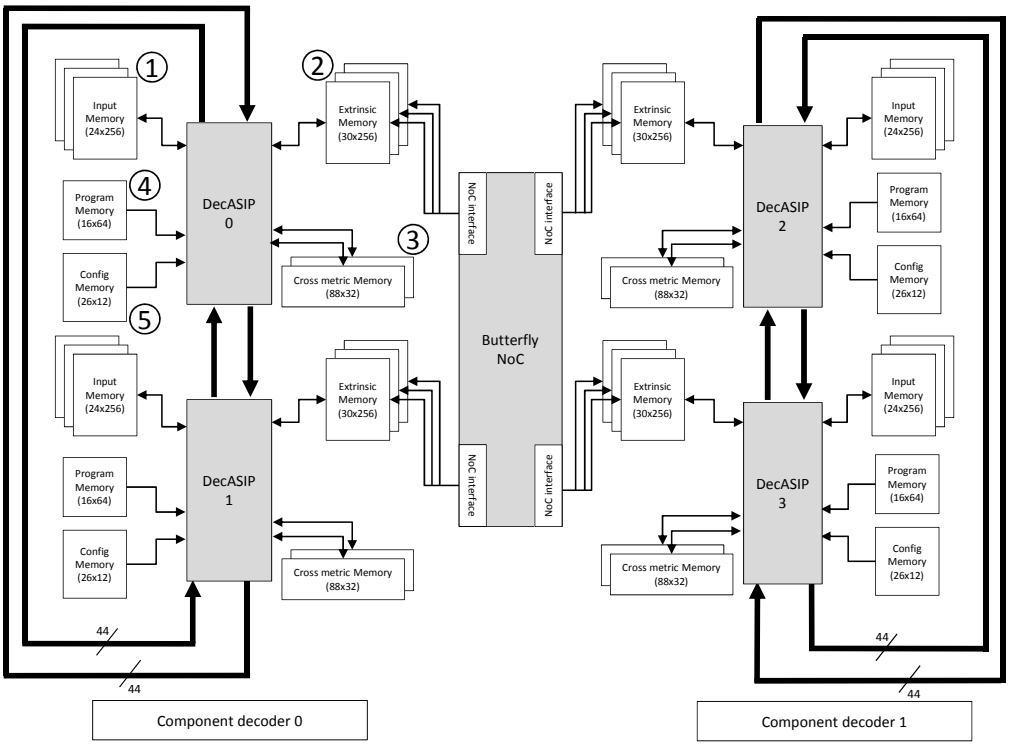


Figure 1.16: *UDec system architecture example with 2x2 ASIPs*

memories which hold state metric values ③. Moreover, each ASIP is configured through a program memory ④ and a configuration memory ⑤. The configuration memory contains all parameters required to perform the initialization of the ASIP while the program memory contains the instructions in order to perform the decoding algorithm. The detailed content of these two memories determining the configuration of the DecASIPs is described in the next chapter.

At the SISO decoder level, multiple SISO decoders are supported by the use of sub-blocking with message passing for boundary state metric initialization through two 88-bit ring buses as shown in Figure 1.16. To increase the use of parallelism degree, shuffled decoding is used by employing multiple SISO decoders grouped into natural (component decoder 0) and interleaved (component decoder 1) domains. Finally, the generated extrinsic information generated is sent to the other domain through the Butterfly NoC interconnection.

When a new frame has to be decoded, the first step consists to load the new configuration parameters and a new program in the configuration memory and program memory of each DecASIPs. Currently, there is no configuration infrastructure associated with the UDec architecture to perform this task. The content of these memories are generated at design time and are loaded from a host

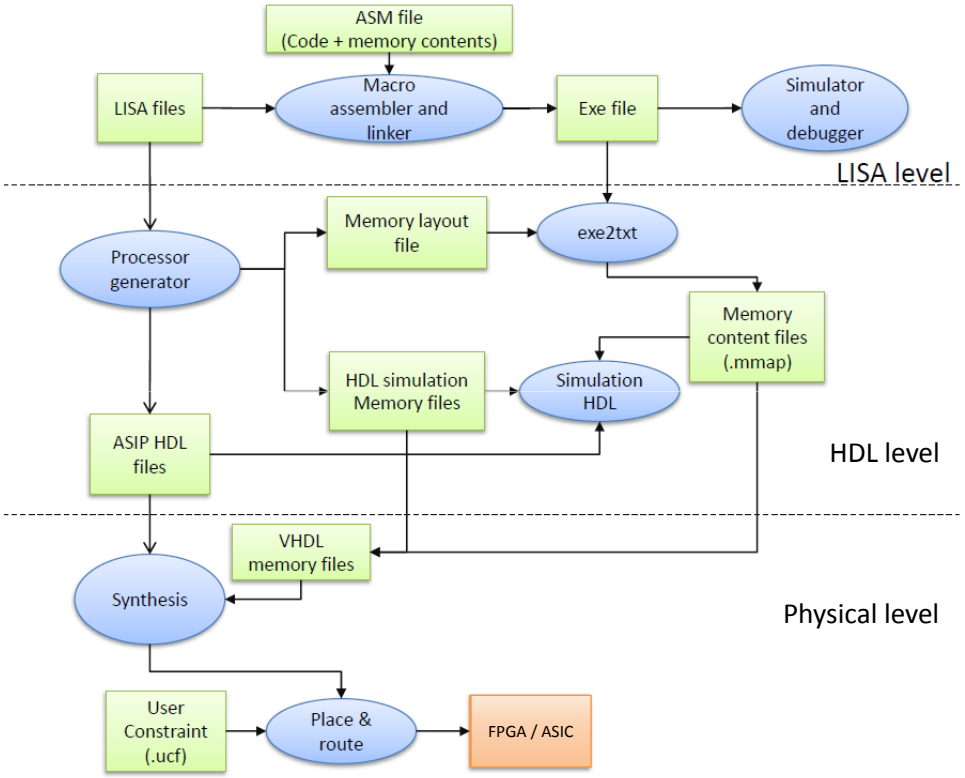


Figure 1.17: *LISA-based ASIP architecture design flow*

PC for FPGA prototyping or thanks to a simulation test bench for simulation process. Then, ASIPs are initialized by reading the content of their configuration memory. Depending on the decoding mode, the two component decoders can run one after the other (serial mode) or concurrently (shuffled mode). Finally, when the decoding process is completed for the fixed number of iterations, the DecASIPs are reseted waiting for a new frame to decode.

1.4.1 Overiew of the DecASIP processor

The initial DecASIP considered in this thesis work corresponds to the DecASIP developed during the thesis work of Purushotham Murugappa [42]. In the context of this thesis work targeting Turbo-decoding only, the support of LDPC decoding introduced in the initial DecASIP is not considered. The ASIP is developed using the Synopsys Processor Designer tool that uses LISA ADL (Architecture Description Language) [56]. This language provides a high flexibility to describe instruction set and provides an efficient solution to describe complex pipelines. Figure 1.17 illustrates the design flow adopted with the Processor Designer tool.

The ASIP design flow is divided in three abstraction levels.

LISA level: 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 file can be later used for simulation and debugging.

HDL level: at this level the design is translated to VHDL equivalent models along with the memory architecture and HDL simulation memory files. A *exe2txt* script that automatically converts memory initialization files described in ASM to equivalent text (*.mmap*) is provided. These *.mmap* files can be used for memory initialization in VHDL.

Physical level: at this level, ASIC or FPGA tools can be used for the hardware implementation. 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.

The DecASIP implements the Max-Log MAP algorithm for WiMAX, DVB-RCS and LTE standards. It supports both single and double binary convolutional turbo codes and implements radix-4 trellis compression technique for SBTC mode. This gives the possibility to decode two source data bits at the same time, thus increasing the throughput by two in this mode and allowing an efficient hardware resources sharing with the DBTC mode. Large frames are processed by dividing the frame into N windows each with a maximum size of 64 symbols. Each ASIP can manage a maximum of 12 windows.

The DecASIP is modeled as a processor with 10 pipeline stages as shown in Figure 1.18. The pipeline stages from *BM1* to*EXTR-CH* implement the computations related to the decoding process. First, in the *BM1* stage, systematic and parity information bits are fetched from the *Input memories* and combined to form the systematic and parity symbols. In parallel, the extrinsic information is fetched from the *Extrinsic memories* and is scaled by the *Scale Factor* stored in the configuration memory. In the *BM2* stage, the systematic symbols and the extrinsic information are combined. Then, the branch metrics are calculated in parallel. The *EX* stage contains the adders required to compute the 32 recursion metric LLRs (α and β). First, this recursion unit is used to process the trellis step in the backward direction (β). Then, it is used to process the trellis step in

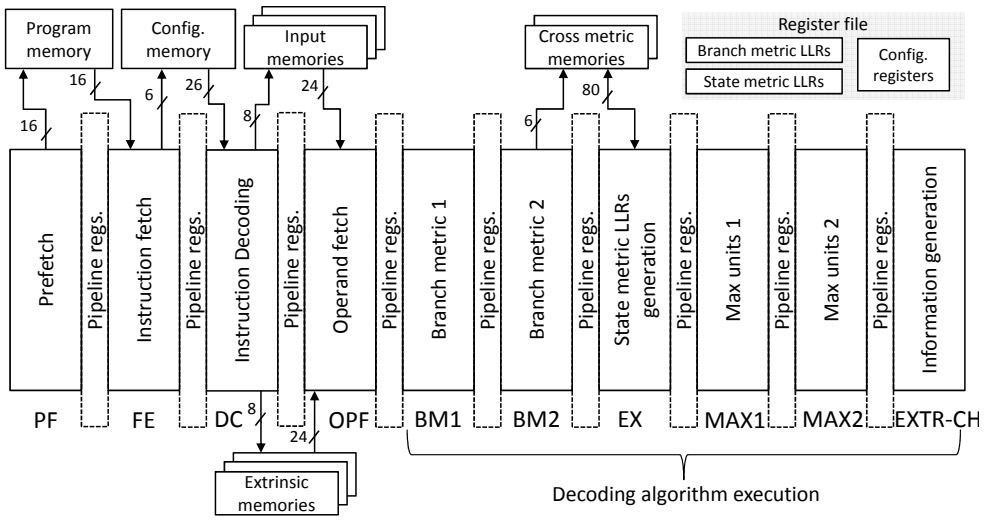


Figure 1.18: *Overview of the DecASIP pipeline stages with its register file and memory banks*

the forward direction (α). In *MAX1*, flexible max operators for calculating the recursion metric LLRs as defined in the Max-Log-MAP algorithm are provided. The max operators are reconfigured for the extrinsic generation phase. The final aposteriori LLRs are generated by the max operators in the *MAX2* pipeline stage. The last stage of the pipeline generates the extrinsic information. The generated extrinsic information packets also carry the address header which determines the destination DecASIP and the memory address at which the information has to be written. The memory address is generated at run-time by the ASIP during the processing using the address generator presented in the next section.

1.4.2 Interleaved/deinterleaved address generator

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 $j = 0...N-1$, $I(j) = (f1 \times j + f2 \times j^2) mod N$, 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 following expressions:

$$I(j+1) = (I(j) + G(j)) mod N \quad (1.12)$$

$$G(j) = (G(j-1) + 2f_1) mod N \quad (1.13)$$

The deinterleaved address pattern required by component decoder1 can be generated recursively as described here. Let the deinterleaved address sequence be

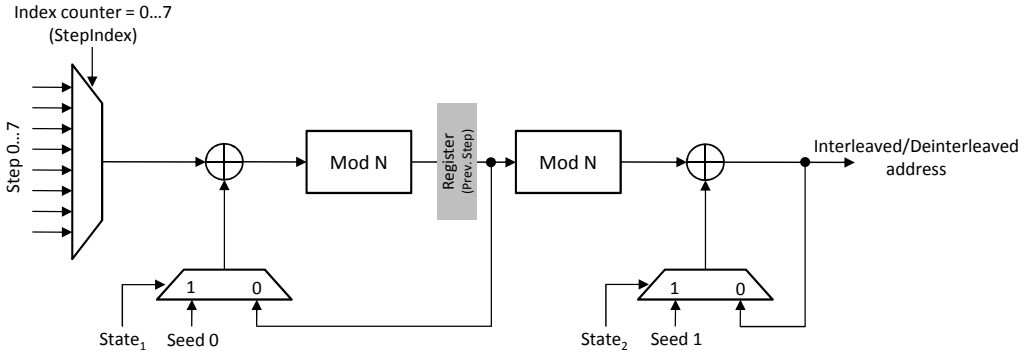


Figure 1.19: *ARP and QPP interleaved/deinterleaved address generator*

$D = [d_0, d_1, ...d_{N-1}]$. Taking a second order modulo-N linear circular difference of the sequence D gives step size values as given by the Equation (1.14) and (1.15) below. The number of steps (N_{steps}) depends on the frame size and can take at most eight different values.

$$D' = [d_0 - d_{N-1}, d_1 - d_0, ..., d_{N-1} - d_{N-2}]modN \quad (1.14)$$

$$D'' = [D'_0 - D'_{N-1}, D'_1 - D'_0, ..., D'_{N-1} - D'_{N-2}]modN \quad (1.15)$$

The following pseudo code illustrates the deinterleave address generation process:

for $i = 1 : N$
$d(i) = (d_{(i-1)} - D'_{i-1})modN;$
$D'_{(i)} = (D_{(i-1)} + D'((i-1)mod(N_{steps})))modN;$
end

Similar sequences is generated for the ARP interleaver specified in the WiMAX standard, where the number of steps obtained is maximum 4. Figure 1.19 presents the corresponding hardware generation architecture. Table 1.3 gives an example of the steps and seeds 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.

For a given configuration, the Seeds and Steps values are stored in the configuration memory of the DecASIP and is read during the initialization phase. The next section presents the Butterfly NoC used for extrinsic information exchanges.

1.4.3 NoC messages

The Butterfly NoC implemented in the UDec architecture consists of two unidirectional Butterfly NoC as shown in Figure 1.20 through an example with four

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 1.3: *Interleaved/deinterleaved address generation step and seed values*

DecASIP per component decoder. The Transmission Network Interface (NI_{Tx}) has the extrinsic message and its attached destination address as input. The address is a global address; it takes value from 0 to blocksize-1. This address cannot be used as it is to route the extrinsic messages through the NoC. The NI_{Tx} transforms this global address to a routing information and in a local address (@ Local). The routing information (3 bits considering the architecture of Figure 1.20) is used by the NoC to transport the message to the appropriate output router at each routing stage. The local address is used by the NI_{Rx} to write the message in the appropriate location in the appropriate memory bank.

In SBTC mode, two NoC packets are generated corresponding to the systematic LLR bits (S_0,S_1) processed as a symbol (by the radix-4 trellis compression technique). As these two addresses can be different they are packetized over two NoC packets each of width 26 bits. This does not cause any congestion in the NoC as two packets are generated every two clock cycles as explained in the previous section. The packet structure for SBTC mode is shown in Figure 1.21(a)

In DBTC mode, three normalized extrinsic information quantized to 8 bits is sent across the NoC in two packets as shown in Figure 1.21(b). The address fields of these two NoC packets are the same as they are targeting to the same location of the other component decoder.

1.4.4 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 DecASIP area of 175930 μm^2 (logic only) with a clock frequency of 500 MHz. Furthermore, the total memory area

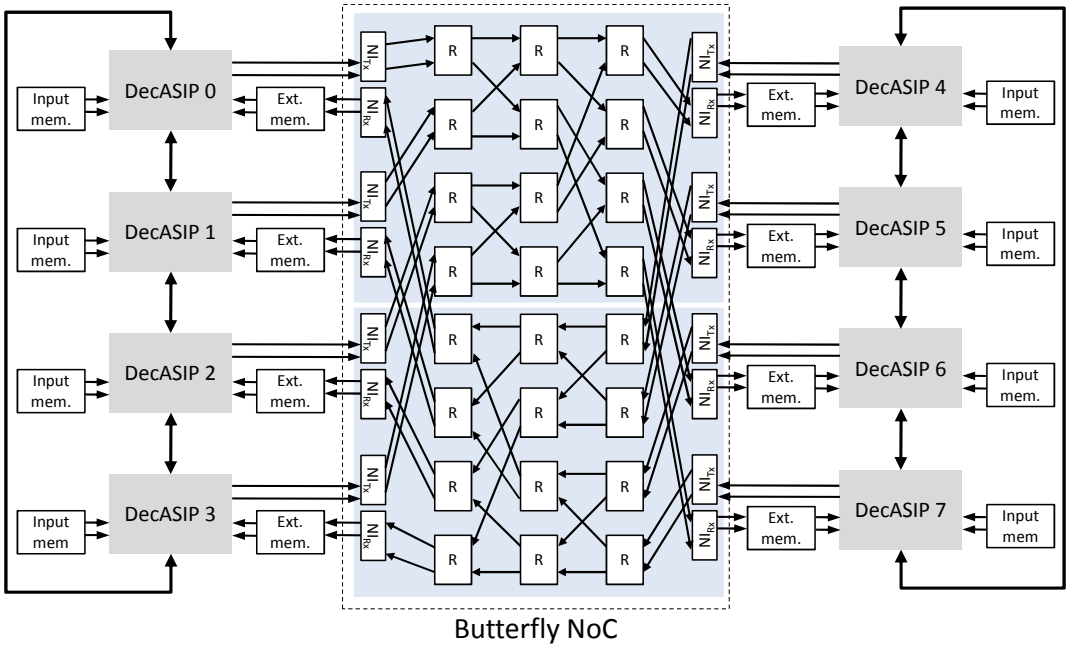
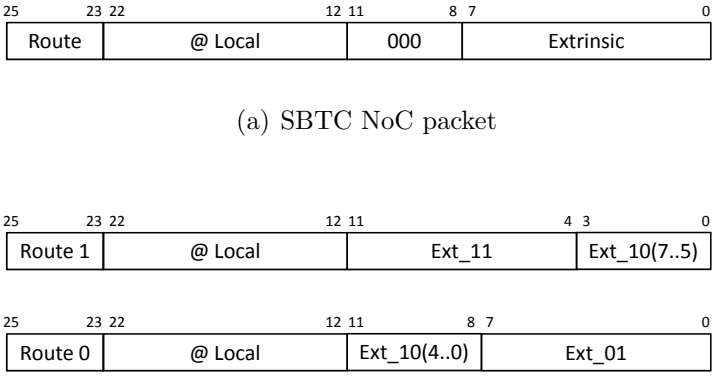


Figure 1.20: *Butterfly NoC structure as used in the UDec architecture*



(a) SBTC NoC packet

(b) DBTC NoC packets

Figure 1.21: *NoC packets format using Butterfly NoC*

for a single ASIP is 216179 μm^2 targeting 65nm CMOS technology.

Equation (1.16) gives the maximum throughput reached by the UDec architecture enabling the shuffled parallelism. An average of $N_{instr} = 4$ instructions

per iteration are needed to process 1 symbol which is composed of 2 bits.

$$Throughput = \frac{2 \times F_{clk} \times (N_{ASIP}/2)}{N_{instr} \times N_{iter}} \quad (1.16)$$

where F_{clk} and N_{iter} are the clock frequency of the system and the number of decoding iterations respectively. This throughput is divided by two when the shuffled decoding is disabled.

1.5 Summary

This chapter provided the basic background on the considered application of turbo decoding. The different parallelism levels which can be exploited in a multiprocessor implementation of a Turbo decoder was also explained. Moreover, an overview of state-of-the-art efforts in flexible solution for Turbo decoding was addressed. The proposed overview presents a selection of recent work related to the thesis scope in terms of dynamic reconfiguration support of Turbo decoding in order to clarify the position of the proposed contributions.

The chapter has also presented the architecture of an initial multi-ASIP Turbo decoder called UDec. This platform has been developed in previous thesis work at the Electronic department of Telecom Bretagne in Brest. This architecture was optimized to provide high performance and high flexibility. However, the dynamic reconfiguration requirement was not addressed. This initial work constitutes the starting point of this thesis work.

2

RDecASIP: optimized DecASIP for an efficient reconfiguration

This second chapter presents the optimization of the DecASIP processor in order to offer an efficient dynamic configuration of the multi-ASIP UDec architecture for Turbo decoding. The chapter starts with an analysis of the program and configuration memory content of the initial DecASIP architecture. This is followed by the proposed optimizations to reach an efficient dynamic reconfiguration of the DecASIP. These optimizations lead to the implementation of a new processor namely RDecASIP. Finally, implementation results and architectural choices are discussed.

2.1 Initial DecASIP configuration

An overview of the DecASIP architecture was provided in the previous chapter (Section 1.4). The DecASIP is the main element of the UDec architecture and proposes a flexible architecture allowing an efficient multi-ASIP implementation in order to reach high throughput of emerging and future communication standards. However the requirement of dynamic reconfiguration was not addressed. Despite its high flexibility, it presents some lacks to offer an efficient dynamic reconfiguration. Its configuration is realized through the configuration and program memories which are detailed in the following sections.

2.1.1 Configuration memory

The initial configuration memory of the DecASIP has been designed to contain configuration parameters for LDPC and Turbo decoding as shown in Table 2.1. The parameters for Turbo and LDPC decoding are shown in gray and white cells respectively. The configuration parameters for Turbo decoding are described in Table 2.2.

@	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			State				TurboInitIteration				Mode				ASIPId				N_{ASIP}			
1	-	-	-	-	N_{steps}				$\lceil \frac{Z}{(2*N_A)} \rceil$				-	-	-	-	NumRows				-	(Z-1)%2		
2	Param0								ScalingFactor								LastWindowSize							
3	Param2								zsize								Param1							
4	-												Parity check											
8	-												Step 0											
9	-												Step 1											
10	-												Step 2											
11	-												Step 3											
12	-												Step 4											
13	-												Step 5											
14	-												Step 6											
15	-												Step 7											
16	-												Seed 0											
17	-												Seed 1											
18	-												Turbo/LDPC FrameSize (in Bits)											
19	-												Prev. Step											
24	$\Pi_{1,1}$								$\Pi_{1,3}$								$\Pi_{1,5}$							
25	$\Pi_{1,2}$								$\Pi_{1,4}$								$\Pi_{1,6}$							
26	$\Pi_{2,1}$								$\Pi_{2,3}$								$\Pi_{2,5}$							
27	$\Pi_{2,2}$								$\Pi_{2,4}$								$\Pi_{2,6}$							
:	:								:								:							
:	:								:								:							
48	$\Pi_{12,2}$								$\Pi_{12,4}$								$\Pi_{12,6}$							

Table 2.1: *Config memory contents of the DecASIP*

They determine the configuration of the address generator for extrinsic information exchange (StepIndex, State, N_{steps}, Steps, Seeds and Prev. Step), the

decoding process (Mode, ScalingFactor, LastWindowSize, FrameSize), the multi-ASIP management (ASIPId, N_{ASIP}) and the debug mode (TurboInitIteration, State MSB). The LDPC parameters will not be taken into account in the rest of this document since this thesis work targets Turbo decoding only.

The configuration parameters contained in the configuration memory of the DecASIP are read and stored in internal registers of the ASIP during an initialization phase of the ASIP. The scheduling of this step is defined in the program stored in the program memory. For the initialization step, the DecASIP reads a configuration memory line each clock cycle. The number of read to be done is directly inserted in a loop instruction word of the program. In the next section, the instruction-set of the program for Turbo decoding is explained and program examples are given.

Parameter	Description
StepIndex	sets the initial value of the counter in interleaved/deinterleaved address generator
State	sets the state values for interleaved/deinterleaved address generation. Moreover the MSB is used to define if the extrinsic memory has to be read for the first decoding iteration or not (used for debug)
TurboInitIteration	sets the initial number of iteration counter (used for debug)
Mode	sets the ASIP mode for DVB-RCS, WiMAX or LTE standard. Moreover the MSB is used to inform the ASIPs concerned by tail bits in SBTC mode
ASIPId	sets the Identifier of the ASIP in a multi-ASIP context.
N_{ASIP}	sets the number of ASIPs used in a multi-ASIP context
N_{Steps}	sets the number of steps as defined in Section 1.4.2
ScalingFactor	sets the extrinsic information scaling factor
LastWindowSize	sets the length of the last window
Step 0...7	sets the number of steps values as defined in Section 1.4.2
Seed 0...1	sets the number of seeds values as defined in Section 1.4.2
FrameSize	sets the size of the frame to decode
Prev. Step	sets the register value shown in Figure 1.19 for interleaved/deinterleaved address generation

Table 2.2: *Configuration parameters of the DecASIP for the Turbo decoding mode*

2.1.2 Program memory

An assembly code example of the DecASIP allowing the decoding of one data frame in DBTC mode is as shown in Listing 2.1. First, the ASIP is initialized by reading configuration parameters from the configuration memory. If the *Repeat until flag for X times* instruction is decoded, the instructions until *flag* are repeated X times. The instruction *ASIP_INIT* initialises the ASIP registers with parameters read from the configuration memory. Then, the current window counter value, the window size and the final window number that have to be processed by the ASIP are configured. The *Repeat until* instruction in line 9 controls the number of decoding iterations while the *PUSH* instruction increments the iterations counter. The *ZOLB* instruction in line 12 enables the execution of backward and forward recursion metric calculation *window size* times set in line 6. In case the current window being processed is the last window of the sub-block assigned to the ASIP, the execution of backward and forward recursion metric calculation is performed *LastWindowSize* times. This parameter is defined in the configuration memory (Table 2.1). Finally, extrinsic information packets are generated and the boundary state metrics are sent to the next ASIP of the same component decoder. *NOP* instructions are inserted at the end of the program in order to flush the pipeline.

Listing 2.1: *Example of an assembly code in DBTC mode*

```
1          Repeat until _init for 63 times;   repeats instructions
2          from 3 to _init for X times
3          NOP;
4          ASIP_INIT;
5  _init:  NOP;
6          SET_WindowsInit 0, 63;   set window counter (0) and window
7          size (63+1)
8          SET_WindowsN 6;   set Number of Windows (6)
9          Repeat until _LOOP for 10 times; set Number of decoding
10         iterations (10)
11         PUSH;   Push the Number of iterations counter
12         ZOLB _RW1,_CW1,_LW1; repeat instr 16 and 18 Window size
13         number of times followed by repeat 22 and 24 Window size number
14         of times
15         NOP;
16         DATA LEFT ADD metric column2 ;calculate backward recursion
17         metric
18 _RW1:   NOP;
19         EXCH_BETA_ALPHA; save old and fetch new boundary state across
20         windows within the ASIP
```

```
21 _CW1:    NOP;
22          DATA RIGHT ADD metric column2; calculate forward recursion
23          metric
24 _LW1:    EXTCALC ADD info line2; generate extrinsic information
25          WINDOW_INIT ALPHABETA_0_3 exchange boundary states(0..3)
26          across windows with the next ASIP
27          NOP;
28          WINDOW_INIT ALPHABETA_4_7 exchange boundary states(4..7)
29          across windows with the next ASIP
30          NOP;
31          NOP;
32          NOP;
33          NOP;
34          NOP;
35          NOP;
36          NOP;
37 _LOOP:   NOP;
```

Listing 2.2 shows an example of assembly code of the DecASIP in SBTC mode including tail bits decoding. In DBTC mode, the first step concerns the initialization of the ASIP with the configuration parameters stored in the configuration memory. Then, from line 6 to 21, instructions are dedicated to tail bits decoding. The trellis termination strategy used in SBTC of the LTE standard is through zero padding and is achieved by inserting 3 zeros at the end of the encoding process (Section 1.1.3.1). This leads to process an extra 1 and half symbols (after radix-4 trellis compression) by the last ASIP of the component decoder. 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 the last ASIP of the component decoder. 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 two symbols separately as a window (as shown in listing 2.2). The instructions in lines 13 and 15 are repeated in order to process these tail bits by initializing the fetch units to one more than the last window, and executing the two symbols in the backward direction. After this tail bits processing is completed, the window counter is initialized back to zero (line 22) and the execution continues similar to the DBTC mode. During the tail bits processing, all other ASIP of the component decoder execute (NOP) instructions to maintain the synchronization ASIPs.

Listing 2.2: *Example of an assembly code in SBTC mode including tail bits decoding*

```
1          Repeat until _init for 63 times;   repeats instructions
2          from 3 to _init for X times
3          NOP;
4          ASIP_INIT;
5  _init:  NOP;
6          SET_WindowsInit 6, 1;   set window counter and window
7          size for tail bits decoding
8          SET_WindowsN 7;   set Number of Windows for tail bits decoding
9          TailBits 361; set absolute address of tail bits
10         ZOLB _RW,_RW,_RW; repeat instr 13 and 15 Window size
11         NOP;
12         Intructions 13 and 15 if last ASIP of the component decoder else NOP
13         DATA LEFT ADD metric column2 ;calculate backward recursion
14         metric
15 _RW:    NOP;
16         NOP;
17         EXCH_BETA_ALPHA; save old and fetch new boundary state across
18         windows within the ASIP
19         NOP;
20         NOP;
21         NOP;
22         SET_WindowsInit 0, 63;   set window counter (0) and window
23         size (63+1)
24         SET_WindowsN 5;   set Number of Windows (5)
25         Repeat until _LOOP for 10 times; set Number of decoding
26         iterations (10)
27         PUSH;   Push the Number of iterations counter
28         ZOLB _RW1,_CW1,_LW1; repeat instr 32 and 34 Window size
29         number of times followed by repeat 38 and 40 Window size number
30         of times
31         NOP;
32         DATA LEFT ADD metric column2 ;calculate backward recursion
33         metric
34 _RW1:   NOP;
35         EXCH_BETA_ALPHA; save old and fetch new boundary state across
36         windows within the ASIP
37 _CW1:   NOP;
38         DATA RIGHT ADD metric column2; calculate forward recursion
39         metric
40 _LW1:   EXTCALC ADD info line2; generate extrinsic information
```

```
41              WINDOW_INIT  ALPHABETA_0_3  exchange boundary states(0..3)
42              across windows with next the ASIP
43              NOP;
44              WINDOW_INIT  ALPHABETA_4_7   exchange boundary states(4..7)
45              across windows with next the ASIP
46              NOP;
47              NOP;
48              NOP;
49              NOP;
50              NOP;
51              NOP;
52              NOP;
53  _LOOP:      NOP;
```

Based on the description of the program and the configuration memories of the initial DecASIP, the next section highlights the configuration lacks of the initial DecASIP and proposes optimization solutions to reach an efficient dynamic configuration.

2.2 Proposed optimizations for an efficient dynamic configuration

In this section, several optimizations are proposed to reach an efficient dynamic reconfiguration of the DecASIP architecture. The first optimization is related to the storage of configuration parameters. The second optimization deals with the way used to load the configuration memory through the configuration memory organization. The third optimization corresponds to the development of a generic program independent of the configuration to be performed and tackles the management of interframe delay since the DecASIP is currently reseted after each processed frame. The final proposed optimization technique deals with the multi-configuration storage.

2.2.1 Configuration parameters storage

In the initial DecASIP architecture, the flexibility is spread over the configuration memory and the program memory. Indeed, flexible parameters are directly inserted in instruction words. From the programs described in Section 2.1.2, seven flexible parameters can be extracted. These parameters are presented in Table 2.3. We can notice that some parameters included in the program are fixed. Indeed, the window size for tail bits decoding is fixed to 1 due to the SBTC

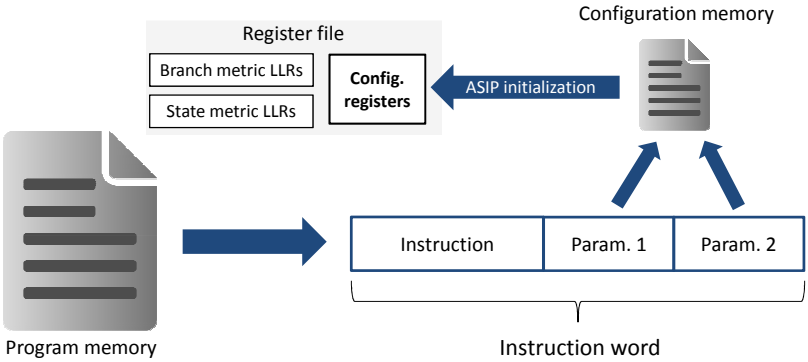


Figure 2.1: *Flexible parameters transfer to configuration memory*

structure and the initial window counter value for *normal* decoding is fixed to 0 i.e. the first window to decode is always the window 0.

Parameter	Description
Config. Size	sets the number of configuration memory lines the DecASIP has to read
WindowSize	sets the window size
WindowN_norm	sets the final window counter value for normal decoding
WindowN_tail	sets the final window counter value for tail bits decoding
WindowID_tail	sets the initial window counter value for tail bits decoding
@ Tail bits	sets the address of tail bits in input memory
Max_Iteration	sets the number of maximum decoding iteration the DecASIP has to performed

Table 2.3: *DecASIP program flexible parameters*

To reach reconfiguration efficiency, all flexible parameters present in instruction words are moved from the program memory to the configuration memory as shown in Figure 2.1. An exception is done for the configuration size parameter (*Config. Size*), which is read from extra input pins of the DecASIP in order to simplify the setting of the initialization loop counter when the ASIP is started. This solution allows to reconfigure a single memory to change all the configuration parameters (instead of loading both new program memory and configuration memory). Furthermore, once the DecASIP is configured, the configuration memory can be accessed without any conflict since the configuration is loaded inside

internal registers of the DecASIP during the initialization step. This is a key point to prepare the next configuration if necessary. Indeed, the entire next configuration can be loaded in the configuration memory during the processing of the current data frame. Thus the configuration loading can be partially or completely masked. If the configuration loading is completely masked, the configuration overhead consists in the initialization step only. However, this modification impacts the area of the configuration memory since it is necessary to store more parameters. It also impacts the number of registers inside the DecASIP in order to store these parameters after the initialization step. Finally, it impacts the initialization step duration since this step consists in reading the configuration memory and then storing each parameter in the corresponding register. The program in its current form contains seven configuration parameters. All these parameters (except for *Config. Size*) can be moved to the configuration memory. So six parameters have to be added to the configuration memory to obtain a parameter independent program memory. The next sub-section presents a way to integrate these new parameters in a dedicated memory organization.

2.2.2 Configuration memory organization

In order to improve the reconfiguration of the DecASIP, it is essential to analyze the organization of the configuration memory. The parameters stored in the configuration memory present different properties. They can be classified in four categories as follows:

1. Domain dependent (for component decoder 0 or component decoder 1 in Figure 1.16).

2. Identical for all DecASIPs.

3. Different for all DecASIPs.

4. Different for the last DecASIPs of each component decoder which decode the tail bits in SBTC mode.

All these characteristics need to be taken into account in order to build a low latency configuration process. Furthermore, in a multi-ASIP context, it is necessary to only configure DecASIPs required for a specific execution context (it means that only a subset of ASIPs may be required depending on the target performance). A dedicated memory organization should allow an efficient broadcasting of the configuration parameters to the required DecASIPs. Thus, depending on the previously described categories, the parameters can be gathered in four groups which occupy four different parts of the configuration memory.

Width (in bits)	Depth	Usage (in %)	Configuration time (in cycles)
32	10	79.1	12
28	12	75.3	13
26	**12**	**97.3**	**13**
24	16	65.9	17
20	18	70.3	19
16	20	79.1	22
14	22	82.1	24

Table 2.4: *Configuration memory architecture alternatives*

As different memory architectures can be considered, it is important to define the best size for the memory. Table 2.4 shows that the choice of the width and depth of the memory impacts the time required to load the configuration into the DecASIP and the area of the memory on the chip. We evaluate five solutions from 14 bits to 32 bits memory width. Since the number and the size of each parameter are fixed, the depth of the memory decreases when the width increases. Furthermore, in order to take into account the final implementation technology of the different memory alternatives, the depth and width of each memory must be a multiple of 2. It represents the minimal constraint to design the memory with the considered memory generator tool. Depending on the alternative, the level of memory usage changes. Results of Table 2.4 show that a 26 bits memory width is the most efficient one (97.3% area usage) while an initial 24 bits memory width is inefficient (65.9% area usage).

The time required to configure the ASIP is proportional to the number of memory lines to be read. Each clock cycle, the ASIP reads one data from the memory and configures the corresponding register. One more clock cycle is necessary to initialize the reading loop, and another cycle is required to complete the initialization of the ASIP. For example, 12 cycles are needed to load the configuration in the ASIP with a 32 bits memory width while 24 cycles are necessary for a 14 bits memory width. Moreover, the memory width has an impact on the global multi-ASIP platform. Indeed, a large memory width increases the number of connections between each ASIP and its configuration memory and between each configuration memory and the configuration infrastructure in charge of the configuration parameters transfer in the memories. Thus, a trade-off between the initialization time, the memory usage and the global impact on the platform has to be found. The 26 bits memory width is an interesting trade-off. This memory alternative offers a fast configuration time (13 cycles) and the highest memory usage efficiency (97.3%). Finally, the impact on the entire platform is low compared to the current solution that implements a 24 bits memory width.

bit	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
@0	-																					Tail	ASIPId			
@1	Turbo Seed 0													Turbo Seed 1												
@2	Turbo Seed 0			-		TurboInitIteration					Maxiteration								State			NumSteps				
@3	Turbo Step 0										Turbo Step 1															
@4	Turbo Step 2										Turbo Step 3															
@5	Turbo Step 4										Turbo Step 5															
@6	Turbo Step 6										Turbo Step 7															
@7	-	@ Tail bits												Scaling Factor									Mode			
@8	Turbo PrevStep										Blocklength in bits															
@9	-	NumASIPs			StepIndex						WindowSize								LastWindowSize							
@10	-	CtWindowN_norm									WindowID_tail								WindowN_tail							

Table 2.5: *New proposed organization of the configuration memory*

Table 2.5 presents the 26 bits width configuration memory organization. The memory is organized as follows: (1) from address @0 to @1, parameters can be different for each ASIP. Furthermore, to optimize the initialization step of the ASIP, the parameter *Tail* which indicates if the ASIP has to perform or not the tail bits is included in this group. Only the last two ASIPs are concerned by the tail bits in a single binary turbo code mode; (2) from address @2 to @6, the parameters are domain dependent; (3) from address @7 to @10, the parameters are the same for all ASIPs. This organization allows a good way for a fast reconfiguration at the platform level. Indeed, multicast and broadcast mechanisms can be used to load the configuration in order to minimize the data transfers load. In this context, two multicast transfers are necessary to send domain dependent parameters stored from address @2 to @6 of each configuration memory to all ASIPs and one broadcast transfer for parameters stored from address @7 to @10 of each configuration memory that are the same for all ASIPs. Finally, unicast transfers are used to load the ASIP dependent parameters from address @0 to @1 of each configuration memory.

2.2.3 Unified program

As shown in Figure 2.2, three different programs are currently used in the DecASIP (Section 2.1.2): two programs for SBTC mode and one program for DBTC mode. In single binary mode, after the initialization step, the last two DecASIPs of each component decoder have to perform the tail bits while other DecASIPs execute NOP operations. So, a particular program is loaded in these last two ASIPs. In DBTC mode, the frames are decoded after the initialization step.

These three programs can be merged to one unique program presented in Listing 2.3. This solution allows to consider a single program for different configurations and thus reduces the configuration load. For this purpose, the program which integrates the tail bits computation is used as a reference program. From instruction 1 to 4 in Listing 2.3, the initialization phase of the ASIP is performed. The number of iterations of the initialization loop (i.e. *Config. Size*) is now read from a set of input pins instead of being directly fixed in the instruction word. The new memory organization previously described requires 11 read accesses. Hence the initialization step requires 13 clock cycles. (i.e. one clock cycle to read the memory, one more clock cycle to initialize the loop, and another clock cycle to complete the initialization of the ASIP).

Instructions from line 5 to 14 concern the tail bits decoding in single binary mode. In DBTC mode, no tail bits have to be decoded. In order to minimize the impact on the complexity of the ASIP by adding new control instructions, we have chosen to modify the *Fetch* (FE) pipeline stage in order to detect and replace the instructions for tail bits with *NOP* instructions if the ASIP is not concerned. This

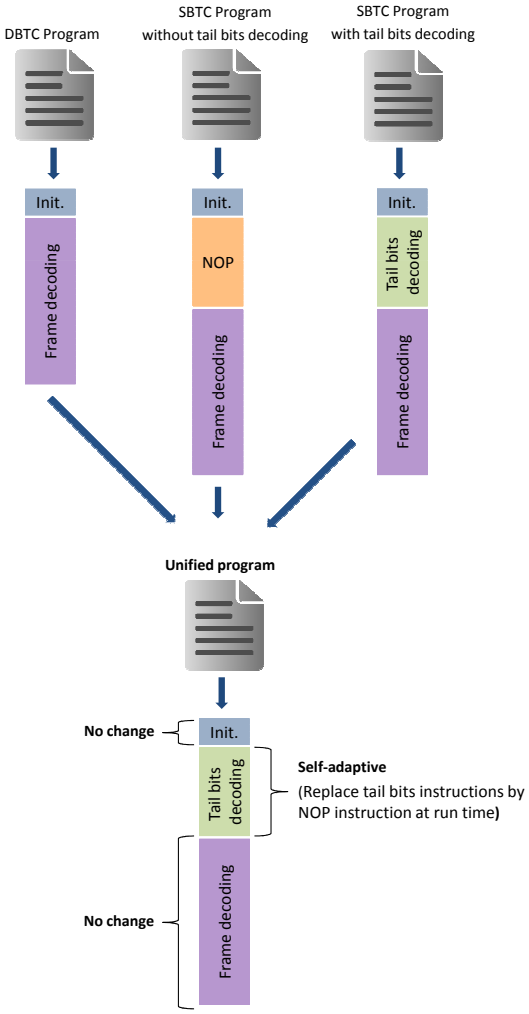


Figure 2.2: *Unified program for SBTC and DBTC modes*

is done by checking the *Tail* register value read from the configuration memory. If *Tail*=0, when the program counter reaches the address corresponding to one of the tail bits instructions, a *NOP* instruction is sent to the *Decode* (DC) pipeline stage instead of the read instruction word. Hence, using a unique program in this mode adds 14 extra *NOP* instructions before the decoding which corresponds to tail bits computation in single binary mode. However, the extra introduced clock cycles are negligible regarding the number of clock cycles required to perform the decoding on one entire frame.

Instructions from line 18 to 40 concern the decoding of the frame. This part of the program is common for SBTC and DBTC modes. In order to create a generic program, the instructions *Repeat until*, *SET_WindowsN* and *SET_WindowsInit*

have to be independent of the configuration parameters. For this purpose, the behavior of these instructions is now dependent of the parameters stored in the configuration memory (WindowSize, WindowN_norm , Max_Iteration).

Listing 2.3: *Unified assembly code for DBTC and SBTC modes*

```
1          Repeat until _init for Config. Size    times;
2          NOP;
3          ASIP_INIT;
4  _init:  NOP;
5          SET_WindowsInit WindowID_tail, 1;
6          SET_WindowsN WindowN_tail;
7          TailBits 361;
8          ZOLB _RW,_RW,_RW;
9          NOP;
10         DATA LEFT ADD metric column2 ;Replaced by NOP when
11         Tail =0
12 _RW:    NOP;
13         NOP;
14         EXCH_BETA_ALPHA; Replaced by NOP when Tail =0
15         NOP;
16         NOP;
17         NOP;
18         SET_WindowsInit 0, WindowSize;
19         SET_WindowsN  WindowN_norm;
20         Repeat until _LOOP for Max_Iteration times;
21         PUSH;
22         ZOLB _RW1,_CW1,_LW1;
23         NOP;
24         DATA LEFT ADD metric column2 ;
25 _RW1:   NOP;
26         EXCH_BETA_ALPHA;
27 _CW1:   NOP;
28         DATA RIGHT ADD metric column2;
29 _LW1:   EXTCALC ADD info line2;
30         WINDOW_INIT ALPHABETA_0_3
31         NOP;
32         WINDOW_INIT ALPHABETA_4_7
33         NOP;
34         NOP;
35         NOP;
36         NOP;
```

```
37          NOP;
38          NOP;
39          NOP;
40  _LOOP:  NOP;
41          ASIP_STOP;
```

In order to allow the decoding of consecutive frames without reconfiguration, the instruction *ASIP_STOP* in line 41 tackles the interframe management. It is used to trigger the reset of all internal registers which are read during the decoding of the first symbol of a new frame. This reset avoids interferences caused by previous data generated during decoding of the previous frame. Afterwards, ASIP is stopped, and waits for a new trigger on an additional *Restart* pin to perform a new frame with the same configuration without a new initialization phase of the ASIP. When a new configuration has been loaded in the configuration memory, the *Reset* pin is triggered to start the program by the initialization loop.

2.2.4 Multi-configuration storage

In a multi-mode and multi-standard context in which mobile terminals deal with several communication standards and are able to concurrently execute applications that simultaneously access to the network, an efficient technique is to have several configurations available in the configuration memory and to switch between one to another quickly. Furthermore, when configurations are often executed, storing these configurations in the configuration memory saves data transfers to load the memory and consequently reduces the penalty due to configuration transfers. For this purpose, we need to design the RDecASIP to manage these different configurations. A simple way to address this point, is to add extra input pins in order to provide the address of the configuration that has to be loaded into the DecASIP. This solution is presented in Figure 2.3. An additional ASIP input defines the memory offset in the configuration memory in order to select one of the configuration loaded in the memory. This offset is added to the configuration memory address counter to form the final address used to drive the configuration memory address port.

The optimization techniques described in this section allow to reduce the reconfiguration impact: 1) locally through the optimization of the storage of configuration parameters to efficiently use the memory capacity and through the possibility to decode several frames without a new initialization step of the ASIP and 2) globally thanks to the new memory organization and the generic program which reduce the total configuration load to transfer when a new configuration

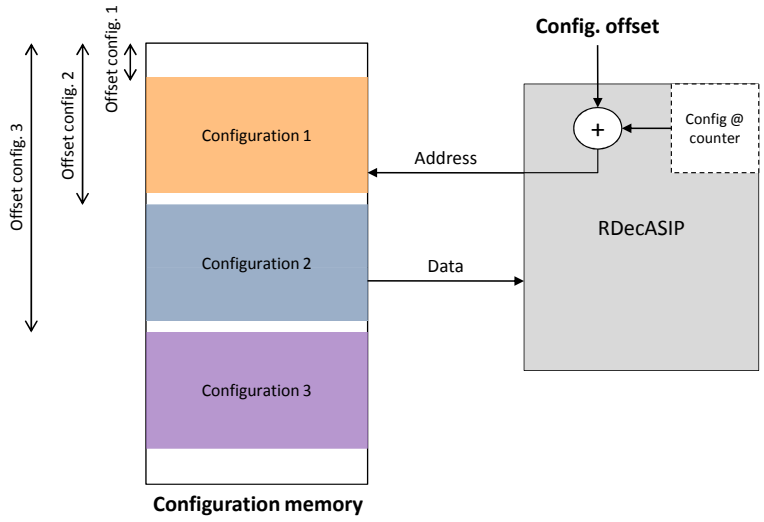


Figure 2.3: *Configuration memory address offset*

has to be performed. Moreover, the multi-configuration management is becoming a key feature to optimize the management of a mobile device which deals with several communication standards and applications. The next section presents the implementation results and the impact of the proposed optimizations on the initial DecASIP.

2.3 RDecASIP Implementation

This section analyzes the implementation results in terms of logic area overhead and configuration load of the optimized ASIP implementing the optimizations described in the previous section.

2.3.1 ASIC synthesis results

The optimizations described in the previous section have been implemented on the initial DecASIP. Synthesis of the previous and the new cores was done with 65nm CMOS technology with a clock frequency objective equals to 500MHz. To evaluate the impact of the new features on the ASIP area, we extracted the area synthesis results for each pipeline stage of the DecASIP and the optimized DecASIP, namely RDecASIP. Regarding the synthesis results presented in Table 2.6, five main design units have been impacted by the proposed optimizations: Prefetch (PF), Fetch (FE), Decode (DC) and Operands fetch (OPF) pipeline stages,

as well as the Register file of the processor. Indeed, the proposed optimizations were implemented along the pipeline stages as follows:

- PF: The PF stage insures the management of the new Restart pin of the ASIP. Indeed, when this pin is triggered, the program has to restart from a particular address instead of starting from the address 0x0 which is the first address of the program memory. In this case, the initialization step is skipped and the decoding process starts with the configuration parameters currently stored in the internal registers.

- FE: The FE stage insures the automatic replacement of instructions for tail bits computation by *NOP* when the ASIP is not concerned by tail bits decoding. This is done by checking the *Tail* register value read from the configuration memory. If *Tail*=0, when the program counter reaches the address corresponding to one of the tail bits instructions, a *NOP* instruction is sent to the *Decode* (DC) pipeline stage instead of the instruction word read in the program memory.

- DC: This stage is mainly impacted by the transfer of all flexible parameters in a unique configuration memory. Instead of a direct access to some parameters in instruction code words, parameters are now read from registers. Thus, the number of connections with the register file has been increased. It is also in charge of the internal counter incrementation to drive the configuration memory address taking into account the configuration memory offset.

- OPF: This stage is impacted by the new configuration memory organization since it is in charge of the parameter registers initialization. The area overhead comes from the increasing number of parameters in the configuration memory and by added control structures that manage the configuration size flexibility. Since more configuration parameters are read from the configuration memory, the number of connections with the register file has been increased to configure additional registers.

Regarding results from Table 2.6 for the complete ASIP area, we observe that despite of the different area overheads on the pipeline stages and on the Register file caused by our optimizations, the global area overhead for the complete processor is 5.1% (0.009 mm^2). Indeed, the most complex part of the pipeline consists of the execution stages that implement the decoding algorithm which are not directly impacted by the proposed optimizations. The observed area overhead is mainly due to the register file. So, the increasing complexity of the ASIP is mainly due to the additional internal registers used to store the configuration

Design unit	Area in μm^2		Diff. in μm^2
	DecASIP	RDecASIP	
PF	3176	3753	577 (+18.2%)
FE	970	1098	128 (+13.2)%)
DC	4958	5662	704 (+14.2%)
OPF	3485	3767	282 (+8%)
BM1	5246	5172	-74 (-1.4%)
BM2	3934	4105	171 (+4.3%)
EX	23296	23348	52 (+0.2%)
MAX1	13650	13999	349 (+2.5%)
MAX2	4906	4943	37 (+0.7%)
EXTR-CH	7135	7032	-103 (-1.4%)
Total Pipeline	70756	72879	2123 (+3%)
Register file	85298	92092	6794 (+8%)
Memory interface	15449	15530	81 (+0.5%)
Total logic	175930	184992	9062 (+ 5.1%)
Total memory	216179	214887	1292 (- 0.6%)

Table 2.6: *ASIC synthesis results for the initial DecASIP and optimized RdecASIP*

parameters read from the configuration memory. On the memory side, the total memory area for a single ASIP is quite similar for both DecASIP and RDecASIP, it is reduced by 0.6% (0.0046 mm^2).

2.3.2 Dynamic reconfiguration performance

In this section, the gain of proposed optimizations on reconfiguration timing performance are analyzed. For this purpose, the following reconfiguration steps are considered:

1. Memories loading: The first step of the configuration process is the transfer of the configuration parameters in the configuration memory of one or several ASIPs.

2. ASIP initialization: When the configuration parameters are available in the memory, the ASIP can start the initialization process. During this step, the ASIP reads the configuration stored in the configuration memory and initializes the internal registers. Then, the ASIP is ready to execute the computation on the input data frame.

Table 2.7 compares the configuration and program load (in bits) for the proposed RDecASIP, the initial DecASIP and the FlexiTreP presented in [44]. For

	Configuration memory	Program memory	1 ASIP	n ASIPs
RDecASIP	286	-	286	n.52+260+104
DecASIP	336	640	976	n.976
Gain	14%	100%	70%	90% ($n = 8$)
FlexiTreP [44]	383 + Interleaver table	∼1080	1463 + Interleaver table	n.1463 + n.Interleaver table
Gain	25%	100%	80%	93% ($n = 8$)

Table 2.7: *Configuration and program bit load comparison in bits*

one ASIP, we observe that RDecASIP can be configured with 286 bits instead of 976 bits thanks to the generic program described in Section 2.2.3 while 1463 bits and the complete interleaver table are required for the FlexiTreP. Moreover, the new memory organization proposed in Section 2.2.2 allows the optimization of the configuration memory loading. Indeed, parameters can be sent to several ASIPs through multicast and broadcast mechanisms. Thus, in a multi-ASIP context, each DecASIP has to be configured with its own configuration and program memory while configuration memory of the RDecASIP can be loaded using a multicast and broadcast mechanisms as follows: 52 bits are independently loaded in each ASIP. ASIPs that compose the same decoder component are loaded with 130 common bits. Finally, 104 configuration bits are broadcasted to all ASIPs. Thanks to this new configuration memory organization, the impact of the number of ASIPs on the configuration load is significantly reduced: n.52 bits instead of n.976 bits, where n is the number of activated ASIPs. For example, if 8 ASIPs are activated in a multi-ASIP platform, the configuration load to configure the 8 ASIPs is 7808 bits with the original DecASIP, 11704 bits for the FlexiTreP plus the interleaver tables and 780 bits with the proposed ASIP. In this case the configuration load is divided by 10 and 15 compared to the DecASIP and the FlexiTreP respectively.

The new configuration organization has also an impact on the initialization time of the ASIP. Indeed, for each new configuration, the ASIP reads the parameters from the configuration memory and initializes the internal registers. The new memory organization reduces the number of read accesses to the memory. Only 11 read accesses are necessary instead of 15 in the DecASIP, i.e. the initialization time is reduced by 4 cycles. Thus, the RDecASIP can be reconfigured in 14 clock cycles if we assume that one extra clock cycle is necessary to drive the Reset pin of the ASIP, i.e 11 + 2 clock cycles for the initialization step + 1 clock cycle.

The configuration memory can store several configurations. In the RDecASIP, 4 bits are available to define the MSB of the configuration address in the memory. Thus, a maximal of 16 configurations can be addressed with this implementation.

When the configuration corresponding to the next frame is stored in the configuration memory, the reconfiguration process consists in: 1) driving the input pins of the RDecASIP to indicate the location of the next configuration and 2) reseting the ASIP to launch the initialization step. In this case the RDecASIP can be reconfigured in 15 clock cycles if we assume that one extra cycle is necessary to drive the input pins of the RDecASIP which define the configuration location in the memory.

Furthermore, when the initialization step is performed, the ASIP does not read the configuration memory until the next initialization. Hence, during the computation on a data frame, configuration parameters can be loaded in the configuration memory. The memory loading process can be partially or totally masked depending on when the reconfiguration order is triggered. If the loading process is masked, the complete reconfiguration of a multi-ASIP platform represents an overhead of 15 clock cycles. This low (re)configuration time overhead allows the implementation of such an optimized ASIP in multi-ASIP architecture for future high throughput and low latency requirements.

It is important to note that the decoding performances of the RDecASIP and DecASIP are purely identical. Indeed, the pipeline architecture is still the same and both implementations are able to reach the maximum frequency of 500MHz. However, in DBTC mode, the RDecASIP requires 12 extra clock cycles after the initialization phase to start decoding. Since the initialization phase for the new version is 4 clock cycles shorter than for the original version, 8 extra clock cycles before the beginning of the decoding in DBTC mode are needed while it is able to start the processing 4 clock cycles before the DecASIP in SBTC mode. Nevertheless, these impacts can be neglected when compared to the decoding time of a data frame (thousands of clock cycles) that is identical for both version of the ASIPs.

2.4 Summary

This chapter started by providing an analysis of the configuration and program memory of the DecASIP. Based on this analysis, optimizations have been proposed and implemented in order to offer an efficient configuration of the DecASIP. The new version of the ASIP, called RDecASIP, provides a new configuration memory organization allowing efficient configuration parameters transfers in a multi-ASIP context by using both multicast and broadcast mechanisms, a generic program reducing the configuration load of the ASIP by 70 % and a multi-configuration storage management. ASIC logic synthesis results show that these optimizations introduce a low area overhead of 0.009 mm^2 while the decod-

ing performance and maximum clock frequency of the RDecASIP have remained identical to the initial implementation. Finally, the new configuration memory allows the reduction of the initialization step latency of the ASIP leading to a reconfiguration latency of the UDec platform in 15 clock cycles when the configuration parameters loading process is masked.

3

Reconfigurable multi-ASIP UDec architecture

This chapter addresses the dynamic configuration requirement at the multi-processor level for the considered turbo decoding application. Besides the optimization techniques proposed in the previous chapter regarding the ASIP architecture and the configuration parameters storage and transfer, further techniques can be explored and proposed in multi-ASIP integration context. Depending on the application requirement, the number and the location of the activated ASIPs can be dynamically tuned. Such a feature can have significant impacts on the communication networks connecting the ASIPs and on the multi-ASIP platform controller. In this context, several techniques are proposed and presented in the first part of this chapter considering a multi-ASIP platform for flexible turbo decoding. This is followed by the definition and implementation of a dedicated configuration infrastructure providing an efficient and low complexity solution for configuration data transfer to each configuration memory of the implemented RDecASIPs. Finally, implementation results and configuration timing performance are discussed.

3.1 Flexible UDec architecture

This section presents several techniques that we propose in order to increase the dynamic configuration ability in the context of multi-ASIP platform for flexible turbo decoding. These techniques concern the communication networks connecting the ASIPs and the multi-ASIP platform controller.

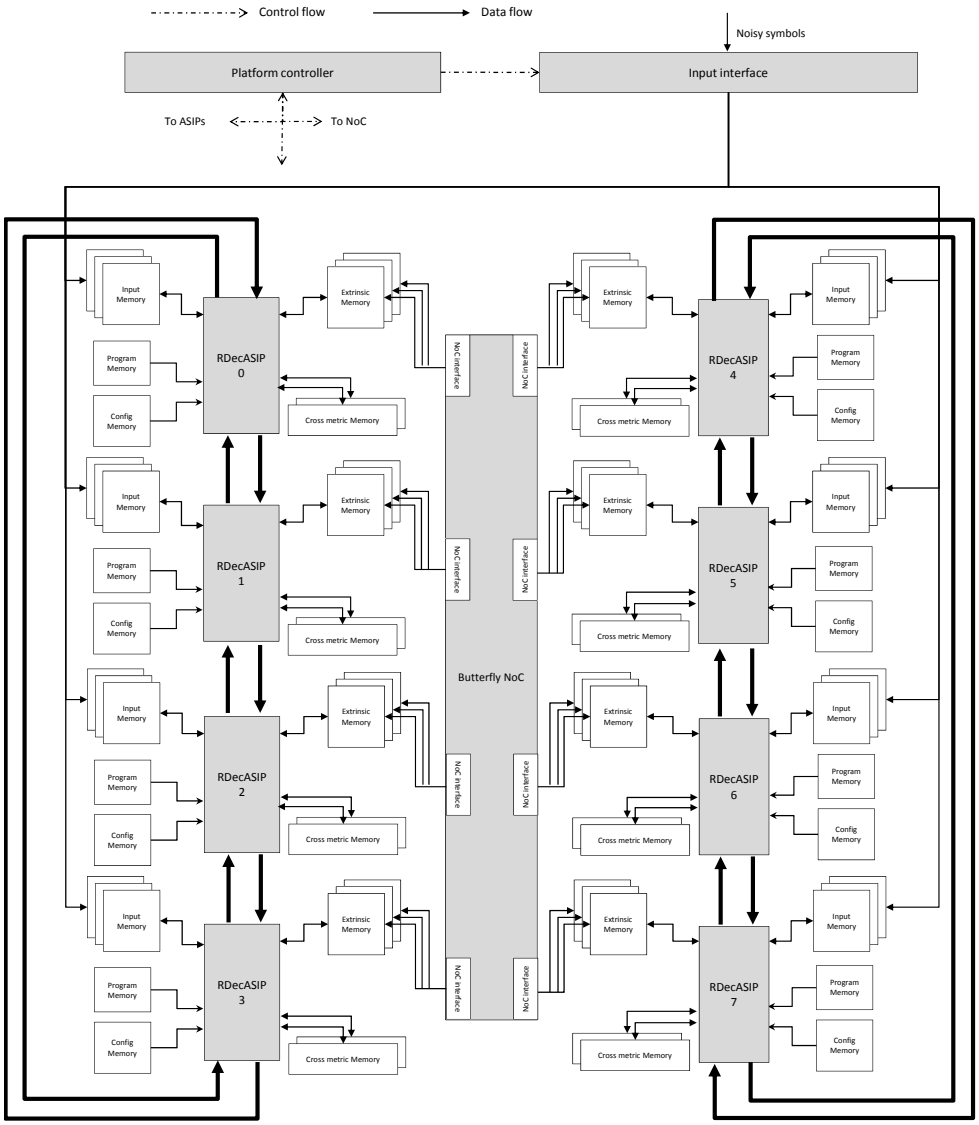


Figure 3.1: *UDec architecture implementing 8 RDecASIPs associated to a platform controller and an input interface*

In this section, we consider a UDec architecture implementing eight RDe-

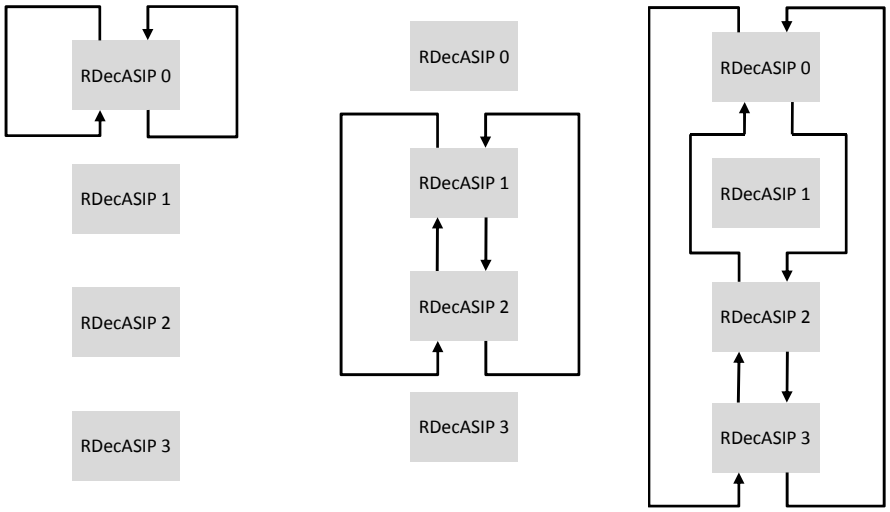
cASIPs associated with a *platform controller* and an *input interface* as shown in Figure 3.1. This specific number of RDecASIPs has been chosen in order to be able to illustrate all dynamic configuration issues that the UDec architecture has to face while keeping reasonable complexity for the clarity of the presentation. In fact, the contributions presented in this chapter remain valid for lower and higher number of RDecASIPs. In the following section, the possibility to adapt at run time the number and the location of the active RDecASIPs used for a given configuration is addressed. This is followed by the description of the new platform controller allowing the management of the dynamically reconfigurable UDec architecture.

3.1.1 ASIP number and location

In a multi-mode and multi-standard context, the requirements in terms of throughput and BER evolve dynamically. Thus, depending of these requirements, the number of activated ASIPs have to be adapted at run-time. Moreover, in order to deal with hot spot and potentially faulty cores management for the UDec architecture, the location of the activated ASIPs has to be dynamically defined. Obviously, this new flexibility impacts the different components of the architecture. In the initial UDec architecture, the number of ASIPs used for a given configuration was fixed at design time and was equal to the total number of implemented cores. Therefore, the two ring buses and the Butterfly topology NoC did not support a dynamic evolution of the number and location of the ASIPs selected for a given configuration.

3.1.1.1 Ring buses adaptation

The ring buses consist of direct connections between the ASIPs allowing to exchange boundary state metrics as shown in Figure 3.1. So, when the number and the location of the selected ASIPs dynamically evolve, the loop connections between the last and the first selected ASIPs have to be adapted. Figure 3.2 shows different examples of the ring buses adaptation for one component decoder. Figure 3.2(a) shows the case where only one ASIP is used in the component decoder while Figure 3.2(b) shows the case where two ASIPs are selected to perform the decoding task. Moreover, the location of the first ASIP has been shifted from *RDecASIP 0* to *RDecASIP 1*. Finally, Figure 3.2(c) shows the case where three ASIPs are selected and the location of the first ASIP has been shifted from *RDecASIP 0* to *RDecASIP 2*. In this case, the last ASIP of the component decoder becomes the *RDecASIP 0*. However, in order to reduce the design and the management complexity of this new flexibility feature, the selected ASIPs have to be physically consecutive in the component decoder as shown in the examples of



(a) 1 selected ASIP (b) 2 selected ASIPs (c) 3 selected ASIPs

Figure 3.2: *Ring buses dynamic adaptation examples*

Figure 3.2 except for the first and last selected ASIPs.

The configuration of the ring buses is done using two parameters. The number of selected ASIPs (*NumASIPs*) in each component decoder reflecting the level of sub-block parallelism and the shift value (*ASIPShift*) determining which ASIP is the first selected ASIP. Considering the configuration examples presented in Figure 3.2(a), $NumASIPs = 1$ and $ASIPShift = 0$ while $NumASIPs = 2$ and $ASIPShift = 1$ for the example of Figure 3.2(b). Based on these two parameters, a selection vector is computed. Each bit of this vector corresponds to one RDecASIP of the component decoder. The same *selection vector* is used for the two component decoders of the platform creating a symmetry. The selection vector drives multiplexers that determine the ring buses configuration.

3.1.1.2 Butterfly topology NoCs adaptation

The extrinsic information transfers through the NoC are also impacted when the location of the selected ASIPs changes dynamically. Indeed, the routing information for the transfer is computed by the network interface associated with each ASIP depending of a global address of the symbol generated by the ASIP. Figure 3.3 illustrates the routing principle for the considered Butterfly topology NoC. The Butterfly NoC is a multistage interconnection network with indirect topology: nodes at the ends and routers in the middle. This topology allows a unique path in the network between each pair of nodes (source to destination). For each

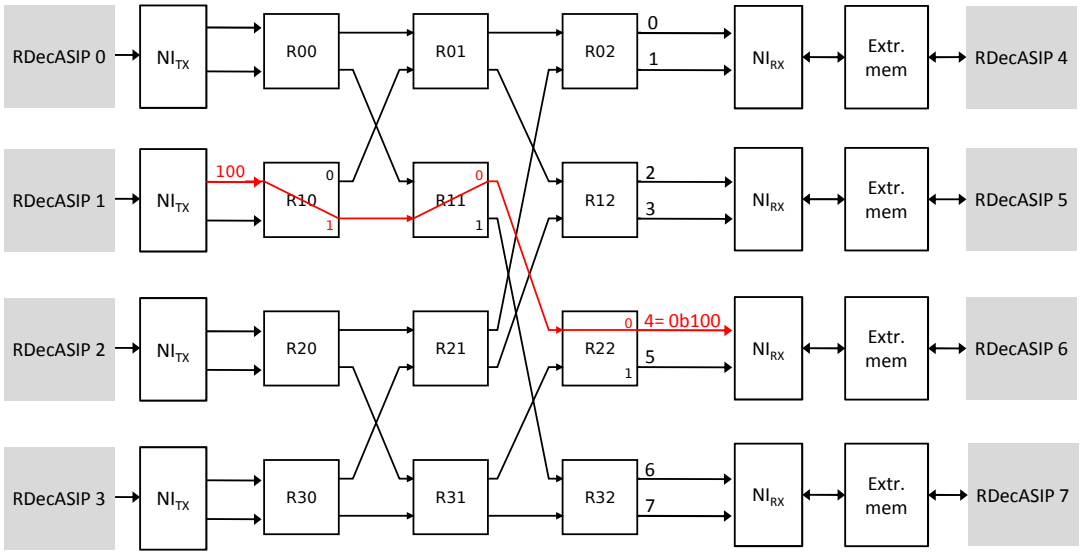


Figure 3.3: *Butterfly topology routing principle*

router, there are two inputs and two outputs. A single bit is used in a router to select the appropriate output: 0 for the first output, 1 for the second. Considering the UDec architecture implementing 8 RDecASIPs, the network has three stages of routers. The routing information is composed of three bits. Each bit indicates the output of the router of the different stages during the transportation. The information routing, due to this topology also directly indicates the number of the output of the network. In the example of Figure 3.3, the output number 4 is targeted. Thus the routing information is 1 for the first router, 0 for the second one and 0 for the last one corresponding to the binary value of 4.

In the initial UDec architecture, the level of sub-block parallelism is fixed at design-time since all the implemented ASIPs are used whatever is the configuration to be performed. Thus, the routing information calculation method was also fixed at design-time and was not reconfigurable. In the proposed architecture, the routing information has to be adapted depending of the location of the ASIPs determined by the *ASIPShift* value and the level of sub-block parallelism determined by the number of selected RDecASIPs for the configuration in each component decoder (*NumASIPs*). Indeed, depending of the interleaving rule of the standard, the level of sub-block parallelism and the frame size, the extrinsic information has to be sent to a specific ASIP of the other component decoder. Moreover, the *ASIPShift* value influences the location of the selected ASIPs, and consequently, it influences the location of the destination ASIPs. Therefore, this value has also to be taken into account.

	DBTC mode		SBTC mode	
	Routing info.	Local @	Routing info.	Local @
$0 \leq$ G@ $< \frac{FS}{2}$	00 & 0 and 1	G@	00 & 0 or 1	$\frac{G@}{2}$
$\frac{FS}{2} \leq$ G@ $< FS$	01 & 0 and 1	G@ - $\frac{FS}{2}$	01 & 0 or 1	$\frac{(G@-FS/2)}{2}$

Table 3.1: *Routing information for a 2 selected ASIPs configuration*

	DBTC mode		SBTC mode	
	Routing info.	Local @	Routing info.	Local @
$0 \leq$ G@ $< \frac{FS}{4}$	00 & 0 and 1	G@	00 & 0 or 1	$\frac{G@}{2}$
$\frac{FS}{4} \leq$ G@ $< \frac{FS}{2}$	01 & 0 and 1	G@ - $\frac{FS}{4}$	01 & 0 or 1	$\frac{G@-FS/4}{2}$
$\frac{FS}{2} \leq$ G@ $< \frac{3.FS}{4}$	10 & 0 and 1	G@ - $\frac{FS}{2}$	10 & 0 or 1	$\frac{G@-FS/2}{2}$
$\frac{3.FS}{4} \leq$ G@ $< FS$	11 & 0 and 1	G@ - $\frac{3.FS}{4}$	11 & 0 or 1	$\frac{G@-3.(FS/4)}{2}$

Table 3.2: *Routing information for a 4 selected ASIPs configuration*

The routing information is generated from a global address (G@) generated by the address generator implemented in the ASIP (Section 1.4.2). As an example, Tables 3.1 and 3.2 show the routing information corresponding to the configurations for 2 and 4 selected ASIPs respectively for both SBTC and DBTC modes. In these examples the *ASIPShift* value is equal to 0.

When two RDecASIPs are selected in each component decoder, the frame which has to be decoded is split and each RDecASIP is associated with half of the frame. Thus two cases are met as shown in Table 3.1. When the global address generated by the ASIP is lower than the frame size (FS) divided by two, the extrinsic information has to be sent to the first ASIP of the second component decoder while it has to be sent to the second ASIP when $G@$ is between $FS/2$ and FS. Considering the UDec architecture implementing eight RDecASIPs, i.e. four in each component decoder, three bits are necessary to route the information through three routers. As shown in Figure 3.3, the first destination ASIP is reached with a routing information equals to 0 or 1 (000 or 001 in binary) and the second destination ASIP is reached with a routing information equals to 2 or 3 (010 or 011 in binary). The last bit of the routing information depends on the Turbo code mode. In DBTC mode a message is split (Section 1.4.3), so one half is sent with the last stage route to 0 and the other half with the last stage route to 1. That is why the routing information is presented in the form "00 & 0 and 1" in Table 3.1.

In SBTC mode, the route of the last stage is determined by the LSB of the global address. That is why the routing information is presented in the form "00 & 0 or 1" in Table 3.1. If the global address is odd, the route of the last stage is 1 while it is 0 when the global address is even. Table 3.2 shows the routing information for 4 selected RDecASIPs. In this case, the frame is divided in four parts and each RDecASIP is associated with a quarter of the frame. Depending on the global address value, the routing information is sent to the first (000 or 001), the second (010 or 011), the third (100 or 101) or the last ASIP (110 or 111).

The extrinsic information has to be stored in the extrinsic memory associated with each RDecASIP. Thus a local address (Local @) is computed depending of the global address computed by the emitter RDecASIP and the level of sub-block parallelism as shown in Tables 3.1 and 3.2. In DBTC mode, one complete extrinsic information for one symbol is stored at each memory address. Thus depending on the level of sub-block parallelism, the extrinsic information is spread over the extrinsic memories of each RDecASIP from the local address 0 to $\frac{FS}{NumASIPs}$ where $NumASIPs$ is the number of the selected RDecASIPs in each component decoder. In SBTC mode, extrinsic information for two symbols is stored at each memory address. Thus depending of the level of sub-block parallelism, the extrinsic information is spread over the extrinsic memories of each RDecASIP from the local address 0 to $\frac{FS/NumASIPs}{2}$. The $\frac{FS}{NumASIPs}$ and $\frac{FS/NumASIPs}{2}$ can be computed using right shifting and addition operators when $NumASIPs$ is a power of two. Else, a divider has to be implemented.

In the example of configurations presented in Tables 3.1 and 3.2, the *ASIPShift* value is equal to 0. However when this value changes, the routing information has to be modified also. In order to reduce the complexity of the design, the same *ASIPShift* value is considered in both component decoders. Thus, considering the UDec architecture implementing eight RDecASIPs,when the RDecASIP 0 is selected the RDecASIP 4 is selected too, and similarly for the pairs (RDecASIP 1 and RDecASIP 5), (RDecASIP 2 and RDecASIP 6), and (RDecASIP 3 and RDecASIP 7). When the *ASIPShift* value differs from 0, the routing information can be computed as shown in Figure 3.4. To obtain the final routing information taking into account the *ASIPShift* value, the initial routing information is first added with $2 \times ASIPShift$ (since each router has two output ports) and then, a modulo operator is used in order to ensure a loop-like configuration where RDecASIP 0 and RDecASIP 4 can logically follow the RDecASIP 3 and RDecASIP 7 respectively.

The optimizations presented in this section provide the support for ASIP number and location run-time adaptation at the ring-buses and network-on-chip

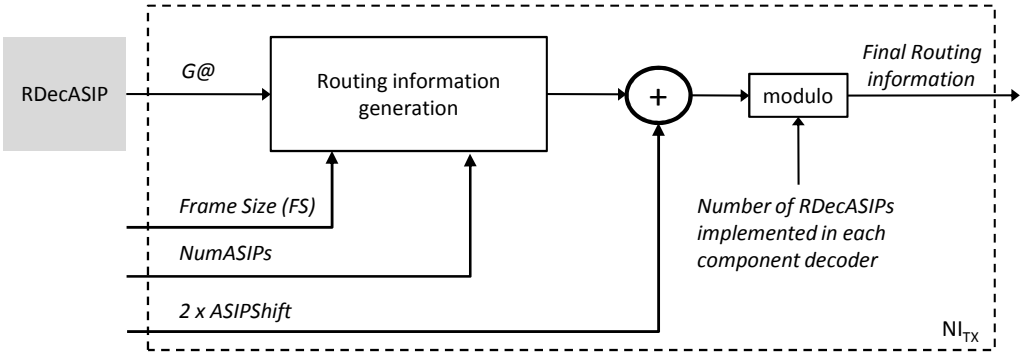


Figure 3.4: *Complete routing information generator*

levels. The next section focuses on the control of the platform through optimizations implemented in the platform controller.

3.1.2 Platform controller

Due to the new flexibility introduced in the previous section and the need for a flexible management of the platform, optimizations have to be implemented on the UDec platform controller. The initial UDec platform controller was dedicated to an architecture implementing eight ASIPs which are all activated whatever the configuration.

In the initial version of the UDec architecture, the components of the platform are reseted after each decoding process. In order to provide the possibility to decode consecutive frames associated with the same configuration without any initialization step before each data frame, we assume that a *Configuration Manager* provides configuration order to the UDec platform controller. Figure 3.5 shows the Finite State Machine (FSM) associated with the platform controller.

Following a reset of the platform or the end of the decoding process, the first step consists in waiting for the Configuration Manager order. When the configuration is already loaded into the platform, the controller checks that the input symbols that have to be decoded are ready and the decoding process is launched depending of the chosen decoding mode given by the configuration parameters. When a new configuration has to be performed, a *platform initialization* step is performed to read and to spread the configuration parameters to the different platform components such as the network interfaces of the Butterfly topology NoCs or the ring buses configuration interface. Once the platform is initialized, the corresponding decoding process is performed i.e. Serial or Shuffled decoding depending of the configuration parameters.

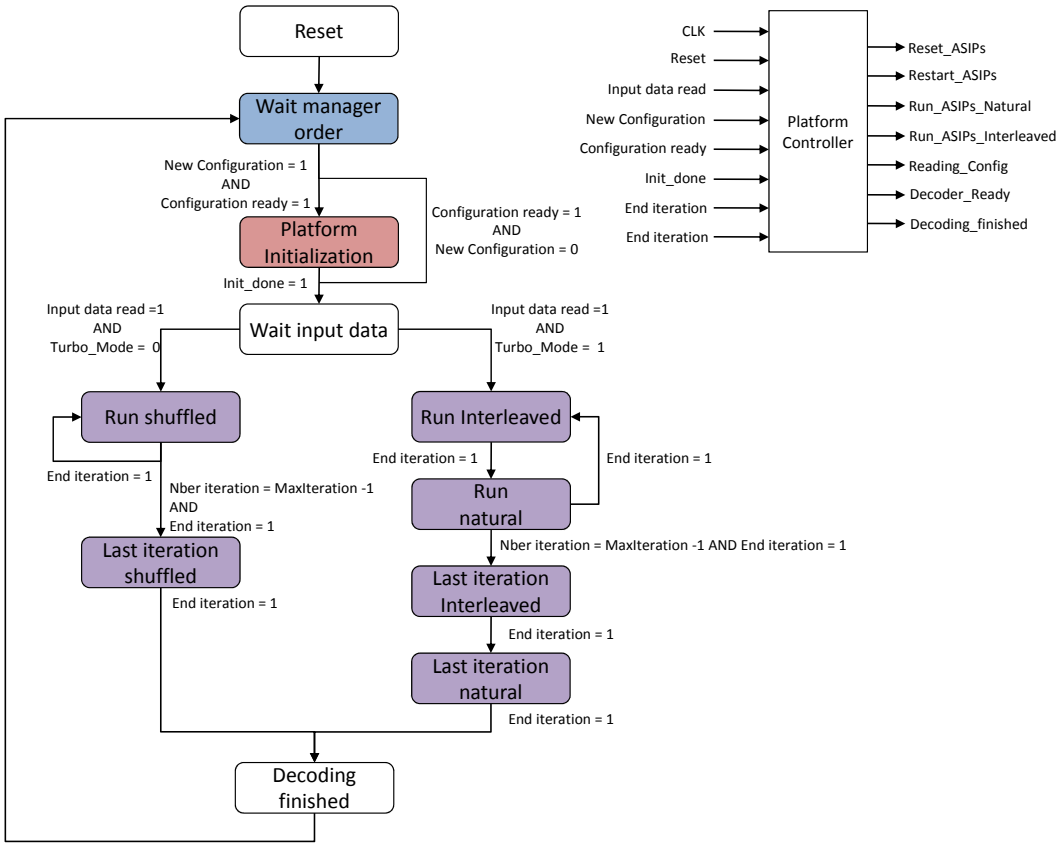


Figure 3.5: *Flexible UDec platform controller FSM*

The communication with the configuration manager is performed through dedicated inputs and outputs control signals. Two 1-bit input signals are used to inform the controller about the configuration status. The *New configuration* signal indicates that a new configuration has to be executed when its value is 1. The *Configuration ready* signal indicates that the configuration is ready to be executed when its value is 1. When both *New configuration* and *Configuration ready* signals values are 0, the controller waits for a new order. When *Configuration ready* value is 1 and *New configuration* value is 0, the current configuration is re-executed while a new configuration is executed when both *New configuration* and *Configuration ready* signals values are 1. Finally, when *Configuration ready* value is 0 and *New configuration* value is 1, the controller considers that the configuration is not loaded yet and waits for the *Configuration ready* signal to be 1.

Platform initialization is performed through a specific configuration mem-

ory presented in Table 3.3. This configuration contains some parameters identical to the RDecASIP processor such as the seeds and steps which are used by the input interface to spread the input symbols over the input memories of each activated ASIP depending of the interleaving law of the selected standard. Compared to the RDecASIP configuration memory presented in Section 2.2.2, four parameters have been introduced.

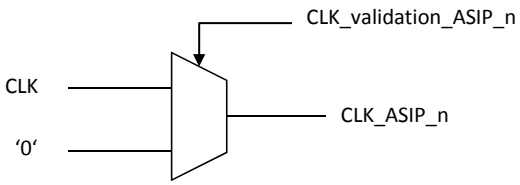
Two of these parameters (*NumASIPs ASIPShift*) are used to manage the number and the location of the activated ASIPs and were presented in the previous section. The *ConfigSize* and *Config@* parameters are related to the RDecASIP initialization step. Indeed, as presented in Section 2.2.4, the configuration memory of each ASIP is able to store several configurations. Thus, in order to address a particular configuration the *Config@* parameter is used to determine the four MSB bits of the memory address where the configuration is stored. The *ConfigSize* parameter determines the size of the configuration that has to be read by the ASIP. This parameter has been introduced for future evolution of the RDecASIP processor providing flexible configuration size.

The RDecASIP processors are launched by enabling the clock signal associated to the ASIP. In the original version of the UDec platform, a particular control signal associated to each ASIP (*CLK_validation_ASIP_n*) was driven by the platform controller as shown in Figure 3.6(a). In order to support the ASIP number and location flexibility, the platform controller generates two clock validation signals corresponding to each component decoder. The local ASIP clock validation signal is then computed depending of the number of activated signals and the *ShiftASIP* parameter value determining the location of the activated ASIPs as shown in Figure 3.6(b). For this purpose the *Selection vector* presented in Section 3.1.1 determining if the ASIP is selected for the configuration and an additional AND logic gate are used.

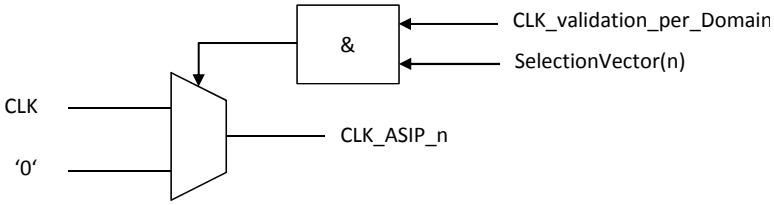
The last two sections have introduced the new flexibility implemented in the UDec architecture to propose a better dynamic conguration management . However, the current UDec architecture is not associated to a configuration infrastructure allowing the configuration information transfer to the different configuration memories of the platform. The next section presents a novel configuration infrastructure optimized for high speed configuration information transfer.

@	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	ConfigSize						Config@						Maxiteration						State				NumSteps			
1	-	NumASIP			ShiftASIP		TurboInitIteration				Frame size in bits												Mode			
2	Turbo Seed 0																		Turbo Seed 1							
3	Turbo Step 0																		Turbo Step 1							
4	Turbo Step 2																		Turbo Step 3							
5	Turbo Step 4																		Turbo Step 5							
6	Turbo Step 6																		Turbo Step 7							

Table 3.3: *UDec platform controller configuration memory*



(a) Initial ASIP clock validation



(b) New ASIP clock validation

Figure 3.6: *Initial and new ASIP clock validation*

3.2 UDec configuration infrastructure

The behavior of the UDec turbo decoder is dependent on the parameters loaded in the configuration memory of each ASIP. The configuration memory presented in Section 2.2.2 has been introduced in order to improve the reconfiguration process. The configuration parameters stored in the configuration memory (Table 2.5) are divided in three categories as follows: (1) parameters can be different for each ASIP, from address @0 to @1; (2) parameters are domain dependent, from address @2 to @6; (3) parameters are the same for all ASIPs, from address @7 to @10. This organization allows an efficient and fast reconfiguration at the platform level. Indeed, multicast and broadcast mechanisms can be used to load the configuration in order to minimize the data transfer load. In this context, two multicast transfers are necessary to send domain dependent parameters to corresponding ASIPs and one broadcast transfer for parameters that are the same for all ASIPs. Finally, unicast transfers are used to load the ASIP dependent parameters. Initially, the UDec architecture is not associated with any configuration infrastructure providing the possibility to transfer new configurations at run-time. The next section highlights the challenges which have to be addressed in order to propose an efficient dynamic reconfiguration of the UDec architecture.

3.2.1 Main challenges for an efficient configuration infrastructure

Dynamic reconfiguration of flexible multiprocessor platforms is one of the main challenges for system designers. This issue is important, especially when dealing with multiprocessor platforms where no global communication interconnect can be shared between application and configuration data. In this case, it is mandatory to implement a communication infrastructure dedicated to configuration data. In the UDec architecture, the 88-bit ring buses and the Butterfly NoC are optimized and dedicated to data exchange between RDecASIPs. These interconnections can not be used to transfer the configuration data. To build an efficient solution, the *configuration infrastructure* has to take into account the following requirements:

1. Low complexity
2. Multicast, broadcast and selection mechanisms
3. Incremental burst transfer

3.2.1.1 Low complexity

The configuration infrastructure only manages configuration memories updates. Therefore, this extra hardware must have a minimal impact on the global design complexity in terms of area overhead. When designing a communication architecture for a multiprocessor platform, two main technologies are available: Network on Chip or On-Chip Bus. Last decade has seen the huge adoption of Network on Chip in complex System on Chip to mainly enhance the throughput and the scalability compared to a bus-based communication infrastructure. However, the design of a communication interconnect dedicated to configuration data does not require such a complex approach. Indeed, broadcasting can be defined as a unidirectional communication between a reconfiguration manager that generates and downloads configuration data to one or a group of processing elements that have to be configured. Hence, there is no transfer concurrency issue, and a unique component, called *Master*, is able to initiate a transfer. Moreover, we assume that the configuration manager is, at any time, aware of the configuration state of the platform. Thus, no read operations are necessary to check a status or the presence of a particular configuration in a given processing element. These features lead to a bus-based structure that provides a simple and low complexity communication interconnect for this particular context.

3.2.1.2 Multicast, broadcast and selection mechanisms

The UDec platform is configured through the RDecASIP configuration memories. As shown in Section 2.2.2, the RDecASIP configuration memory is organized in order to allow multicast and broadcast mechanisms for an efficient and fast configuration of the multi-ASIP platform. Moreover, depending on the application requirements, the number of activated RDecASIPs to perform a given configuration can be tuned at run-time. Hence, a mechanism of processor selection has to be introduced in order to send configuration data to activated RDecASIPs only.

3.2.1.3 Incremental data burst transfer

The last point to build an efficient configuration infrastructure for the UDec platform is related to the transfer mode. Since some of the configuration data has to be loaded in adjacent parts in the configuration memory, all related transfers can be defined as a burst starting from a *base address* in the configuration memory. For example, based on the RDecASIP configuration memory organization of Table 2.5, configuration data identical for all RDecASIPs can be incrementally transfered starting from the base address @7.

Many On-Chip Buses have been developed these last years that propose different topologies and different communication protocols. Table 3.4 proposes a comparison of the most representative On-Chip-Buses of the literature. Representative On-chip Buses are the AMBA [57], the CoreConnect [58] or the Avalon bus [59]. Unfortunately, these solutions do not support multicast. Work presented in [60] supports multicast but this solution implements complex arbitration mechanisms and communication protocols that are not necessary in our context. The Fast Simplex Link (FSL) [61] proposes a low complexity unidirectional bus for data transfer. Unfortunately multicast is not supported. It is thus mandatory to propose an optimized bus dedicated to configuration data that could be used for the UDec platform in order to reach the configuration latency challenge.

3.2.2 Configuration infrastructure

To address the main requirements highlighted in the previous section, we propose a new bus-based communication infrastructure as well as the associated communication protocol. Our goal is to optimize configuration data transfers into RDecASIPs configuration memories for the UDec platform. In this section, we detail the proposed architecture, dynamic selection of activated RDecASIPs, and communication protocol.

Challenge	AMBA [57]	CoreConnect [58]	Avalon [59]	SiliconBackplane [60]	FSL [61]	This work
Unidirectional (or 1 master)	√	√	√	√	√	√
Multicast	X	X	X	√	X	√
Broadacast	X	X	X	√	X	√
Incremental burst	√	√	√	√	X	√
Low complexity	X	X	X	X	√	√

Table 3.4: *SoA Buses comparison*

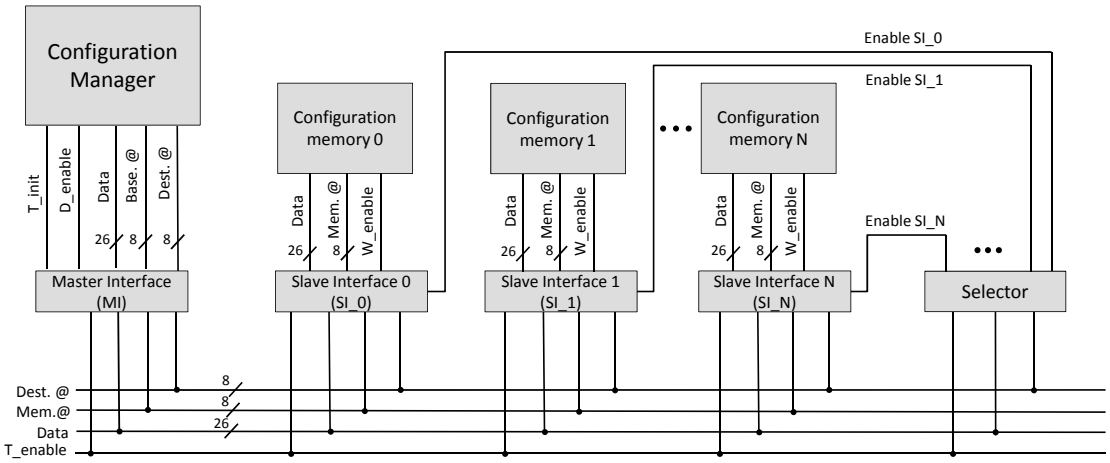


Figure 3.7: *Architecture of the proposed bus interconnect*

3.2.2.1 Architecture overview

The proposed bus architecture is presented in Figure 3.7. This architecture can be split in four functional blocks: *Master Interface* (MI), *Slave Interface* (SI), *Selector* and *interconnect*. Each configuration memory is connected to the bus through a SI. The configuration manager deals with the configuration generation which is based on internal decisions and external information and commands (this point is not addressed in this work).

The MI provides an interface allowing the connection of the configuration manager to the bus. To initiate a transfer, the MI receives, from the configuration manager, the address of a SI or a group of SIs (called *Destination address*) and the memory base address where the transfer starts. During a transfer, the MI also manages the increment of the memory address based on the base address. An overview of the MI architecture is shown in Figure 3.8. It consists of a com-

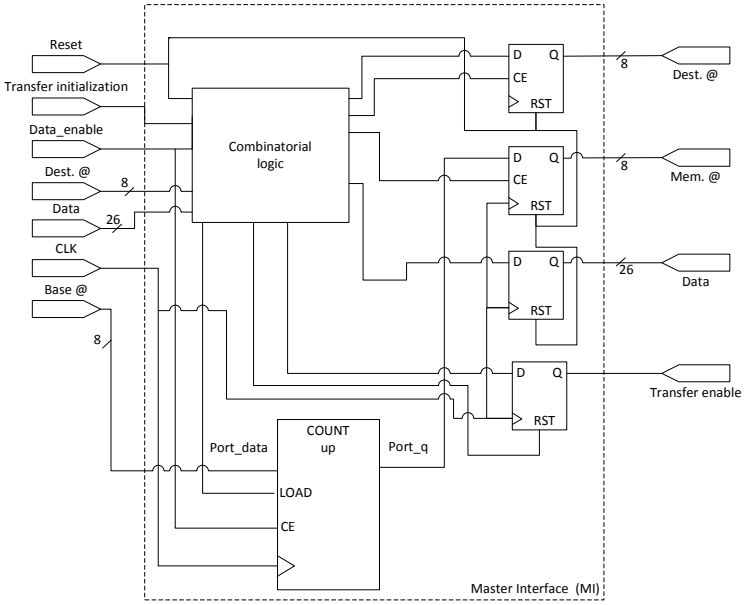


Figure 3.8: *Master interface architecture overview*

binatorial logic block for control signals generation, an 8-bit counter for memory address generation implementing a $LOAD$ function for counter initialization, and several Flip-Flops for clock synchronization.

The SI provides an interface between the bus and the configuration memory. Its role is, when a transfer is enabled, to check if the destination address corresponds to its own address or one of its associated multicast addresses. Then, the SI retrieves the data (and the associated memory address) from the bus and writes it into the configuration memory. An overview of the SI architecture is shown in Figure 3.9. It mainly consists of a 8-bit comparator used to compare the received destination address with the SI's addresses, a combinatorial logic block for control signals generation and a row of Flip-Flops for clock synchronization.

The Selector provides a simple and efficient solution to select, at run-time, RDecASIPs that are targeted by the next configuration data. For this purpose, each SI has a 1-bit input that is driven by the Selector. When this input is enabled, the associated SI is activated and reacts to the events on the bus while it ignores all transfers in the other case. An overview of the Selector architecture is shown in Figure 3.10. It mainly consists of a 8-bit comparator used to compare the received destination address with the Selector's address, a row of Flip-Flops

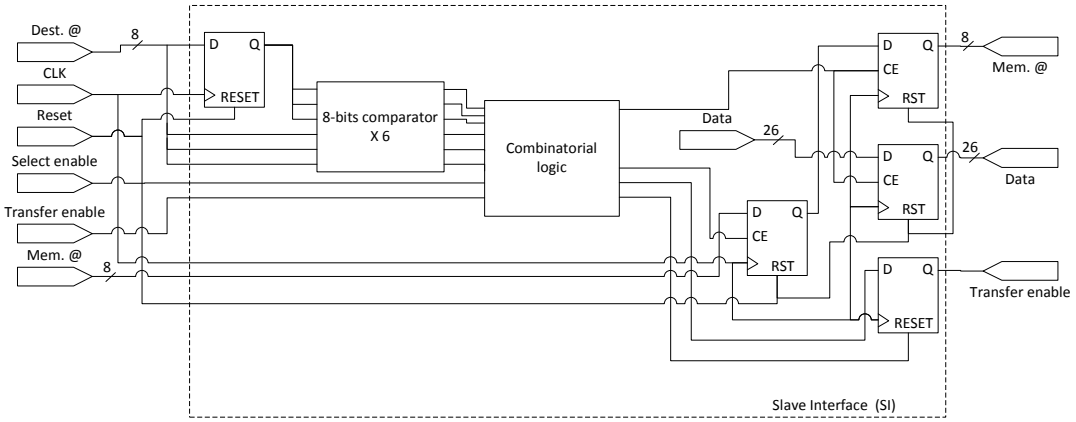


Figure 3.9: *Slave interface architecture overview*

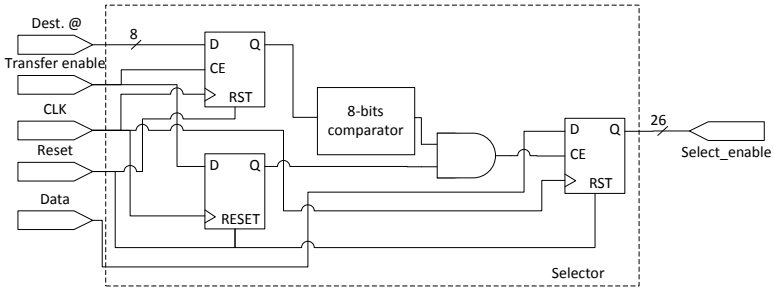


Figure 3.10: *Selector architecture overview*

for clock synchronization and an AND logic gate for control signal generation.

The interconnect part of the proposed architecture consists of three buses and a transfer enable control signal. Two address buses are required. The first one (*Dest.@* in Figure 3.7) is used to select the destination (i.e. one SI or a group of SIs) and the second (*Mem.@*) is used to indicate the target memory address. The third bus is used to send the configuration data. Finally, a control signal (*T_enable*) is used to inform SIs that a transfer has been enabled.

3.2.2.2 Addressing

Due to the context of this work which deals with a platform that consists of several implementations of specialized processing elements, and to minimize the

design complexity, we choose to statically, at design-time, define the SIs addresses and the different multicast addresses associated with each SI. Indeed, the interest of a run-time address definition is reduced by the fact that the RDecASIPs are statically grouped depending on the domain in which they perform the processing. Each SI connected to the interconnect block owns an unique address. This address allows configuration information transfers to a particular configuration memory only. Moreover, each SI is associated to a multicast address. This address is common for SIs associated with RDecASIPs in the same domain. Finally, all the SIs are associated to a single broadcast address.

3.2.2.3 Transfer protocol

The transfer of configuration data can be divided in three steps: 1) initialization and data transfer from the configuration manager to the MI, 2) data transfer from the MI to one or several SIs and 3) configuration memory loading from the SI.

From the configuration manager to the MI (upper part of Figure 3.11)

During the initialization step, the configuration manager sends the destination address and the base memory address to the MI. The *T_init* control signal (Figure 3.7) is driven to indicate to the MI that a transfer initialization is required. On the MI side, when these two addresses are read, the first one is stored and the second one is used to initialize the memory address increment process. These addresses are used until a new transfer initialization step is performed. After the initialization step, the configuration manager can send one data per cycle on the *Data* bus. The *D_enable* control signal is also driven at the same time to inform the MI that a data is available. Obviously, the data transfer can be suspended if no data is available. Figure 3.11 shows an example of transfer initialization and data transfer between the configuration manager and the MI.

From the MI to the SI(s)(middle part of Figure 3.11)

Figure 3.11 presents two examples of data transfer on the bus. The first one shows the transfer of a single data, and the second one shows a data burst. The transfer on the bus consists of two phases: address phase and data phase. The address phase lasts for a single clock cycle. During this cycle, the destination and the base memory addresses are sent on the corresponding bus. The *T_enable* control signal is also driven to indicate that a transfer occurs. During the data phase, the data is sent on the *Data* interconnect. When a data burst is performed, a data is available at each clock cycle. The destination address is maintained on the bus during the transfer procedure while, for each data, the memory address is incremented by the MI.

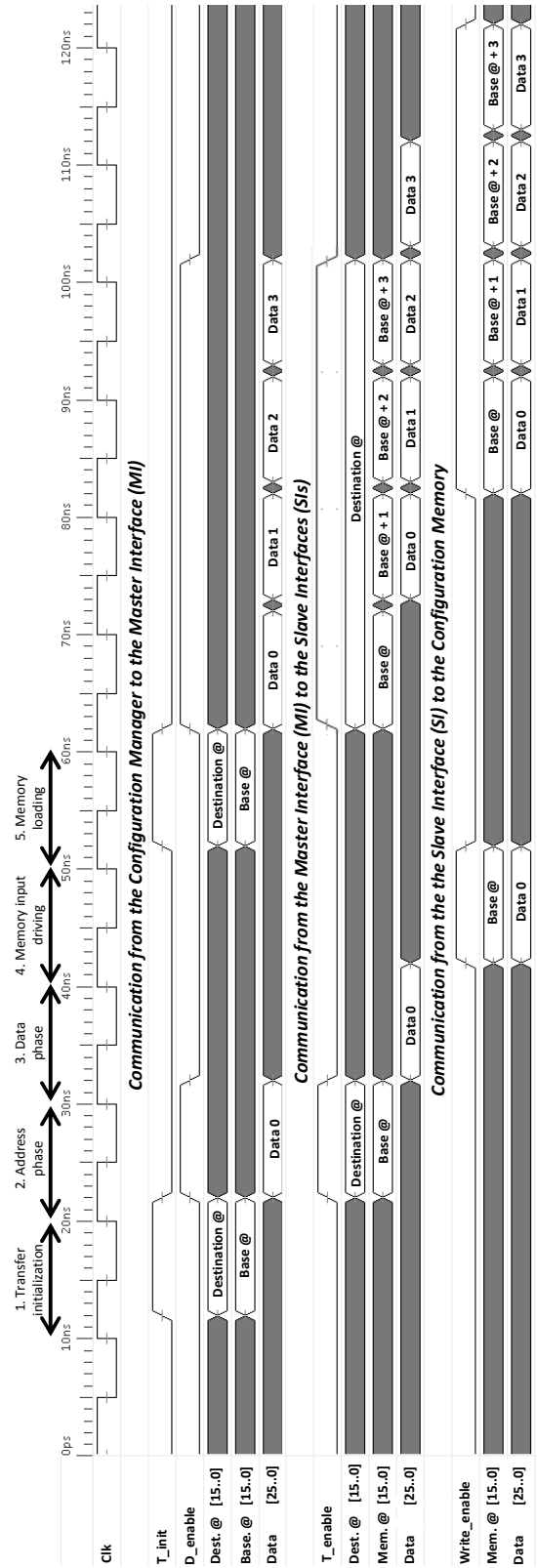


Figure 3.11: *Communication from the configuration manager to the configuration memory through the communication infrastructure*

From the SI to the configuration memory (lower part of Figure 3.11)
When a transfer occurs, the SIs involved in the transfer store the memory address (read during the address phase) and get the data on the next clock cycle. To write into the configuration memory, the memory address is stored during one clock cycle. When the data is available, the control signal *write_enable* of the memory is driven and the memory address and the data are sent on the interconnect between the SI and the configuration memory.

These three steps allow the transfer of a data into the configuration memory in 5 clock cycles while 11 clock cycles and 6 clock cycles are necessary through the CoreConnect PLB4 [58] and the AMBA AXI4 [57] buses respectively. Moreover, thanks to the pipeline nature of the transfers, the configuration infrastructure is able to provide one data per clock cycle to the destination.

3.2.2.4 Selection

The proposed optimizations to the UDec platform allow to dynamically select the number of RDecASIPs involved in the decoding process depending on the requirements of an application (e.g. throughput, error rate, etc.). When a configuration command occurs, a selection mechanism is launched to select the SIs associated to the configuration memories connected to the RDecASIPs involved in the next configuration. When an SI is not selected, it ignores all transfers on the bus. The Selector is configured through the bus infrastructure by the configuration manager which sends a configuration vector on the bus. Each bit of this vector corresponds to the state of one SI. This solution allows a fast selection of SIs compared to a mechanism in which each SI is selected through a unicast transfer. To reduce the complexity of the Selector, one Selector is associated with a number of SIs corresponding to the width of the data interconnect (that determines the width of the configuration vector). For the UDec platform 26 SIs can be associated to one Selector. Depending on the number of SIs, several Selectors can be distributed along the bus infrastructure.

Since the selection is performed through a transfer on the bus, the SIs targeted by the selection process are ready to receive configuration data after 5 cycles (Figure 3.11). However, taking into account the pipeline nature of the bus, a data transfer can be initialized by the configuration manager with a delay of one clock cycle after the selection data has been sent to the MI. This delay is sufficient to guarantee that the targeted SIs are ready when the first configuration data arrives.

This section has detailed the configuration infrastructure highlighting main features and providing an analysis of the latency. Next sections focus on the two-step validation process. First a SystemC/VHDL mixed model is described,

then an FPGA prototype is detailed and logic synthesis results targeting an ASIC implementation are given.

3.2.3 SystemC/VHDL mixed Validation

To validate our contribution we propose to analyze the configuration infrastructure through a mixed SystemC/VHDL simulation model based on the UDec architecture implementing 4 RDecASIPs. For this purpose, a complete Cycle Accurate and Bit Accurate (CABA) SystemC [62] model of the proposed bus architecture was implemented. Such an approach has been selected in order to provide more flexibility all along the prototyping process. This model was connected to a VHDL model of the UDec platform to allow a mixed SystemC/VHDL simulation. Finally, A non-timed SystemC model was developed to simulate the configuration manager (which is in charge of the configuration generation based on parameters set by the designer). The goal of this mixed model was to validate and to provide early evaluation of the proposed configuration infrastructure architecture through realistic configuration scenarios.

3.2.3.1 Platform model

Figure 3.12 presents the different components implemented in the platform model. On the VHDL model side, A *Random generator* associated with an *Emitter* produce the encoded symbols. The channel model is an Additive White Gaussian Noise (AWGN). The *Input interface* distributes the received symbols in the *Input memories* of each RDecASIP. A verification module compares original symbols with decoded symbols to evaluate the decoding performance. Finally, A controller is used to manage the platform. In Figure 3.12, data and control flow are highlighted with full black and broken lines respectively.

On the SystemC model side, 1 Selector and 5 SIs are implemented. Each SI is connected to a configuration memory: 4 RDecASIP configuration memories (see Section 2.2.2) and one configuration memory associated with the controller of the platform. This last memory is used to configure the modules of the platform (see Section 3.1.2). For clarity reasons, connections between the Selector and the SIs are not represented in Figure 3.12. The SystemC model of the Configuration Manager allows an interaction with the designer in order to simulate complete dynamic reconfiguration scenarios. In Figure 3.12, Configuration data flow is shown in doted lines.

3.2.3.2 Model evaluation

The proposed SystemC/VHDL mixed model allows a fast evaluation of the configuration performance with respect to the decoding time for realistic configuration

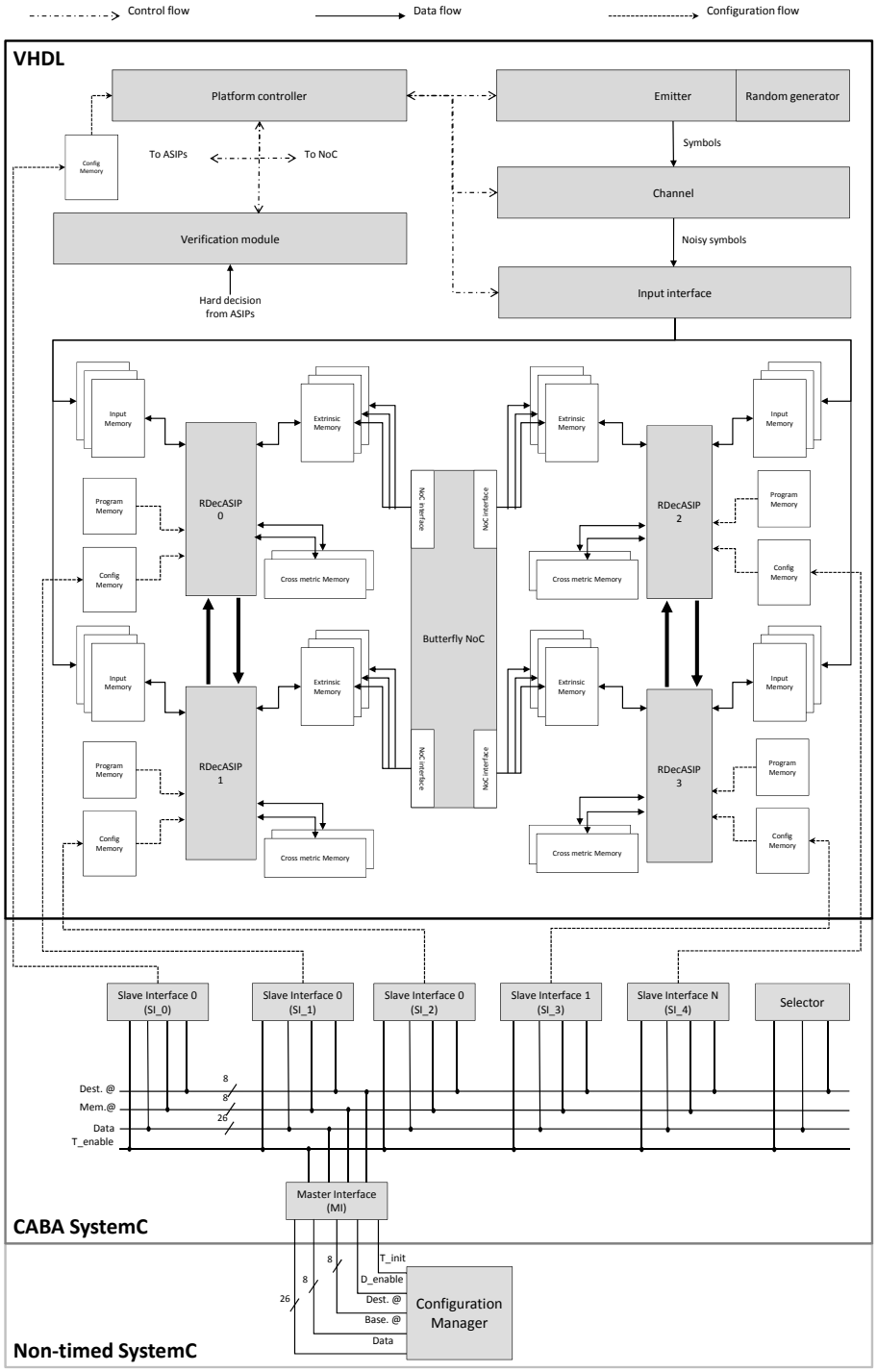


Figure 3.12: *Architecture model for SystemC/VHDL mixed simulation*

Standard	frame size (in bits)	Config. time (in cycles)	Dec. time (in cycles)	Config. time ratio (in %)
DVB-RCS	440	43	4528	0.95
DVB-RCS	1728	43	25548	0.17
LTE	440	43	4751	0.91
LTE	1728	43	26455	0.16
LTE	3008	43	45215	0.095

Table 3.5: *Configuration loading impact for 4 active ASIPs*

scenarios generated at run time by the configuration manager. The configuration latency of the UDec platform using the proposed approach is defined by Equation (3.1).

$$ConfigurationLatency = \frac{31 + (3.N_{ASIP})}{F} \quad (3.1)$$

Where N_{ASIP} is the number of RDecASIPs and F is the frequency of the proposed bus architecture. 31 clock cycles are necessary to transfer the parameters common to all ASIPs and the parameters common to ASIPs of the same decoder component. 3 additional clock cycles are necessary to transfer parameters that are different for each ASIP.

Table 3.5 shows, for DVB-RCS and LTE standards and for different data frame sizes, the ratio of the configuration loading regarding the decoding time for 4 active ASIPs and 6 decoding iterations. The last column represents the configuration time when compared to the total time required to configure and decode a frame. The configuration time is constant since the number of ASIPs is fixed. However, when the data frame size increases, the decoding time increases too. Consequently, the configuration time impact becomes negligible with high frame size (0.095% for a 3008-bit frame in 3GPP-LTE mode). And even with smaller frame size, the impact of configuration loading is still low (less than 1% for a 440-bit frame size).

The SystemC/VHDL mixed model has enabled an early and fast validation of the main features described in Section 3.2.2. Starting from this model, next section presents the first hardware implementation on FPGA and an evaluation for an ASIC target technology.

3.2.4 FPGA prototype

To validate the proposed configuration architecture and the communication protocol presented in Section 3.2.2, a hardware prototype has been developed on a

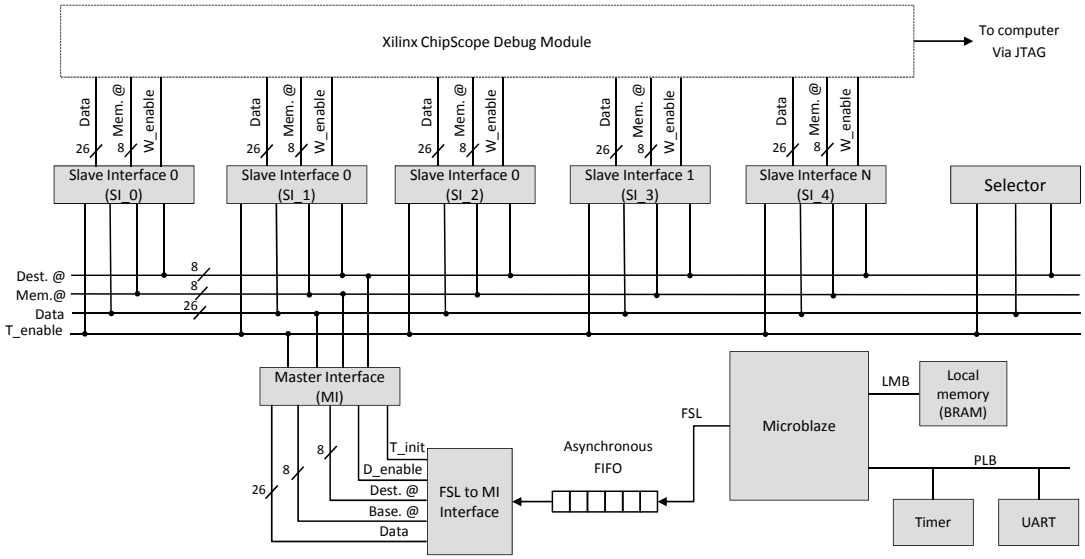


Figure 3.13: *Architecture of the prototype*

Xilinx XUPV5 platform implementing a Virtex 5 LX110T FPGA for CoreConnect PLB4 comparison and on a Digilent Atlys board implementing a Spartan-6 XC6SLX45 FPGA for AMBA AXI4 comparison.

The prototype architecture is shown in Figure 3.13. It consists of a Xilinx Microblaze soft core that generates the configuration at run-time. The configuration is then sent through an FSL bus to an FSL to MI interface. The FSL [61] connection has been considered as this interconnect structure proposes a fast, simple and unidirectional connection for the Microblaze which ease its integration within the configuration infrastructure (other interconnection solution to the Microblaze would lead to a higher complexity and lower performance). An asynchronous FIFO is associated to the FSL connection in order to provide frequency domain flexibility on both Microblaze and configuration infrastructure sides. The FSL to MI interface realizes the protocol adaptation between the FSL communication protocol and our bus protocol. Finally, the outputs of each SI are connected to a Xilinx ChipScope module that allows the run-time monitoring of the related signals. This module replaces the configuration memories associated with the SIs.

The FSL bus provides a solution to transfer, each cycle, a 32-bit width data. Hence, we define an adaptation protocol in order to, starting from a 32-bit frame, extract control and data information. Figure 3.14 shows the two FSL frame models used to adapt the communication protocol. Depending on the control flag, the FSL to MI interface provides a transfer initialization or a data transfer service on the bus. Figure 3.14(a) shows the FSL frame model sent by the Microblaze

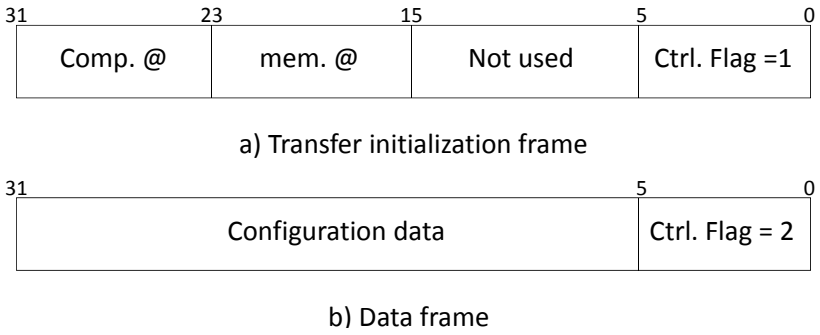


Figure 3.14: *FSL to MI protocol adaptation*

to initialize a transfer on the bus. In this case the control flag is set to 1, the flag indicates that this frame corresponds to a transfer initialization. The 8-bit component destination and the 8-bit base memory address are extracted. One cycle later, the interface drives MI's inputs as described in Section 3.2.2.3. Figure 3.14(b) shows the FSL frame model sent by the Microblaze to transfer data after a transfer initialization. In this case the control flag is set to 2, the flag indicates that this frame corresponds to a data transfer. The 26-bit data is extracted from the frame and the interface drives MI's inputs as described in Section 3.2.2.3.

Using the proposed hardware implementation, configuration transfer time were evaluated for several numbers of RDecASIPs. For this purpose, the Microblaze and the proposed bus frequency is set to 125 MHz. The ChipScope module is configured to monitor the output signals of the SIs. Table 3.6 shows the configuration transfer times of the bus compared to designs we have implemented using CoreConnect PLB4 [58] and AMBA AXI4 [57] buses connected to a Microblaze with the same clock frequency (set up to 125MHz). Thanks to the multicast mechanisms, a low overhead of 144 ns is necessary to configure each additional couple of RDecASIPs (one ASIP in both natural and interleaved domains) while 1936 ns and 1056 ns are necessary for [58] and [57] respectively. Results of Table 3.6 illustrate that the proposed implementation significantly reduces the configuration time overhead when the number of active RDecASIPs increases compared to classical bus approaches.

Compared to Equation (3.1), the performance of the proposed configuration infrastructure FPGA prototype is defined by Equation (3.2).

$$ConfigurationLatencyFPGA = \frac{93 + (9.N_{ASIP})}{F_{Microblaze}} \quad (3.2)$$

Where N_{ASIP} is the number of RDecASIPs and $F_{Microblaze}$ is the frequency of the Microblaze and the proposed bus. 93 clock cycles are necessary to transfer the parameters common to all ASIPs and the parameters common to ASIPs of

Nb. ASIPs	Transfer latency (in ns)			Speedup	
	This work	CoreConnect [58]	AMBA [57]	vs. [58]	vs. [57]
4	1 032	3 872	2 212	3.75	2.14
6	1 176	5 808	3 168	4.94	2.69
8	1 320	7 744	4 224	5.87	3.2
16	1 896	15 488	8 448	8.17	4.45
32	3 048	30 976	16 896	10.16	5.54
64	5 352	61 952	33 792	11.57	6.31
128	9 960	123 904	67 584	12.44	6.78

Table 3.6: *Configuration transfer time in ns*

Infrastructure Component	CoreConnect [58] Virtex 5		AMBA [57] Spartan 6		This work Virtex 5		This work Spartan 6	
	FF	LUT	FF	LUT	FF	LUT	FF	LUT
Master Interface	143	286	18	6	85	14	81	13
Slave Interface	123	67	207	355	52	6	49	4
Selector	-	-	-	-	35	2	34	2
Total	266	353	225	361	172	22	164	19

Table 3.7: *FPGA synthesis results comparison*

the same decoder component. 9 additional clock cycles are necessary to transfer parameters that are different for each ASIP. Compared to Equation (3.1), the performance penalty is due to two factors. Indeed, the FSL bus used to connect the Microblaze processor to the configuration infrastructure and the protocol adaptation between the FSL bus and the MI lead to additional cycles which impact the transfer latency. Moreover, the Microblaze does not send one data per cycle on the FSL bus since several cycles are necessary to build the frames presented in Figure 3.14 before each transfer. However, as shown in Table 3.6, using this solution combined with our low latency configuration infrastructure allows us to perform the configuration of up to 128 ASIPs in less than 10 μs (9.960 μs) which opens very interesting perspectives for future reconfigurable decoders.

In order to compare the complexity of the proposed solution, the FPGA synthesis results are presented in Table 3.7. This table shows that the area in terms of flip flops (FF) and LUTs of the AMBA AXI4 master interface is the lowest when compared to the CoreConnect PLB4 and the proposed bus master interfaces. Indeed, the AXI-lite version of the AXI bus has been implemented in

this prototype. However, the slave interface of the propose bus owns the lowest complexity leading to the lowest total complexity.

Infrastructure Component	Area (in μm^2)
MI	1 790
SI	1 150
Selector	784
Infrastructure for 4 RdecASIPs	15 199
4 RDecASIP	739 968

Table 3.8: *Area of the proposed configuration architecture*

Nb. ASIPs	Transfer latency (in ns)		Speedup
	FPGA	ASIC (estimated)	
4	1 032	86	12
6	1 176	98	12
8	1 320	110	12
16	1 896	158	12
32	3 048	254	12
64	5 352	446	12
128	9 960	830	12

Table 3.9: *Estimated Configuration transfer time in ns for an ASIC implementation*

A logic synthesis of the proposed bus components was also done with a 65nm CMOS technology with a clock frequency objective equals to 500MHz. Table 3.8 shows the area evaluation for the three components of the proposed configuration infrastructure. The logic overhead due to the configuration infrastructure is 0.015 mm^2 which leads to a low area penalty of 2% regarding the logic area of the 4 RDecASIPs (0.704 mm^2). The complexity of the Selector is the lowest one since only a comparison with the input component address is necessary to know if the input vector has to be copied into the output or not. The complexity of the SI and MI components is quite similar. The difference is mainly due to the presence of a counter in the MI for incremental burst while 8-bit comparators are implemented in the SI for address comparison. Furthermore, considering an ASIC implementation with the frequency objective of 500 MHz, a significant speedup on the configuration transfer latencies shown in Table 3.6 can be expected compared to the 125 MHz FPGA prototype. Table 3.9 shows the estimated configuration transfer time for an ASIC implementation. Results are estimated using Equation (3.1) with a bus frequency fixed to 500 MHz and show that a speedup of 12 can be reached compared to the FPGA implementation.

Compared to the recent related work proposed in [46], where the configuration infrastructure consists of several buses, each connected to a group of 4 PEs. Up to 8 buses have been implemented to configure 35 PEs. However, the way the buses are driven is not described in details in [46] but the management of the 8 buses in parallel should increase the complexity of the *configuration manager* used to load new configurations. Results presented in this section show that the proposed configuration infrastructure offers an efficient solution for the UDec implementing up to 128 ASIPs. It guarantees configuration latency below $10\mu s$ in a FPGA implementation providing a very low configuration latency overhead and meeting the configuration latency challenge as explained in Section 1.2.2. Such a solution meets future standard requirements where a code switch can be done as early as one data frame ahead [2].

3.3 Summary

This chapter has provided an analysis of the lack of flexibility of the UDec architecture and proposes original solutions in order to reach an efficient dynamic configuration management. Flexibility has been brought at the Butterfly topology NoCs level, at the ring buses level and at the platform controller level in order to adapt at run-time the number and the location of the activated processing elements. In the second part of this chapter, the definition and implementation of a dedicated configuration infrastructure providing an efficient and low complex solution for configuration data transfer to each configuration memory of the implemented RDecASIPs and to the platform controller configuration memory have been presented. Implementation results show that configuration transfer latency below $10\mu s$ is reached with an FPGA implementation considering up to 128 implemented RDecASIPs. Moreover, for an ASIC implementation considering up to 128 RDecASIPs, configuration transfer latency below $1\mu s$ can be reached. This very low configuration transfer latency is a key feature in order to support dynamic configuration in the multi-mode and multi-standard scenario presented in the first chapter (Section 1.2.2).

4

Configuration management for the UDec architecture

This last chapter addresses the configuration management of the UDec architecture. Indeed previous chapters provide a solid base for an efficient dynamic configuration of the platform through the RDecASIP processor proposed in chapter 2 and the dedicated configuration infrastructure described in chapter 3. In order to provide a complete solution it is now required to propose some techniques in order to meet at run-time application constraints through dynamic reconfiguration of the whole platform. Indeed to respect decoding performances in terms of throughput and FER in a multi-mode and multi-standard scenario, it is mandatory to dynamically tune the level of sub-block parallelism (which directly impacts the number of active ASIPs). The main point when performing the reconfiguration of the platform is to define the new configuration parameters which depend of the execution context. Thus, this chapter proposes two configuration approaches in order to manage at run-time the execution of the platform. The first one proposes to store the configuration information for all possible configurations in a global memory. Such a solution requires an off-line analysis of potential configurations that will be supported by the platform. The second one is more dynamic and proposes to generate at run-time each new configuration. Such an approach provides more flexibility at the cost of additional computation as will be detailed. To address these points, this chapter starts with the analysis of the decoding performances in terms of FER taking into account the level of sub-block parallelism and shuffled mode in order to determine at-run time, the number of necessary decoding iterations. The second part of this chapter presents and analyses the two configuration managements. Finally, a discussion and a comparison with the most relevant SoA work is provided.

4.1 Parallelism impact on decoding performance

This section introduces the studies on sub-block parallelism and shuffled decoding parallelism impact on turbo decoding performance in terms of FER or BER. Indeed, this point is important since the level of sub-block parallelism determines the number of RDecASIPs that have to be activated and reconfigured.

4.1.1 Sub-block parallelism

In the context of this work considering the UDec architecture, sub-block parallelism method is associated with initialization by message passing which enables better error rate performance compared to initialization by acquisition as demonstrated in [15]. As explained in Section 1.1.4.2, this method initializes dynamically a sub-block with recursion metrics computed during the last iteration in the neighboring sub-blocks. The authors of [15] have studied the impact of

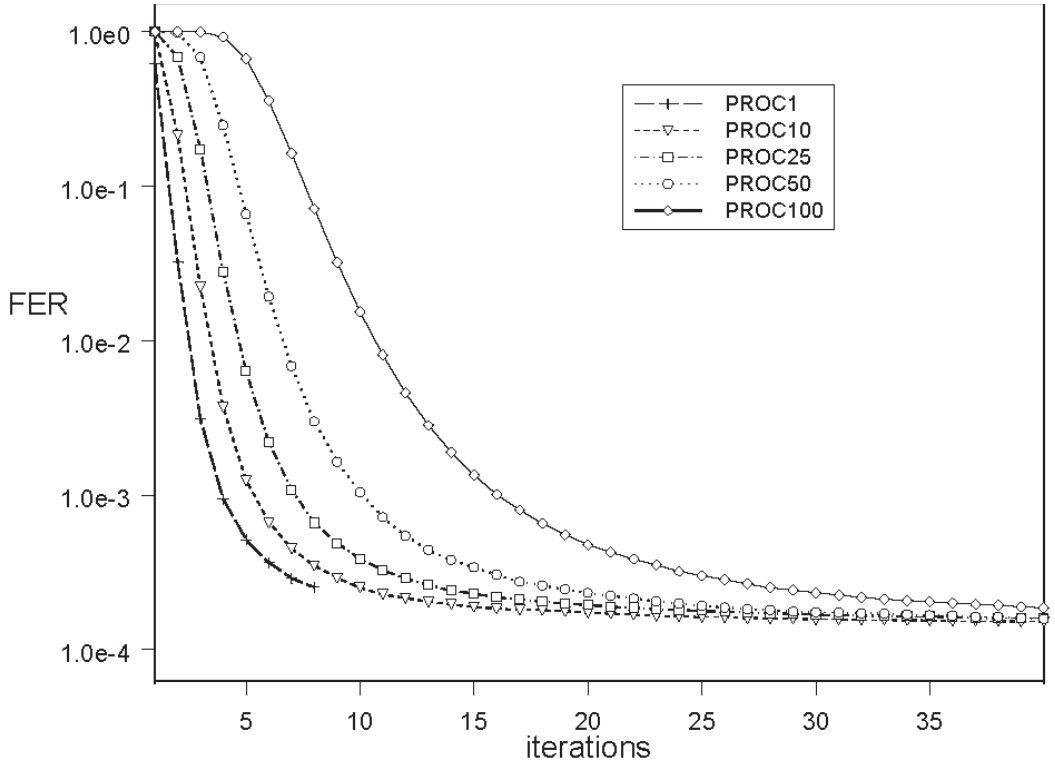


Figure 4.1: *Convergence of message passing method example of DVB-RCS, code rate=6/7, 188 bytes frame, SNR=4.2dB, 5 bit quantification, Log-MAP algorithm*

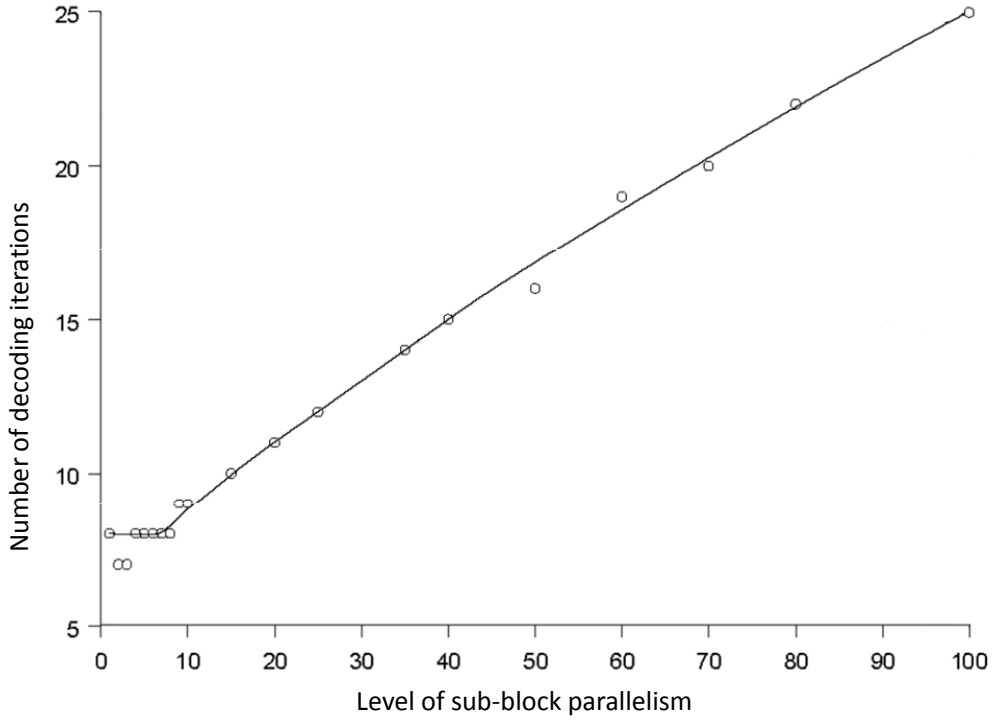


Figure 4.2: *Number of iterations with message passing method, DVB-RCS, code rate=6/7, 188 bytes frame, SNR=4.2dB, 5 bit quantification, Log-MAP algorithm*

sub-blocking on the turbo decoding performance in terms of FER considering message passing. Figure 4.1 presents the FER performance for different parallelism (from 1 to 100) degrees in function of iterations number. This figure shows that asymptotic error rate is not affected by message passing approach whatever the parallelism degree. Thus it ensures that initialization by message passing can be used without performance degradation in terms of decoding quality by increasing the number of decoding iterations in parallel with the level of sub-block parallelism. Also, it can be noticed that the convergence of the decoding process is slowed down when the parallelism degree increases reducing the architecture efficiency.

Figure 4.1 reveals that with sub-block parallelism, additional iterations are mandatory to reach a given FER. This is a key point in the context of this work targeting the dynamic configuration of the multi-ASIP UDec architecture. The number of decoding iterations with message passing for a given level of sub-block parallelism can be estimated at fixed FER value as shown in Figure 4.2 for a

Code rate / Frame size (in bits)	6/7	3/4	1/2	1/3
424	3	3	4	5
848	4	5	8	9
1504	8	11	16	20
Key sub-block size	180	135	100	85

Table 4.1: *Threshold values for DVB-RCS*

given DVB-RCS configuration example.

It appears that the necessary number of iterations is constant for low level of sub-block parallelism (8 on Figure 4.2). After a specific threshold, it increases linearly with the level of parallelism. Thus, in order to reach a given decoding performance defined by a fixed FER value, the number of necessary decoding iterations is given by Equation (4.1).

$$N_{iter} = N_{iterP=1} + \frac{P}{T} \tag{4.1}$$

where $N_{iterP=1}$ is the number of decoding iterations that have to be performed when the level of sub-block parallelism $P = 1$ for a fixed SNR, and T is a constant threshold which depends of the code rate and data frame size. The threshold position can be interpreted as the minimum sub-block size which provides reliable recursion values at the end of the first iteration. Under this minimum size, recursion values have to be refined using more iterations.

In [55], intensive simulations have been performed in order to extract threshold values for the DVB-RCS standard considering three frame sizes and four code rate defined in the standard. These threshold values are presented in Table 4.1. On the one hand, we observe that the threshold linearly increases with the frame size for a given code rate. Thus, a key sub-block size can be extracted for each code rate. This key sub-block size corresponds to the initial sub-block size from which the number of decoding iterations is increased. It is determined by computing the average value of sub-block size from which the number of decoding iterations is increased for the different frame sizes considering a fixed code rate. For example, considering the code rate equals to 6/7, the key sub-block size equals to $(\frac{424}{3} + \frac{848}{4} + \frac{1504}{8})/3 = 180$ bits. On the other hand, we observe that the threshold decreases when the code rate increases for a given frame size. Indeed, a low code rate provides more redundant information increasing the reliability of obtained recursion metrics.

This section has illustrated how the number of decoding iterations evolves depending of the level of sub-block parallelism. The next section shows the impact

Level of Sub-block parallelism	Nber. of iterations without shuffled	Nber. of iterations with shuffled	shuffled efficiency
1	8	12	0.66
4	11	15	0.73
8	16	20	0.8
53	47	51	0.92

Table 4.2: *Comparison of necessary number of decoding iterations regarding the level of sub-block parallelism for 53 bytes DVB-RCS interleaving code rate=6/7, SNR=4.0dB, Log-MAP algorithm, FER=*$1.6e^{03}$

Level of Sub-block parallelism	Nber. of iterations without shuffled	Nber. of iterations with shuffled	shuffled efficiency
1	8	11	0.72
2	9	11	0.82
4	9	12	0.75
16	13	15	0.86
64	19	23	0.83
128	34	37	0.92

Table 4.3: *Comparison of necessary number of decoding iterations regarding the level of sub-block parallelism for 188 bytes DVB-RCS interleaving code rate=6/7, SNR=4.0dB, Log-MAP algorithm, FER=*$1.6e^{03}$

of the shuffled decoding on the turbo decoding performance.

4.1.2 Shuffled decoding

For shuffled decoding, the parallelism degree is limited to the number of component decoders (2 in the UDec architecture). The work presented in [63] shows that when using shuffled decoding the number of decoding iterations is slightly increased compared to the number of decoding iterations required to reach the same error rate performance using serial decoding. For example, in [63], the authors study the shuffled decoding parallelism efficiency (i.e. decoding performance in terms of FER) for the DVB-RCS interleaving rules. Table 4.2 and Table 4.3 show that the number of decoding iterations guaranteeing the same decoding performance has to be increased when shuffled decoding is used. However, it can be highlighted that the shuffled decoding efficiency increases when the level of sub-block parallelism increases too. Consequently, the number of additional decoding iterations when shuffled decoding is enabled is relatively constant, i.e. 4 in Table 4.2 and between 2 and 4 in Table 4.3.

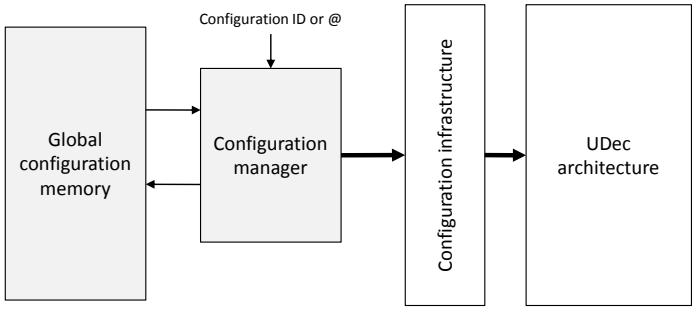


Figure 4.3: *Pre-computed configuration principle*

This section has shown that the number of decoding iterations evolves depending of both sub-block and shuffled decoding parallelisms. This is a key feature considering the context of this thesis work targeting the dynamic configuration of the UDec architecture. Indeed, in order to adapt the platform for a given throughput, the number of activated RDecASIPs can be tuned at run-time. However, as shown in this section, when the level of sub-block parallelism increases, the number of decoding iterations has to be adapted in order to guarantee the decoding performance in terms of FER. Obviously, the number of decoding iterations influences the throughput of the decoder as demonstrated by Equation 1.16. This fact must be taken into account when a configuration is generated at run-time. The next section analyzes the configuration management scenario where the configuration information is pre-computed and stored in a global configuration memory.

4.2 Pre-computed configuration management

This section proposes to study the solution in which the configurations are pre-computed and stored in a global configuration memory. Since the different configurations are pre-computed at design time in this solution, all different configurations that can be launched at run-time have to be stored in the global configuration memory as shown in Figure 4.3. In this context, the configuration manager is a simple Finite State Machine that performs the configuration process from a configuration ID or a configuration address in the global configuration memory. The configuration information is sequentially sent to the configuration infrastructure presented in the previous chapter that loads the different configuration memories of the UDec architecture.

In this section we consider the following context: the RDecASIP configuration memory presented in Section 2.2.2, the platform configuration memory presented

in Section 3.1.2 and the UDec architecture implementing the configuration infrastructure presented in Section 3.2 and a maximum of 128 processor (i.e. 64 RDecASIPs for each component decoder) leading to a maximum level of sub-block parallelism of 64. Considering a global configuration memory that contains all different configuration alternatives, it is necessary to compute the required size for this memory. For that purpose, the following equations is used.

$$ConfigLoad_{ASIPs}(P) = (4.P + 14).26 \quad (4.2)$$

Equation (4.2) highlights the configuration load in bits for the ASIPs for a given level of sub-block parallelism P. Indeed, thanks to the configuration infrastructure associated with the UDec architecture allowing multicast and broadcast transfers the following transfers are required:

- one transfer of two 26-bit memory lines for each RDecASIPS (i.e. $2 \times 2 \times P$ 26-bit transfers)
- two multicast 26-bit transfers of five memory lines
- one broadcast 26-bit transfer of four memory lines.

Since the configuration information is pre-computed and stored in a global configuration memory, all possible configurations have to be taken into account to provide the complete flexibility of the platform. Thus, considering 64 levels of sub-block parallelism, Equation (4.2) becomes Equation (4.3).

$$ConfigLoad_{ASIPs} = \sum_{p=1}^{64}(4.P + 14).26 \quad (4.3)$$

However, configuration parameters such as the frame size, the number of decoding iterations, and the decoding mode (shuffled or serial) can be tuned depending on the configuration. Thus configuration load expressed by Equation (4.3) has to be multiplied by the number of different frame sizes of each supported standard, the number of different possibilities for decoding iterations and by 2 for each decoding mode (shuffled or serial). Consequently, Equation (4.4) expresses the total ASIPs configuration load in bits for all possible configurations considering the UDec architecture implementing 128 RDecASIPS.

$$TotalConfigLoad_{ASIPs} = 2.ConfigLoad_{ASIPs}.N_{Frame}.N_{iterations} \quad (4.4)$$

where N_{Frame} is the number of frame sizes in each supported standard (e.g. 188 for LTE) and $N_{iterations}$ is the number of different possibilities for decoding iterations that can be necessary to decode a frame and to reach a given decoding performance. Moreover, besides the ASIPs configuration memories, the configuration information for the configuration memory of the platform controller has also to be stored in the global configuration memory. As for the RDecASIP

configuration memory all different configurations have to be stored leading to a configuration load in bits as expressed by Equation (4.5).

$$ConfigLoad_{Controller} = 1490944.N_{Frame}.N_{iterations} \tag{4.5}$$

where the constant value 1490944 is obtained by multiplying the 64 possibilities of sub-block parallelism, the 64 possibilities of ASIP shift location as shown in Section 3.1.1, the two turbo decoding modes and the seven 26-bit memory lines of the controller configuration memory presented in Section 3.1.2. Thus, the total configuration load in bits that have to stored in a global configuration memory is obtained by Equation (4.6).

$$\begin{aligned} TotalConfigLoad &= ConfigLoad_{Controller} + TotalConfigLoad_{ASIPs} \\ &= N_{Frame}.N_{iterations}.[1490944 + 2.(4.P + 14).26] \\ &= N_{Frame}.N_{iterations}.1970176 \end{aligned} \tag{4.6}$$

Considering Equation (4.6), the 188 frame sizes of the LTE standard and a range of decoding iteration of 20, the total configuration load that has to be stored in a global configuration memory is around 883 MBytes. This result shows that the memory cost of such an approach in order to offer the complete range of flexibility of the UDec architecture is prohibitive and can not realistically be implemented. In the next section, a solution where the configuration information is generated at run-time is proposed.

4.3 Run-time configuration generation management

This section introduces a configuration manager that generates the configuration information at run-time and studies its impact on the configuration latency in the context of the multi-mode and multi-standard scenario presented in Section 1.2.2. The next section builds on this scenario where the total configuration latency (including the configuration information generation and the configuration information transfer) must be lower or equal to the decoding duration of the current data frame whatever is the configuration that has to be performed. Indeed, in this scenario the Turbo decoder deals with input frames that are associated with different throughput and FER objectives. Consequently, each frame received by the Turbo decoder is associated to a specific configuration which takes into account the application requirements and the channel quality.

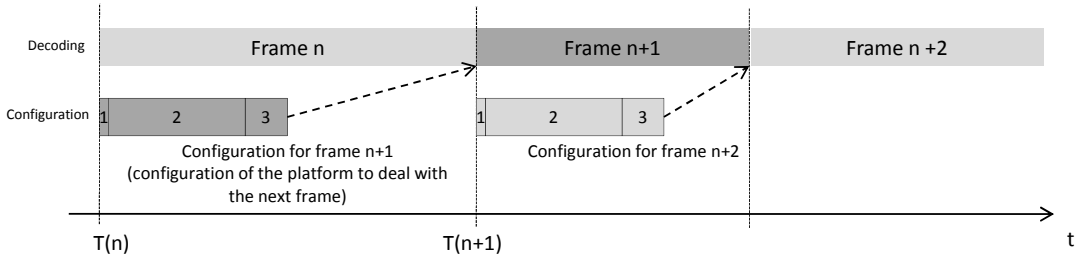


Figure 4.4: *Configuration steps of the UDec platform*

4.3.1 Restricted configuration management

In the context of this work, the maximum configuration latency of a frame is constrained by the previous frame decoding duration as demonstrated in Section 1.2.2. The configuration of the UDec platform is divided in three steps as shown in Figure 4.4:

1. The configuration manager receives the configuration order associated with the frame parameters (i.e. frame size, standard, throughput, targeted BER) necessary to generate the configuration for the RDecASIPs.

2. the configuration manager generates the configuration parameters for each selected RDecASIP presented in Section 2.2.2.

3. The configuration parameters for each selected RDecASIP is transfered through the configuration infrastructure presented in Section 3.2.2.

For this sudy, we assume that the configuration manager is a GPP that generates at run-time the configuration information for the entire UDec architecture based on the configuration parameters received with the configuration order for the next frame. These parameters are: the frame size, the throughput requirement, the standard, the decoding mode and the number of decoding iterations. From these basic parameters, the number of activated RDecASIPs is first determined depending on the number of decoding iterations using Equation (4.7) which is deduced from Equation (1.16) that determines the throughput of the UDec architecture.

$$N_{ASIP} = \frac{2 \times Throughput \times N_{instr} \times N_{iter}}{F_{clk}} \tag{4.7}$$

where F_{clk} and N_{iter} are the clock frequency of the system and the number of decoding iterations respectively. Moreover, an average of $N_{instr} = 4$ instructions per iteration are needed to process 1 symbol. The number of active ASIPs using the previous Equation (4.7) has to be divided by two when the shuffled decoding is

enabled. Then, the contents of the different configuration memories of the UDec architecture can be generated. For that purpose, a C-code has been developed and configuration generation latency has been studied.

In order to evaluate the configuration generation latency of the proposed approach the developed C-code allowing a run-time configuration generation has been implemented on an ARM cortex A9 core with a frequency of 600 MHz. It is important to note that the considered C-code has not been fully optimized and not parallelized. The configuration generation latencies for a level of sub-block parallelism of 1 and 32 (2 and 64 RDecASIPS respectively) are 5 μs and 14 μs respectively and it can be assumed that it evolves linearly since the number of configuration parameters that have to be generated evolves linearly regarding the level of sub-block parallelism. This configuration generation latency is quite large compared to the configuration transfer latency, which is lower than $1\mu s$ (table 3.9) considering an ASIC implementation of the configuration infrastructure proposed in the previous chapter. However, it does not impact the generality of the contributions presented in the rest of this chapter. A discussion on this specific point is provided in Section 4.4.

In the context of the multi-mode and multi-standard scenario considered, where the configuration latency for a frame is limited by the decoding duration of the current frame, the minimum decoding duration respecting this rule can be deduced from the maximum configuration latency of the platform which is reached when all the RDecASIPs processors have to be configured. It guarantees that the configuration latency is lower than the decoding duration whatever is the configuration which has to be performed. Thus, the maximum achievable throughput is theoretically limited for a given frame size and is given by Equation (4.8) where $Frame\ duration_{min}$ is equal to the maximal configuration generation latency plus the maximal configuration transfer latency. For instance, $Frame\ duration_{min}$ is equal to $14 + 0.446$ μs considering an ASIC implementation of the UDec architecture implementing 64 RDecASIPs associated to the proposed configuration infrastructure with a frequency of 500 MHz for the entire platform.

$$T_{max}\ (in\ bps) \quad = \quad \frac{Frame\ size\ (in\ bits)}{Frame\ duration_{min}\ (in\ s)} \tag{4.8}$$

Considering the UDec platform implementing RDecASIPs, the maximum achievable throughput is limited by the number and the performance of the ASIPs. Considering Equation (4.7) and Equation (4.8) where the number of RDecASIPs is limited to 64, Table 4.4, shows the maximum estimated throughput achievable allowing the support of a multi-mode and multi-standard scenario and the corresponding number of RDecASIPs that have to be used for different frame sizes and

Frame size (bits)	$Throughput_{max}$ (Mbps)	N_{ASIP}	Configuration latency (μs)	number of decoding iteration
96	6.6	2	5.074	8
480	33.2	6	5.678	8
880	60.9	8	5.98	8
1920	132.9	18	7.49	8
4800	332.2	54	12.926	10
6144	363,6	64	14.446	11

Table 4.4: *Estimated maximum throughput supporting the considered multi-mode and multi-standard scenario with $N_{iterP=1}$=8, T=10 and P_{max}=32*

a number of reference decoding iterations (i.e. $N_{iterP=1}$) equals to 8 . Based on $N_{iterP=1}$, the necessary number of decoding iteration is computed using Equation (4.1). Moreover, for the example of this section, a fixed threshold T equals to 10 is considered in Equation (4.1).

Results of Table 4.4 show that a throughput up to 363,6 Mbps can be reached considering a 6144-bit frame size for a maximum level of sub-block parallelism (P_{max}) equals to 32. This maximum throughput does not correspond to the theoretical maximal throughput computed using Equation (4.8) (i.e. 425.3 Mbps). Indeed, the maximum level of sub-block parallelism (i.e.P_{max} = 32) is used to reach 363.6 Mbps respecting the decoding performance thanks to 11 decoding iterations. Thus, for the 6144-bit frame size, the throughput performance is limited by the maximum level of sub-block parallelism P_{max} (i.e. the number of implemented ASIPs) and not by the maximum configuration latency. For the other frame sizes, the maximal throughput is defined by Equation (4.8). Equation (4.7) is then used in order to determined the number of RDecASIPs that have to be activated to reach this throughput considering 8 decoding iterations. For example, considering this maximum throughput and the 4800-bit frame size, 54 RDecASIPs are necessary. Thus, considering this approach the 64 ASIPs can not be used to reach a throughput higher than the throughput defined by Equation (4.8) without violating the limit imposed by the considered scenario (i.e. the configuration latency for a frame must be lower or equal to the decoding duration of the current frame). However, a greater number of ASIPs can be activated in order to support the maximum throughput with a higher number of decoding iterations than the 8 decoding iterations considered in the example of Table 4.4. The strict respect of the configuration latency considered in this section reduces the performance possibilities offered by the UDec architecture in terms of throughput since the maximum performance is limited by the configuration management method and not by the architecture itself. In order to reduce the

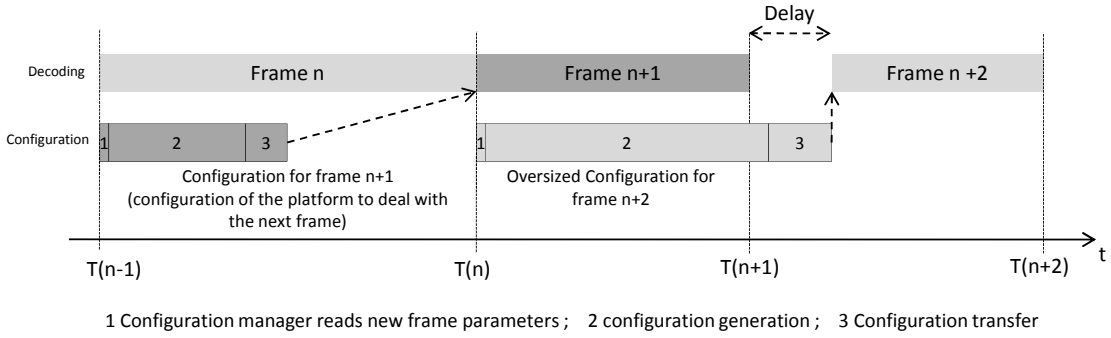


Figure 4.5: *Oversized configuration principle*

impact of the configuration management on the maximum performance, the next section proposes a solution in which the configuration latency for a frame can be greater than the decoding duration of the current frame respecting the targeted throughput.

4.3.2 Oversized configuration management

This section proposes to reduce the configuration management impact on the maximum throughput which can be reached by the UDec architecture implementing 64 RDecASIPs by generating oversized configuration at run-time.

4.3.2.1 Oversized configuration principle

The Section 4.2 has shown that the configuration management method impacts the maximum throughput performance of the UDec architecture. In order to reduce this impact, a configuration management method introducing a flexible configuration latency that can become greater than the decoding duration of the current frame is presented in this section.

In the example of Figure 4.5, the configuration latency corresponding to the frame (n+2) is greater than the decoding duration of the frame (n+1). Thus, the decoding of the frame (n+2) can not start at $T(n+1)$. This leads to an extra delay between the end of frame (n+1) decoding and the beginning of frame (n+2) decoding. However, in the context of the multi-mode and multi-standard scenario considered in this thesis work, the decoding duration allocated to the frame (n+2) is fixed. Consequently, the new decoding duration for a data frame becomes:

$$Latency_{frame(n+2)} = (T_{n+2} - T_{n+1}) - Delay \tag{4.9}$$

where $Delay$ is defined by:

$$Delay_{frame(n+2)} = ConfigurationLatency_{frame(n+2)} - (T_{n+1} - T_n) \quad (4.10)$$

when $ConfigurationLatency_{frame(n+1)} > (T_n - T_{n-1})$ else $Delay = 0$.

Considering the new decoding duration, the configuration for the frame (n+2) has to be oversized in order to adapt its associated throughput to the reduced decoding duration. This method provides a solution to go beyond the limit fixed by the configuration latency. Compared to the Restricted configuration management presented in Section 4.3.1, the decoding duration can now be lower than the 14.446 μs which was imposed by the maximum configuration latency of the UDec architecture implementing 64 RDecASIPs, i.e. 14 μs to generate the configuration and 0.446 μs to transfer the configuration parameters. This new flexibility aspect leads to the possibility for increasing the maximum throughput supported for a given frame size. In the next section, the algorithm allowing to generate oversized configuration at run-time is studied.

4.3.2.2 Oversized configuration generation

This section tackles the implementation of an algorithm for the generation of the configuration information for the UDec platform by implementing the oversized configuration management previously presented. It also considers the dynamic evolution of the number of decoding iterations regarding the level of sub-block parallelism as explained in Section 4.1.

When a new configuration has to be generated, the configuration can be analyzed in order to determine if this configuration introduces a configuration *Delay* (i.e. the configuration latency for this configuration is greater than the current decoding duration). If the introduced delay is not null, an oversized configuration has to be generated in order to counterbalanced the effect of the delay and to respect the original deadline associated to the frame. To respect the original deadline the level of sub-block parallelism is increased to reduce the decoding duration. However, as demonstrated in Section 4.1, when the level of sub-block parallelism increases, the number of decoding iterations has to be increased respecting the Equation (4.1). Consequently, an algorithm allowing the selection of a UDec configuration respecting the deadline associated to a frame and the decoding performance in terms of FER is presented in Listing 4.1.

The algorithm presented in Listing 4.1 requires the following inputs: (1) the frame size and the throughput of the current configuration in order to compute $(T_n - T_{n-1})$, (2) the frame size and the throughput associated with the new configuration in order to compute both $ConfigurationLatency_{frame(n+1)}$ and

$Throughput_{oversized}$, (3) the reference number of decoding iterations for P=1, (4) the threshold value T is used to compute the necessary number of decoding iterations and (5) P_{max} is the maximum level of sub-block parallelism supported by the platform. This algorithm is built with a search loop with level of sub-block incrementation. This incrementation increases the throughput by rising the number of activated ASIPs. For each level of parallelism the corresponding number of decoding iterations is deduced from $N_{iterP=1}$ and T. Then the throughput ($Throughput$) corresponding to the level of sub-block parallelism and the computed number of decoding iterations is calculated. The configuration Delay is then computed as explained in Section 4.3.2.1. The new throughput ($Throughput_{oversized}$) that has to be reached by the architecture is then calculated taking into account the configuration delay. Finally, the architecture throughput and the objective oversized throughput are compared. If the oversized throughput is greater than the current throughput of the architecture, the level of sub-block parallelism has to be increased to reach the throughput requirement. Once the loop iterations finished, the architecture throughput and the objective oversized throughput values are compared. If the architecture throughput is greater than the objective oversized throughput then a configuration solution exists with a level of sub-block parallelism of P and N_{iter} decoding iterations. If no solution is found, shuffled decoding can be enabled if the condition in terms of frame size and code rate are met. Indeed, shuffled decoding can not be used efficiently on small frame and high code rates configuration [55]. If shuffled decoding can not be used, the UDec architecture is not able to support such a configuration respecting the required decoding performance.

Listing 4.1: *Oversized configuration research algorithm*

```
1  // Level of sub-block parallelism initialization
2              P = 0;
3              do {
4  // Level of sub-block parallelism incrementation
5                  P = P + 1;
6  // Number of decoding iterations computation taking into account the level
7   of sub-block parallelism
8                  N_iter = N_iterP=1 + P/T;
9  // Throughput computation in serial mode
10                 Throughput = (F_clk × P) / (N_instr × N_iter);
11 // Delay computation
12                 Delay = ConfigurationLatency_P − (T_n − T_n−1);
13 // Delay validation
14                 if (Delay <= 0) Delay = 0;
15 // Throughput adaptation considering the delay
```

16 $Throughput_{oversized} = \frac{FrameSize_{frame(n+1)}}{OriginalDecodingDuration_{frame(n+1)} - Delay}$;
17 // If the required throughput is greater than the reached throughput then
18 a new loop iteration is performed
19 }**while** ($Throughput_{oversized} > Throughput$ **and** $P \leq P_{max}$);
20 // Result analysis
21 **if** ($Throughput_{oversized} < Throughput$ **and** $P \leq P_{max}$)
22 **then EXIT**;
23 **else ERROR**;

This algorithm presents two advantages compared to the configuration management method presented in Section 4.3.1. Firstly, the maximum throughput for a given frame size and for a reference number of decoding iterations is not limited by the maximum configuration latency anymore. The actual decoding duration of the current frame is now taken into account at run-time in the algorithm through the $(T_n - T_{n-1})$ term in the configuration delay equation (line 12 in Listing 4.1).

Secondly, this algorithm allows the introduction of a configuration latency greater than a decoding duration introducing much more flexibility in the configuration management.

Figure 4.6 shows an example of oversized configuration search in which two frames with different configurations are sequentially decoded. The first frame is a 4800-bit frame decoded at 400 Mbps and the second frame is a 1920-bit frame decoded at 280 Mbps. Moreover, $N_{iterP=1} = 8$, $T = 10$ are considered. The decoding duration of the first frame is $\frac{4800}{400} = 12\mu s$. The original decoding duration objective is $\frac{1920}{280} = 6.86\mu s$. The algorithm presented in Listing 4.1 is executed in order to determine the number of RDecASIPs which have to be activated to perform the second frame. Figure 4.6 shows that from a level of sub-block parallelism equals to 23, a configuration delay is inserted leading to a lower decoding duration objective allowing the respect of the deadline. It is interesting to note that a configuration beyond the configuration management limit of the previous method which does not introduce a configuration delay and proposes a better throughput can be found (shown in red diamonds). Finally, this figure shows that four configurations (shown in green diamonds) for the second frame respect the original deadline and the decoding performance by taking into account the dynamic evolution of the number of decoding iterations necessary to reach a given FER.

The algorithm described in Listing 4.1 does not guarantee to find a solution. When the conditions that satisfy efficient shuffled decoding mode requirement are not met (i.e low code rate and small frames), a second search step can be performed. This search consists in: setting the maximal value for the level of

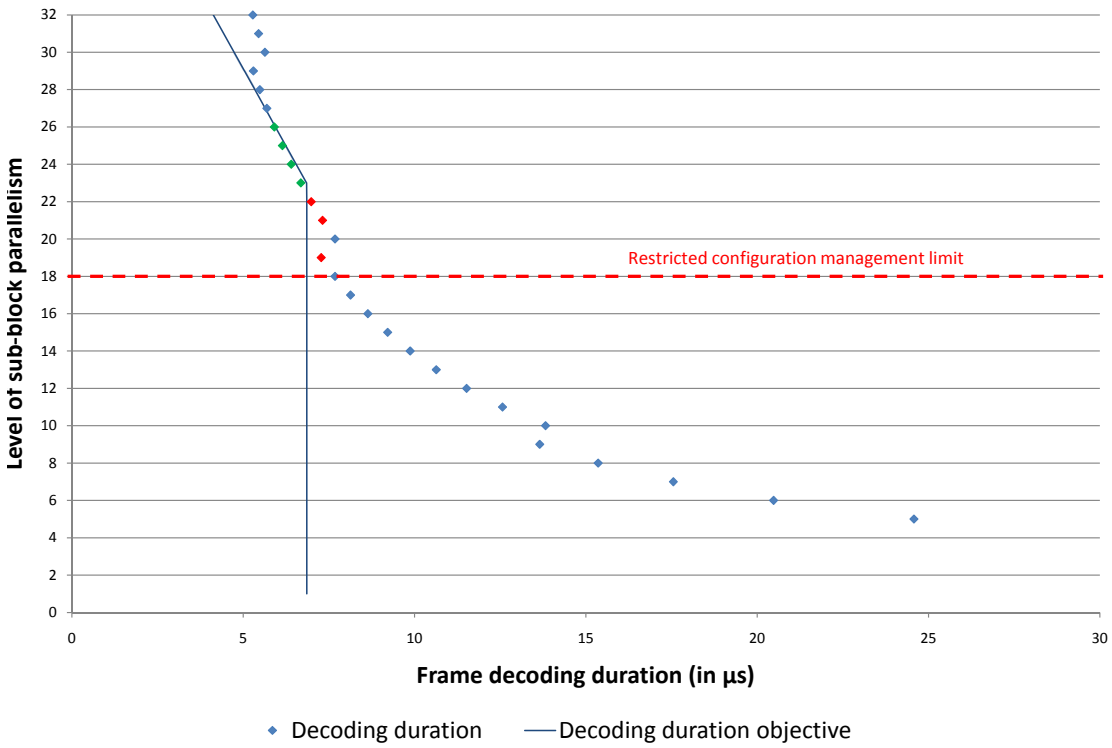


Figure 4.6: *Oversized configuration search example.* 1^{st} *frame: size* $= 4800$ *bits, throughput* $= 400$ *Mbps.* 2^{nd} *frame: size* $= 1920$ *bits, throughput* $= 280$ *Mbps.* $N_{iterP=1} = 8$, $T = 10$ *and* $P_{max} = 32$

sub-block parallelism and then, determining the maximum number of iterations that can be performed respecting the decoding deadline. The complete algorithm is presented in Listing 4.2.

Listing 4.2: *Oversized conguration research algorithm including shuffled decoding*

```
1  // serial decoding mode similar to Listing 4.1
2          P = 0;
3          do {
4              P = P + 1;
5              N_iter = N_iterP=1 + P/T;
6              Throughput = (F_clk × P) / (N_instr × N_iter);
7              Delay = ConfigurationLatency_P − (T_n − T_n−1);
8              if (Delay <= 0) Delay = 0;
9            Throughput_oversized = FrameSize_frame(n+1) / (OriginalDecodingDuration_frame(n+1) − Delay);
10         }while (Throughput_oversized > Throughput and P ≤ P_max);
11         if (Throughput_oversized < Throughput and P ≤ P_max)
12         then EXIT;
```

```
13          else {
14  // If shuffled decoding possible
15              if (shuffled = 1) {
16                  P = 0;
17                  do {
18                      P = P + 1;
19  // Number of decoding iterations computation taking into account the level
20   of sub-block parallelism and shuffled decoding
21                      N_iter = N_iterP=1 + N_Iter.shuffled + P/T;
22  // Throughput computation in shuffled mode
23                      Throughput = (2×F_clk×P)/(N_instr×N_iter);
24                      Delay = ConfigurationLatency_frame(n+1) − (T_n − T_n−1);
25                      if (Delay <= 0) Delay = 0;
26                      Throughput_oversized = FrameSize_frame(n+1)/(OriginalDecodingDuration_frame(n+1)−Delay);
27                  }while (Throughput_oversized > Throughput and P ≤ P_max);
28  // Shuffled result analysis
29                  if (Throughput_oversized < Throughput and P ≤ P_max)
30                  then EXIT;
31              }
32  // No solution in shuffled mode
33  // Maximal value for the level of sub-block parallelism
34              P = P_max;
35  // Maximal delay computation
36              Delay = ConfigurationLatency_p=P_max − (T_n − T_n−1);
37              Throughput_oversized = FrameSize_frame(n+1)/(OriginalDecodingDuration_frame(n+1)−Delay);
38              N_iter = N_iterP=1 + P/T;
39  // Loop for number of iterations decrementing
40              while (Throughput_oversized > Throughput and){
41                  N_iter = N_iter − 1
42                  Throughput = F_clk×P/(N_instr×N_iter);
43              }
44          }
```

This algorithm starts by a search considering serial mode as presented in Listing 4.1. Then if no solution is found, a search considering shuffled decoding is performed if the shuffled mode is enabled. In this case, a constant $N_{Iter.shuffled}$ is added to the number of decoding iterations (line 21) and the throughput of the architecture is multiplied by 2 (line 23). If no solution is found in shuffled mode or if the shuffled mode is disabled for the configuration, a solution with the maximum level of parallelism in serial decoding mode is searched (line 34 to

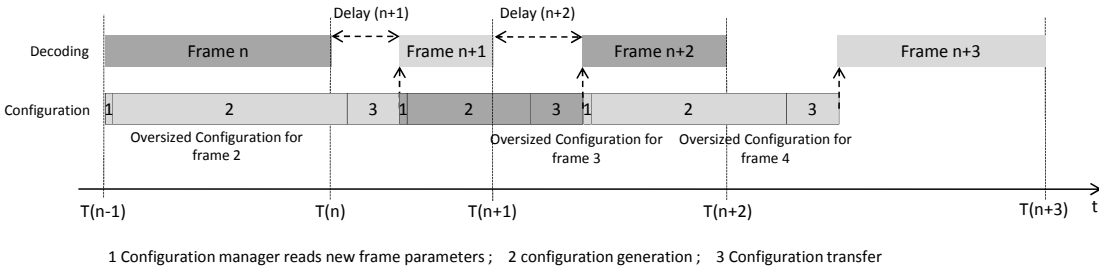


Figure 4.7: *Stable configuration scenario*

43) by reducing the number of decoding iterations until the decoding deadline is respected. Obviously, the found solution does not guarantee the decoding performance in terms of FER since the number of iterations can be hugely reduced. However this method guarantees to find a configuration that respect the decoding deadline whatever the throughput objective reachable by the UDec architecture implementing P_{max} RDecASIPs.

The next section studies the impact on the maximum decoding performance in terms of throughput of the proposed flexible configuration management regarding multi-mode and multi-standard configuration scenarios.

4.3.3 Oversized Configuration management scenario

As demonstrated in the previous section, the generation of oversized configuration is a promising approach in order to benefit from the UDec architecture by allowing an extra configuration delay which is compensated by an oversized configuration. However, considering a flow of data frames each associated with specific parameters and high requirements in terms of throughput and decoding performance, this method can lead to a scenario where the decoding performance is not guaranteed anymore since the number of decoding iterations has to be reduced in order to respect the decoding deadline. In this section, two scenarios guaranteeing the decoding performance are presented.

4.3.3.1 1 frame - 1 configuration

This first scenario, presented in Figure 4.7, illustrates a stable scenario where the configuration delay is managed whatever is the parameters and requirements associated to a frame which has to be decoded. In this scenario, when the configuration latency for a frame is greater than the decoding latency of the current frame, the extra delay does not impact the existence of a configuration solution for the next frame. Indeed, in the example of Figure 4.7, the extra delay

($Delay(n+1)$) imposed by the configuration for the frame (n+1) delays the configuration generation and the configuration transfer for the frame (n+2) (i.e. $Delay(n+2)$ increases) which should start at $T(n)$. Thus, this delay reduces the chance to find an oversized configuration for the frame (n+2) allowing the respect of the decoding deadline ($T(n+2)$) and the decoding performance.

This scenario guarantying the decoding performance requirement is verified when a configuration solution respecting the decoding deadline and the decoding performance is found for each configuration whatever is the frame parameters and requirements in terms on throughput and decoding performance. The existence of such a scenario depends of the consecutive frame parameters and requirements. Indeed, to decode consecutive data frames associated to different parameters with high throughput and high decoding performance will lead to an unstable scenario where the configuration delay can not be managed anymore. The stable scenario presented in this section can be reached when frames associated with low requirements and frames associated with high requirements are mixed. In this case, the frame associated with low requirements leads to a high decoding duration allowing the management of the configuration delay. This scenario corresponds, for example, when a low throughput DVB-RCS standard transfer for TV reception is realized in parallel with web browsing through the LTE or WiMAX standard in a mobility context leading to dynamic decoding parameters evolution to reach both throughput and decoding performance requirements. When this stability in not ensured anymore, the number of decoding iterations has to be reduced to compensate the delay. In order to go beyond this lack of flexibility of this scenario, the next section presents a scenario where a set of frames is associated with the same parameters and requirements.

4.3.3.2 Decoding of multiple frames

In order to benefit from the possibilities offered by the UDec architecture in a multi-mode and multi-standard context decoding consecutive data frames associated to high throughput and high decoding performance in terms of FER, it is necessary to reduce the flexibility of the previously considered scenario. Indeed, the previous sections have shown that the extra configuration delay can lead to a reduction of the decoding performance in order to compensate it when consecutive frames associated to high requirements are decoded. Thus, in order to avoid the decoding performance loss, a set of frames associated to the same parameters and requirements can be consecutively decoded as shown in Figure 4.8.

Figure 4.8 illustrates a scenario where four frames (from frame (n+1) to frame (n+4)) associated to the same parameters and requirements have to be consecutively decoded. First an oversized configuration is generated for frame (n+1). Since the frame (n+2) is associated with the same parameters and requirements,

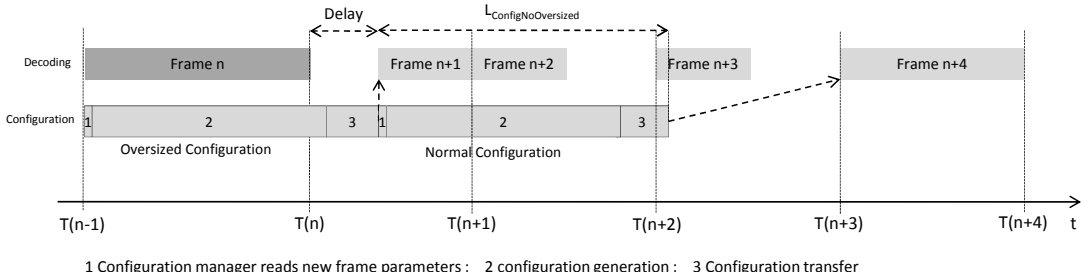


Figure 4.8: *Multi-frame configuration scenario*

the same oversized configuration can be used to decoded this frame. Thus, no extra configuration delay is inserted between the frame (n+1) and the frame (n+2). The decoding duration of the frame (n+2) is lower than $T(n+2)-T(n+1)$ since the oversized configuration is used. Since a set of frames have to be decoded with the same parameters and requirements, the *normal configuration* is generated directly after the end of the previous configuration process. In this case the system behaves as if the delay is null, so the configuration can take place at the beginning of the next frame slot. In the example of Figure 4.8, the frame (n+4) is decoded with the normal configuration at $T(n+3)$. Once the configuration process is done for the normal configuration, the rest of the set of frames can be decoded with this configuration. In the example of Figure 4.8, it is necessary to decode three frames, i.e. frame (n+1), frame (n+2) and frame (n+3) to provide the normal configuration and to compensate the configuration delay in parallel.

The number of frames that have to be decoded in order to entirely compensate the configuration delay is given by Equation 4.11.

$$N_{frames} \geq \frac{L_{ConfigNoOversized} + Delay}{T(n+2) - T(n+1)} \tag{4.11}$$

where $L_{ConfigNoOversized}$ and $T(n+2) - T(n+1)$ are the configuration process latency for the normal configuration and the maximal decoding latency associated with each frame of the set respectively.

Table 4.5 shows examples of number of frames associated with the same parameters and requirements that have to be consecutively decoded regarding a series of two configurations associated with high throughput requirements. In this table, frame (n) and frame (n+1) correspond to the example of the scenario presented in Figure 4.8. The parameters of the frame (n) define maximum decoding duration of the frame independently of both the number of decoding iterations and the level of sub-block parallelism. N_{frames} defines the number of frames associated to the same parameters as the frame (n+1) that have to be decoded to compensate the configuration delay caused by the two configuration

Parameters frame (n)		Parameters frame (n+1)				
Frame size (in bits)	Throughput (in Mbps)	Frame size (in bits)	Throughput (in Mbps)	N_{iter}	P	N_{frames}
880	120	1920	170	9	16	2
880	300	4800	150	9	14	1
1920	130	4800	360	10	29	2
4800	350	6144	360	10	29	1

Table 4.5: *Multi-frame scenario examples:* $N_{iterP=1} = 8$, $T = 10$ *and* $P_{max} = 32$

Frame size (bits)	$Throughput_{max}$ Table 4.4 (Mbps)	New $Throughput_{max}$ (Mbps)
96	6.6	34.6
480	33.2	172.9
880	60.9	317
1920	132.9	363,6
4800	332.2	363,6
6144	363,6	363,6

Table 4.6: *Estimated maximum throughput comparison with* $N_{iterP=1}$*=8,* T*=10 and* P_{max}*=32*

processes illustrated in Figure 4.8. Results show that the number of extra frames to compensate the configuration delay is quite low (from 1 to 2 in Table 4.5) and it demonstrates that this approach is interesting to guarantee the decoding performance in terms of FER with a low cost in scenario flexibility. Moreover, compared to results presented in Table 4.4 considering the restricted configuration management presented in Section 4.3.1, we observe that the throughput reached in Table 4.5 are quite greater. This is particularly true for low frame size since this configuration management allows the usage of higher level of sub-block parallelism. For instance, the maximum throughput for a 880-bit frame considering the restricted configuration management is 60.9 Mbps while it can reached 300 Mbps using the oversized configuration management.

4.4 Configuration management discussion

The Section 4.3 demonstrates that the adoption of a smart and flexible configuration management is a key feature to reach high performances in a multi-mode and multi-standard context. However, the maximum performances in terms of throughput and FER are critically dependent on the configuration latency pro-

cess. In Section 4.3, the assumed configuration generation latency is quite large (i.e. from $5\mu s$ to $14\mu s$ for a level of sub-block parallelism up to 32) compared to the configuration transfer latency offered by the configuration infrastructure ($\leq 0.446\mu s$ for a level of sub-block parallelism up to 32, cf. Table 3.9). This configuration generation time has been evaluated thanks to a non fully optimized and non parallelized C-code on a 600 MHz ARM core. This C-code is highly parallelizable since the configuration parameters specific to each ASIP, which represent the major part of the computation when the number of active ASIPs increases can be generated in parallel. Moreover, the recent processors in embedded systems such as mobile phone and tablets provide processors implementing two or more cores with a clock frequency close to 2 GHz. Considering this feature, we estimated that the configuration latency can be divided by 6 leading to a configuration generation latency from $0.83\mu s$ to $2.33\mu s$. Indeed, the increasing processor clock frequency provide a improvement factor of $\frac{2000}{600} = 3.33$ which can be multiplied by two considering a 2-core processor. Table 4.6 shows maximal throughput considering the new configuration generation latency and a restricted configuration management. These results are compared to the results from Table 4.4. The comparison results show that the reduction of the configuration generation latency has a huge impact on the maximum throughput. The maximum throughput offered by the UDec architecture by considering a maximum level of sub-block parallelism is now reached starting from 1920-bit frame. It is important to note that the improvements provided by the oversized configuration management stay true in these new conditions.

The configuration management proposed in this thesis work can be compared to the configuration management presented in [46] which proposes a configuration management where the number of decoding iterations can be reduced when the configuration latency is too big. However, contrary to the approach proposed in this thesis work, the current configuration and the next configuration are stored simultaneously in the same memory. Moreover, the current configuration is read during the decoding process. Thus, the authors of [46] have to deal with two situations: (1) the configuration memory can not contain two configurations for all cases and (2) the configuration latency can be too big to support high throughput. For the first situation, the authors propose to stop the current decoding process in order to load the rest of the configuration for the next configuration leading to a decoding performance loss in term of FER since the number of decoding iterations has been reduced. For the second situation, the authors determine a minimum number of frames that have to be decoded with the current configuration in order to provide more time for the next frame configuration process.

The configuration management proposed in this chapter differs from the work presented in [46] from different points. Table 4.7 compares the two configuration

	[46]	This work
Dynamic evolution of the number of iterations	X	√
Run-time configuration generation	X	√
Number of iterations reduction	√ (if the configuration is too large)	√ (if the configuration delay is not managed anymore)
Decoding of multiple frames	√ (When configuration latency is too large)	√ (to ensure stability)

Table 4.7: *Configuration management comparison*

management approaches. Firstly, the dynamic evolution of the number of decoding iterations regarding the level of sub-block parallelism described in Section 4.1 is taken into account. Indeed, this dynamic evolution can require more computation and thus increase the number of processors that have to be activated to respect the throughput requirement of a given configuration. Secondly, the proposed configuration management provides a high level of configuration flexibility since the configuration information is generated at run-time on a independent processor. Indeed, the storage of all the possible configurations for a standard leads to a prohibitive cost in terms of memory as shown in Section 4.2. Thirdly, the oversized configuration management proposed in Section 4.3.2 provides an efficient solution to guarantee the throughput and the decoding performance in terms of FER even if the configuration latency for the next frame becomes bigger than the decoding duration of the current frame. Moreover, it can be noticed that when the number of decoding iterations has to be reduced when an oversized configuration is searched, the performance loss impacts the next configuration and not the current configuration as in [46].

4.5 Summary

This chapter studied the configuration management of the UDec architecture implementing RDecASIPs and the configuration infrastructure presented in Chapter 3 in order to offer high throughput and high decoding performance in terms of error rate. An analysis of the dynamic evolution of the number of decoding iterations regarding the level of sub-block parallelism has been provided in order to be integrated in the configuration management of the UDec architecture. Then a

configuration management where all possible configurations are stored in a global configuration memory has been studied. Results show that such an approach is prohibitive in terms of memory cost. Thus a configuration management where the configuration information is generated at run-rime has been proposed. The proposed oversized configuration management enabling configuration latency bigger than decoding duration allows to take advantage of the possibilities offered by the UDec architecture thanks to a novel algorithm for run-time configuration solution research.

Conclusion and perspectives

Conclusion

The 21st century will be the century of connectivity. People as well as objects will share information through different connected devices. In this context, the last years have seen considerable evolutions of wireless communication standards in the domain of mobile telephone networks, local/wide wireless area networks, and digital video broadcasting. Channel coding is a key feature of a wireless standard allowing reliable data transfer. Among channel coding techniques, Turbo codes are frequently adopted to reach a very low bit error rate. The work accomplished during this Ph.D thesis is motivated by the emergence of reconfigurable multiprocessor architectures for Turbo decoding in order to provide high flexibility and high throughput considering current and future wireless communication standards requirements. Moreover, the multiplication of communication standards leads to complex scenarios where the configuration process becomes a key point in order to guarantee high performances. In fact, most of the existing related works have proposed flexible hardware platforms while trying to optimize their efficiency in terms of area, throughput, and energy consumption. Very few contributions have considered the crucial requirement of rapid dynamic configuration and the related implementations and costs. In this context, this thesis work tackles the dynamic configuration issues of multiprocessor platform for Turbo decoding in order to respect hard constraints imposed by emerging multi-mode and multi-standard scenario. For that purpose, contributions at the processing element level as well as at the system level are proposed. These contributions are illustrated with the multi-ASIP UDec architecture developed at the Electronic Department of Telecom Bretagne in Brest.

In this manuscript, the first chapter provides the basic background on Turbo codes along with its construction and decoding process. A presentation of the different parallelism levels which can be exploited in the implementation of a Turbo decoder is proposed. This chapter also provides a description of the initial UDec architecture which constitutes the starting point of this thesis work.

The set of ASIPs implemented in the UDec architecture is used for the decoding process. They are the main components that have to be configured to adapt the platform to the input frame parameters and requirements. The second chapter of this thesis work presents the contributions allowing the optimization of the dynamic configuration of the initial DecASIP processor for Turbo decoding. For that purpose, the configuration lacks of the DecASIP have been highlighted and optimizations have been proposed and implemented in order to offer an effi-

cient configuration of the DecASIP in a multi-ASIP context. The new processor named RDecASIP provides new features such as a new configuration memory organization which takes into account the multi-ASIP context of the UDec architecture, a generic program reducing the configuration load of the ASIP and a multi-configuration storage management. These optimizations lead to a configuration load reduced by 70% compared to the initial ASIP. Results show that a dedicated memory organization taking into account the multiprocessor context hugely reduces the configuration load (i.e. more than 90%) when a configuration infrastructure implementing multicast and broacast mechanisms is used. Logic synthesis results targeting 65 nm CMOS technology show that these optimizations introduce a low area overhead of 0.009 mm^2 while the decoding performance and maximum clock frequency of the RDecASIP have remained identical to the initial implementation.

The second contribution of this thesis work concerns the development of a complete configuration infrastructure for the UDec architecture. Since the initial UDec architecture has not been designed to support the dynamic configuration, the third chapter describes optimizations at the platform level in order to reach an efficient dynamic configuration management. Flexibility has been brought at the Butterfly topology NoCs level, at the ring buses level and at the platform controller level in order to adapt at run-time the number and the location of the active processing elements. The configuration process of the proposed platform consists in loading configuration information in configuration memories associated with each active RDecASIP and with the platform controller. For that purpose, a complete bus-based configuration infrastructure has been developed and validated using a SystemC/VHDL mixed simulation model. It is optimized for data memory loading using unicast, multicast and broadcast mechanisms. An FPGA prototype has been developed and results have shown that the configuration of the entire platform implementing 128 RDecASIPs can be performed in less than 10 μs providing a speedup up to 12 compared to classical approaches. ASIC synthesis results considering up to 128 RDecASIPs have demonstrated that a configuration transfer latency below $1\mu s$ is reachable providing an efficient solution in order to support dynamic configuration in the multi-mode and multi-standard scenario.

The last chapter of this manuscript addresses the configuration management issues of high throughput Turbo decoder. The limits of both pre-compiled management where all configurations are stored in a global configuration memory and restricted configuration management where the configuration information is generated and sent at run-time respecting a strict deadline fixed by the decoding duration of the current frame are studied. In order to exploit efficiently the Turbo decoder performances, the oversized configuration management is proposed. It

provides a solution in order to manage large configuration latencies by generating oversized configurations that guarantee throughput and decoding performance requirements in terms of FER. For that purpose, an algorithm for configuration solution search has been proposed. It explores configuration solutions by incrementing the level of sub-block parallelism sequentially until throughput and decoding performance requirements are met. If not, shuffled decoding mode can be enabled. In the case no configuration solution is able to respect throughput and decoding performance requirements, the number of decoding iterations is decremented in order to find a configuration solution respecting the throughput requirement only. The analysis of the results has demonstrated that, for example, the maximum throughput for a 880-bit frame considering the restricted configuration management is 60.9 Mbps when the maximum configuration latency considered is 14.446 μs while it can reach 300 Mbps using the oversized configuration management. Moreover, considering the oversized configuration management and the decoding of multiple frames associated to the same configuration, results have shown that the maximum number of extra frame to compensate the configuration delay inserted by large configuration latency is quite low, i.e. one or two regarding the scenario considered in this thesis work. This result demonstrates the interest of the proposed approach in order to guarantee the decoding performances with a low cost in scenario flexibility.

Overall, the contributions proposed in this thesis work have been implemented and validated on the UDec architecture for Turbo decoding. The analyses provided in this Ph.D manuscript have shown that the configuration latency of multiprocessor architectures can be reduced by optimizing the configuration process at the processor level, at the system level and at the configuration management level. The proposed studies have demonstrated that the optimization of a processing element by taking into account the configuration process allows the reduction of the configuration load that becomes a key feature when the number of processing elements that have to be reconfigured increases. Moreover, this configuration process must take into account the multiprocessor context. This thesis work has shown that the configuration memory organization of a processing element has a significant impact on the configuration transfer latency when it allows the usage of optimized transfer mechanisms. Considering a context where the data exchange between processing elements is a critical point of the architecture, dedicated interconnection structures are mandatory in order to ensure high speed exchanges. These interconnection structures are not designed to support configuration flows that can disturb the data transfer. To deal with this issue, an efficient and low complexity configuration infrastructure for memory loading has been presented. It has been associated to the UDec architecture to demonstrate its efficiency when an important number of memories has to be loaded during the

configuration process by taking advantage of a dedicated memory organization. Finally, the last contribution of this thesis work introduced a novel configuration management of a high throughput Turbo decoder, considering a high constrained multi-mode and multi-standard scenario, that reduces the impact of the configuration process latency on the maximum decoding performance and throughput.

Perspectives

Regarding future perspectives, several ideas can be investigated. At the RDecASIP level, the application of turbo decoding only has been considered in this work. In order to propose LDPC decoding also with a high speed configuration, the programs and the memory organization have to be analyzed for this new context. Concerning the configuration memory organization, LDPC parameters have to be studied in order to provide a dedicated memory organization merging Turbo and LDPC codes parameters. As for Turbo mode, the different programs for LDPC mode have to be optimized in order to reduce the configuration load. Since this optimization leads to a unique program per code, a solution where programs are replaced by a FSM dealing with the two modes can be studied and compared to a programmable architecture in terms of area and flexibility taking into account future component evolutions in order to support more standards and codes.

At the system level, the new UDec architecture proposed in this thesis work allows the management of the number and the location of the active RDecASIPs. However, decision methods in order to define the location of the active RDecASIPs for a given configuration have not been addressed in this work. Considering the UDec features, decision algorithm for optimizing the power consumption, the fault tolerance or the hardware degradation can be explored.

The input data flow management is also a key point to support the multi-mode and multi-standard scenario considered in this work which has to be addressed for the UDec architecture. Indeed, the current input interface and input memories associated to the ASIPs are not able to deal with more than one input data flow. Thus the input symbols of the following frame can not be loaded in the input memories before the end of the current decoding process. A simple solution consists in doubling the number or the size of the input memories in order to load input symbols during the decoding process. However the cost of such a solution when an important number of ASIPs is implemented seems prohibitive. Consequently, alternatives have to be explored.

At the configuration management level, an implementation of the proposed oversized management has to be proposed in order to evaluate the impact of the proposed algorithm on the configuration process latency. For this purpose, software, hardware and ASIP-based solution can be explored in order to propose an efficient solution for the configuration management and configuration generation processes. Furthermore, the proposed approach allowing the definition at run-time of the number of decoding iterations regarding the level of sub-block parallelism has been built regarding simulation results in DBTC. Thus, the approach has to be validated for SBTC mode also. Moreover, the approach has to be validated through detailed realistic scenarios where the probability that configuration parameters change regarding mobility speed and environment conditions (for each supported standard) and a set of running applications dealing with different standards have to be taken into account.

Finally, the contributions and methods proposed in this thesis work can be adapted to other components and architectures developed during previous works at the Electronic Department of Telecom Bretagne in Brest. Indeed, ASIP-based architecture for Turbo demapping and Turbo equalization have been recently introduced. In order to ensure an efficient dynamic configuration of a complete reception chain, the principles described in this thesis work can be adapted and optimized in order to build a dynamic reconfigurable heterogeneous multi-ASIP architecture for a complete multi-mode and multi-standard receiver.

Glossary

3GPP	3rd Generation Partnership Project
4G	Fourth Generation
ADL	Architectural Description Language
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
AWGN	Additive White Gaussian Noise
AXI	Advanced eXtensible Interface
BCJR	Bahl-Cock-Jelinek-Raviv
BER	Bit Error Rate
CABA	Cycle Accurate and Bit Accurate
CMOS	Complementary Metal Oxide Semi-conductor
DBTC	Double Binary Turbo Codes
DVB-RCS	Digital Video Broadcasting Return Channel Satellite
DSP	Digital Signal Processor
FER	Frame Error Rate
FEC	Forward Error Correction
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
FSM	Finite State Machine
GPP	General Purpose Processor
IP	Internet Protocol
HDL	Hardware Description Language
LDPC	Low-Density Parity-Check
LLR	Log-Likelihood Ratio
LTE	Long Term Evolution
LUT	Look Up Table

MAP	Maximum A Posteriori
MI	Master interface
MPSoC	Multiple Processor System on Chip
NI	Network Interface
NoC	Network on Chip
PE	Processing Element
PLB	Processor Local Bus
QPP	Quadratic Permutation Polynomial
RAM	Random Access Memory
RSC	Recursive Systematic Convolutional
RTL	Register Transfer Level
SBTC	Single Binary Turbo Codes
SDR	Software-Defined Radio
SI	Slave Interface
SIMD	Single Instruction Multiple Data
SISO	Soft In Soft Out
SNR	Signal to Noise Ratio
SoA	State of the Art
SOVA	Soft Output Viterbi Algorithm
SRAM	Static Random Access Memory
UMTS	Universal Mobile Telecommunications System
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
VLSI	Very-large-scale integration
WiMAX	Worldwide Interoperability for Microwave Access

Bibliography

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

[2] "3GPP TS 36.212: Multiplexing and channel coding, version 8.4.0," Sept. 2008.

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

[4] *IEEE Standard for Local and Metropolitan Area Networks Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems*, Std., 2006.

[5] S. Lin and D. Costello, Jr., *Error Control Coding: Fundamentals and Applications*. Prentice Hall, Englewood Cliffs, NJ., 1983.

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

[7] S. Dolinar and D. Divsalar, "Weight distributions for turbo codes using random and nonrandom permutations," in *The Telecommunications and Data acquisition Report, Tech. Rep.*, 1995, pp. 56–65.

[8] J. Hagenauer and P. Hoeher, "A viterbi algorithm with soft-decision outputs and its applications," in *IEEE Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond' (GLOBECOM)*, 1989, pp. 1680–1686 vol.3.

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

[10] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal map decoding algorithms operating in the log domain," in *IEEE International Conference on Communications*, vol. 2, 1995, pp. 1009–1013 vol.2.

[11] G. Masera, G. Piccinini, M. Roch, and M. Zamboni, "Vlsi architectures for turbo codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 3, pp. 369–379, 1999.

[12] E. Boutillon, W. Gross, and P. Gulak, "Vlsi architectures for the map algorithm," *IEEE Transactions on Communications*, vol. 51, no. 2, pp. 175–185, 2003.

[13] Y. Zhang and K. Parhi, "Parallel turbo decoding," in *Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS '04)*, vol. 2, 2004, pp. II–509–12 Vol.2.

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

[15] O. Muller, A. Baghdadi, and M. Jezequel, "Parallelism efficiency in convolutional turbo decoding," *EURASIP Journal on Advances in Signal Processing*, vol. 2010, no. 1, pp. 927–920, 2010.

[16] C. Schurgers, F. Catthoor, and M. Engels, "Memory optimization of map turbo decoder algorithms," *IEEE Transactions on Very Large Scale Integration (VLSI) System*, vol. 9, no. 2, pp. 305–312, 2001.

[17] N. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Soft-output decoding algorithms in iterative decoding of turbo codes," *tech. rep., JPL TDA Progress report*, 1996.

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

[19] J.-M. Hsu and C.-L. Wang, "A parallel decoding scheme for turbo codes," in *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (ISCAS '98)*, vol. 4, 1998, pp. 445–448 vol.4.

[20] Z. Wang, H. Suzuki, and K. Parhi, "Vlsi implementation issues of turbo decoder design for wireless applications," in *1999 IEEE Workshop on Signal Processing Systems (SiPS 99)*, 1999, pp. 503–512.

[21] D. Gnaedig, "Optimisation des architectures de dcodage des turbo-codes," Ph.D. dissertation, Institut Mines-Télécom-Télécom Bretagne-UEB, 2005.

[22] F. Viglione, G. Masera, G. Piccinini, M. Roch, and M. Zamboni, "A 50 mbit/s iterative turbo-decoder," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE) 2000*, 2000, pp. 176–180.

[23] L. Zhang and Y. Li, "Implementing and optimizing a turbo decoder on a ti tms320c64x device," in *2011 International Conference on Computational Problem-Solving (ICCP)*, 2011, pp. 401–404.

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

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

[26] F. Gilbert, M. J. Thul, and N. Wehn, "Proceedings of communication centric architectures for turbo-decoding on embedded multiprocessors," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 356–361.

[27] G. K. Rauwerda, G. J. Smit, C. R. B. van, and P. M. Heysters, "Reconfigurable turbo/viterbi channel decoder in the coarse-grained montium architecture," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2006)*. CSREA Press, June 2006, pp. 110–116. [Online]. Available: http://doc.utwente.nl/65627/

[28] J. Liang, R. Tessier, and D. Goeckel, "A dynamically-reconfigurable, power-efficient turbo decoder," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, 2004, pp. 91–100.

[29] C.-C. Wong and H.-C. Chang, "Reconfigurable turbo decoder with parallel architecture for 3gpp lte system," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 57, no. 7, pp. 566–570, 2010.

[30] J.-H. Kim and I.-C. Park, "A unified parallel radix-4 turbo decoder for mobile wimax and 3gpp-lte," in *IEEE Custom Integrated Circuits Conference (CICC '09)*, 2009, pp. 487–490.

[31] D.-S. Cho, H.-J. Park, and H.-C. Park, "Implementation of an efficient ue decoder for 3g lte system," in *International Conference on Telecommunications (ICT 2008)*, 2008, pp. 1–5.

[32] D. Wu, R. Asghar, Y. Huang, and D. Liu, "Implementation of a high-speed parallel turbo decoder for 3gpp lte terminals," in *IEEE 8th International Conference on ASIC (ASICON '09)*, 2009, pp. 481–484.

[33] 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 *2011 13th International Symposium on Integrated Circuits (ISIC)*, 2011, pp. 168–171.

[34] M. May, T. Ilnseher, N. Wehn, and W. Raab, "A 150mbit/s 3gpp lte turbo code decoder," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'10*, 2010, pp. 1420–1425.

[35] T. Ilnseher, F. Kienle, C. Weis, and N. Wehn, "A 2.15gbit/s turbo code decoder for lte advanced base station applications," in *2012 7th International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, 2012, pp. 21–25.

[36] R. Shrestha and R. Paily, "Design and implementation of a high speed map decoder architecture for turbo decoding," in *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID)*, 2013, pp. 86–91.

[37] Xilinx, Partial Reconfiguration User Guide UG702 (v14.5) . [Online]. Available: http://www.xilinx.com, April 26, 2013.

[38] S. Zhang, R. Qian, T. Peng, R. Duan, and K. Chen, "High throughput turbo decoder design for gpp platform," in *2012 7th International ICST Conference on Communications and Networking in China (CHINACOM)*, 2012, pp. 817–821.

[39] L. Huang, Y. Luo, H. Wang, F. Yang, Z. Shi, and D. Gu, "A high speed turbo decoder implementation for cpu-based sdr system," in *IET International Conference on Communication Technology and Application (ICCTA 2011)*, 2011, pp. 19–23.

[40] O. Muller, A. Baghdadi, and M. Jezequel, "Asip-based multiprocessor soc design for simple and double binary turbo decoding," in *Proceedings of Design, Automation and Test in Europe (DATE '06)*, vol. 1, 2006, pp. 1–6.

[41] H. Moussa, O. Muller, A. Baghdadi, and M. Jezequel, "Butterfly and benes-based on-chip communication networks for multiprocessor turbo decoding," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, 2007, pp. 1–6.

[42] P. Murugappa, "Towards optimized flexible multi-asip architectures for ldpc/turbo decoding," Ph.D. dissertation, Institut Mines-Télécom-Télécom Bretagne-UEB, 2012.

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

[44] T. Vogt and N. Wehn, "A reconfigurable asip for convolutional and turbo decoding in an sdr environment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1309–1320, 2008.

[45] "Ieee standard for local and metropolitan area networks part 16: Air interface for fixed and mobile broadband wireless," *IEEE Std 802.16e-2005*, 2006.

[46] C. Condo, M. Martina, and G. Masera, "VLSI Implementation of a Multi-Mode Turbo/LDPC Decoder Architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers, Early Access Articles*, 2012.

[47] ——, "A Network-on-Chip-based turbo/LDPC decoder architecture," in *Proc. of the Design, Automation and Test in Europe Conference & Exhibition (DATE)*, 2012.

[48] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn, "MAGALI: A Network-on-Chip based multi-core system-on-chip for MIMO 4G SDR," in *Proc. of IEEE International Conference on IC Design and Technology (ICICDT)*, 2010, pp. 74 –77.

[49] J. Glossner, D. Iancu, M. Moudgill, G. Nacer, S. Jinturkar, S. Stanley, and M. Schulte, "The sandbridge SB3011 platform," *EURASIP J. on Embedded Systems*, pp. 16–16, 2007.

[50] T. Limberg, M. Winter, M. Bimberg, R. Klemm, E. Matus, M. Tavares, G. Fettweis, H. Ahlendorf, and P. Robelly, "A fully programmable 40 gops sdr single chip baseband for lte/wimax terminals," in *Solid-State Circuits Conference, 2008. ESSCIRC 2008. 34th European*, sept. 2008, pp. 466 –469.

[51] J. Declerck, P. Raghavan, F. Naessens, T. Aa, L. Hollevoet, A. Dejonghe, and L. Van der Perre, "SDR platform for 802.11n and 3-GPP LTE," in *Proc. of International Conference on Embedded Computer Systems (SAMOS)*, 2010, pp. 318 –323.

[52] U. Ramacher, "Software-Defined Radio Prospects for Multistandard Mobile Phones," *Computer*, vol. 40, no. 10, pp. 62 –69, 2007.

[53] D. Lattard, E. Beigne, F. Clermidy, Y. Durand, R. Lemaire, P. Vivet, and F. Berens, "A reconfigurable baseband platform based on an asynchronous network-on-chip," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 223 –235, jan. 2008.

[54] V. Derudder, B. Bougard, A. Couvreur, A. Dewilde, S. Dupont, L. Folens, L. Hollevoet, F. Naessens, D. Novo, P. Raghavan, T. Schuster, K. Stinkens, J.-W. Weijers, and L. Van der Perre, "A 200mbps+ 2.14nj/b digital baseband multi processor system-on-chip for sdrs," in *VLSI Circuits, 2009 Symposium on*, june 2009, pp. 292 –293.

[55] O. Muller, "Architectures multiprocesseurs monopuces generiques pour turbo-communications haut-debit," Ph.D. dissertation, Institut Mines-Télécom-Télécom Bretagne-UEB, 2007.

[56] 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 *IEEE/ACM International Conference on Computer Aided Design(ICCAD 2001)*, 2001, pp. 625–630.

[57] ARM, AMBA specifications v2.0. ARM. [Online]. Available: http://www.arm.com.

[58] IBM, CoreConnect Bus Architecture. IBM Microelectronics. [Online]. Available: http://www.ibm.com/chips/products/coreconnect.

[59] Altera, Avalon bus specification: Reference manual. Altera Corporation. [Online]. Available: http://www.altera.com.

[60] Sonics network technical overview. Sonics, Inc. [Online]. Available: http://www.sonicsinc.com.

[61] Xilinx, FSL V2.0 specification. Xilinx, Inc. [Online]. Available: http://www.xilinx.com.

[62] T. Grotker, *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[63] O. Muller, A. Baghdadi, and M. Jezequel, "Exploring parallel processing levels for convolutional turbo decoding," in *Information and Communication Technologies, 2006. ICTTA '06. 2nd*, vol. 2, 2006, pp. 2353–2358.

List of publications

International conferences

V. Lapotre, M. Huebner, G. Gogniat, P. Murugappa, A. Baghdadi, J.-P. Diguet, "An efficient on-chip configuration infrastructure for a flexible multi-asip turbo decoder architecture," in *Proc. of 8th International Workshop on Reconfigurable Communicationcentric Systems-on-Chip (ReCoSoC)*, 2013.

V. Lapotre, P. Murugappa, G. Gogniat, A. Baghdadi, J.-P. Diguet, J.-N. Bazin, M. Huebner, "Optimizations for an efficient reconfiguration of an asip-based turbo decoder," in *Proc. of 2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013.

V. Lapotre, P. Murugappa, G. Gogniat, A. Baghdadi, M. Huebner, J.-P. Diguet, "A reconfigurable multi-standard asip-based turbo decoder for an efficient dynamic reconfiguration in a multi-asip context," in *Proc. of 2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2013.

V. Lapotre, P. Murugappa, G. Gogniat, A. Baghdadi, M. Huebner, J.-P. Diguet, "Stopping-free dynamic configuration of a multi-asip turbo decoder," in *Proc. of 2013 16th Euromicro Conference on Digital System Design (DSD)*, 2013.

P. Murugappa, V. Lapotre, A. Baghdadi, M. Jezequel, "Rapid Design and Prototyping of a Reconfigurable Decoder Architecture for QC-LDPC Codes," in *Proc. of 2013 IEEE International Symposium on Rapid System Prototyping (RSP)*, 2013.

V. Lapotre, G. Gogniat, J.-P. Diguet, S. Haddad, A. Baghdadi, "An analytical approach for sizing of heterogeneous multiprocessor flexible platforms for iterative demapping and channel decoding," in *Proc. of 2012 IEEE International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2012.

National conferences

V. Lapotre, P. Murugappa, G. Gogniat, A. Baghdadi, J.-P. Diguet, "Plateforme multi-ASIP reconfigurable dynamiquement pour le turbo dcodage dans un contexte multi-standard," in *Proc. of Gretsi*, 2013.

V. Lapotre, P. Murugappa, G. Gogniat, A. Baghdadi, S. Haddad, J.-P. Diguet, J.-N. Bazin, M. Huebner, "Efficient dynamic configuration of a multi-ASIP turbo decoder," in *Proc. of Colloque National du GDR SoC-SIP: Groupe de Recherche System on Chip - System in Package*, 2013.

V. Lapotre, G. Gogniat, A. Baghdadi, S. Haddad, J.-P. Diguet, J. Shield, "Management of reconfigurable multi-standards ASIP-based receiver," in *Proc. of Colloque National du GDR SoC-SIP: Groupe de Recherche System on Chip - System in Package*, 2011.

Appendix

An analytical approach for sizing of heterogeneous multiprocessor flexible platforms for iterative demapping and channel decoding

As mentioned is the last perspective of this thesis manuscript, an heterogeneous platform for Turbo demapping (also called iterative demapping) has been developed during previous work leaded at the Electronic Department of Telecom Bretagne in Brest. A contribution concerning the sizing of an heterogeneous platform integrating both Turbo demapping and channel decoding has been proposed during this Ph.D thesis. However, in order to focus the contributions on the multiprocessor platform for Turbo decoding only, we did choose not to introduce this specific work directly in the core of the manuscript.

This contribution has been presented at the 2012 International Conference on ReConFigurable Computing and FPGAs (ReConFig 2012) in a paper that can be found in the next pages. It proposes a novel analytical approach for sizing of heterogeneous multiprocessor flexible platforms for iterative demapping and channel decoding, which could be used both at design-time and run-time. Indeed, for a given communication requirement many architecture alternatives exist and selecting the right one at design-time and at run-time is an essential issue. The proposed approach defines the mathematical expressions which exhibit the number of heterogeneous cores and their features. It has been applied on a flexible multi-ASIP hardware platform for iterative demapping and channel decoding. Results analysis demonstrates a reduction of the chip area of 9.6% compared to a traditional approach where exploration is performed manually.

An analytical approach for sizing of heterogeneous multiprocessor flexible platforms for iterative demapping and channel decoding

Vianney Lapôtre, Guy Gogniat, Jean-Philippe Diguet
Université de bretagne Sud, CNRS Lab-STICC UMR 6285
Lorient, France
Email: firstname.lastname@univ-ubs.fr

Salim Haddad, Amer Baghdadi
Institut Telecom, Telecom Bretagne, CNRS Lab-STICC UMR 6285
Brest, France
Email: firstname.lastname@telecom-bretagne.eu

***Abstract*—Flexible baseband receivers gain the interest of many research efforts to enable the design of future multi-modes multi-standards terminals. A main challenge in this domain is to provide this flexibility with minimum overhead in terms of area, speed, and energy. In this regard, heterogeneous multiprocessor platforms are emerging as a promising implementation solution. However, the heterogeneity of such platforms makes it complex to find the required number of processors supporting a specific configuration (i.e. requirements level).**

This paper investigates, in this context, the significant optimization potential both *at design-time* and *at run-time* regarding the selection of the most appropriate hardware configuration of a multiprocessor platform for iterative demapping and channel decoding. A formal representation of the architectural solution space which allows designers to find the minimum hardware configuration is proposed. The proposed approach is illustrated through a flexible multi-ASIP hardware platform for iterative demapping and channel decoding.

***Keywords*-Multiprocessor, ASIP; Self-adaptation; Wireless multi-standards receiver; Platform sizing; Run-time; Design-time;**

I. Introduction

Last years have seen considerable evolutions of wireless communication standards in the domain of cellular telephone networks, local/wide wireless area networks, and Digital Video Broadcasting (DVB). Besides the increasing requirements in terms of throughput and robustness against destructive channel effects, the convergence of services in single smart terminal becomes a crucial and challenging feature. As an example, the fourth generation (4G) of cellular wireless standards aims at providing mobile broadband solution to laptop computer wireless modems, smartphones, and other mobile devices. Diverse features such as ultra-broadband Internet access, IP telephony, gaming services, and streamed multimedia will be provided.

In order to enable such advanced services at the algorithmic level, new state of the art data processing techniques have been developed and adopted in the emerging wireless communication standards. At the architecture level, many efforts are being conducted towards the design of flexible high throughput hardware platforms which can be configured to the required configuration. The overall flexibility of the radio platform can be achieved through the flexibility of individual components at transmitter side (encoder, interleaver, mapper, etc.) and at receiver side (demapper, deinterleaver, decoder, etc.). In this context, heterogeneous multiprocessors platforms [1], [2], [3] have been widely adopted. These platforms usually integrate different tiles that provide high performances and high flexibility to respect services requirements. ASIP based tiles have been adopted to provide flexible and powerful solutions. For example, in [1], an 10.8 Mbps ASIP core is used for turbo-decoding. However, the high throughput requirement of emerging services imposes the efficient exploitation of different parallelism levels. Several recent works propose multiprocessor approaches to build these tiles [4], [5], [6], [7]. In this work we investigate the sizing of a heterogeneous multiprocessor flexible platform for iterative demapping and channel decoding and propose a novel approach for efficient *design-time* and *run-time* sizing. We illustrate how for a given level of requirement several architecture alternatives with different number of processors exist. A formal representation of the architectural solution space is proposed. This formulation enables the designer to find the most efficient hardware configuration. Based on this formal representation, the architecture can be chosen both at *design-time* and at *run-time* according to an optimization objective which could be, for example, minimizing the number of processors, reducing the active area on the chip, reducing the clock frequency, etc.

The proposed approach is illustrated through a flexible multi-ASIP hardware platform for iterative demapping and channel decoding. This platform integrates two different types of ASIPs (Application-Specific Instruction-set Processor): one for demapping, called DemASIP, and the second for turbo decoding, called DecASIP. This paper presents the following contributions:

- A formal representation of the architectural solution space is proposed.
- A method to apply this formal representation at *design-time* and at *run-time* is defined.
- A use case that demonstrates the interest of the proposed method to reduce the chip area at *design-time* and the active area at *run-time* is presented and evaluated.

The rest of the paper is organized as follows. Section II provides an overview of relevant literature. Section III presents the system model and the configuration parameters. Section IV describes the proposed formal representation of the architectural solution space which allows the designer to size the platform depending on the system configuration. Section V evaluates the impact of a design-time sizing on the chip area and the impact of a run-time sizing on the active area for different receiver configurations. Finally, section VI provides a discussion on the proposed work and concludes the paper.

II. State of the Art

The high throughput requirement of emerging services imposes the efficient exploitation of different parallelism levels. In this context, multiprocessor architecture [4], [5], [6], [7] is a promising approach to reach high flexibility, high throughput and energy efficiency. In [4], an heterogeneous architecture for convolutional and turbo-decoding consisting of a dedicated 150Mbps IP block and a cluster of ASIPs is presented. The dedicated IP block is used when high throughput is required while ASIPs are used for lower throughput. Even if the authors superficially describe a multi-ASIP architecture in which the ASIPs are connected through a crossbar, the sizing of such an architecture is not addressed. The presented results are limited to two ASIPs that share a memory. In [5] and [6], the authors present a multi-ASIP platform for decoding in which the ASIPs are connected through a Network

978-1-4673-2921-7/12/$31.00

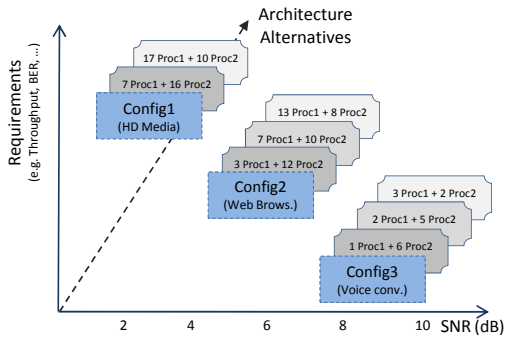


Figure 1. Usage scenario example of the considered heterogeneous multi-processor platform integrates two different types of processors that perform demapping and decoding algorithms respectively (Proc1 and Proc2).

on Chip. The high flexibility of such architectures allows dynamic reconfiguration at run-time but the run-time sizing task is not addressed. In [5], different decoding tasks can be mapped at run time on different ASIPs but no methodology to define the number of processors necessary to perform a given configuration is presented. As in [7], recent works propose to combine several functionalities, like decoding and demapping, in a multi-ASIP heterogeneous platform. Unfortunately, the sizing of such platforms is not well addressed in the literature. We assume that the designer, based on his background and simulations, has to deal with the sizing of these complex heterogeneous platforms. However this approach could provide sub-optimal solutions and decrease the sizing flexibility at run-time since all the decisions are taken at design-time. In fact, flexible hardware multiprocessor platforms for iterative demapping and channel decoding are generally designed to support a set of communication standards which correspond to some specific application needs and usage scenarios. Each usage scenario corresponds to particular requirements for example in terms of throughput, latency, error rates, and/or others. Fig. 1 gives an example of such usage scenario which corresponds to a mobile terminal supporting different services (High Definition Multimedia, Web Browsing, Voice Conversation) at different channel conditions. Hence, *at design-time*, the platform must be dimensioned to support the highest requirements while, *at run-time*, the number of processors can be chosen depending of the current level of requirements. Furthermore, for a heterogeneous multiprocessor platform, and for a specific requirement level, several architectural configurations (i.e. with a different number of each type of processors) exist (Fig. 1). The alternatives exploration and the selection of the most appropriate one is a complex task. However, it represents an important optimization room which is not investigated in the literature. In this paper we address this point and propose a formal approach to find the optimal configuration.

III. System Model and Configuration

In order to illustrate the proposed approach for sizing of heterogeneous multiprocessor flexible platform for iterative demapping and channel decoding we consider in this paper the communication system model of Fig. 2. It consists of a multi-modes advanced wireless communication system integrating convolutional turbo code, Bit Interleaved Coded Modulation (BICM), various modulation schemes, and Signal Space Diversity (SSD). A brief presentation of the system model and the considered parameters is given in this section.

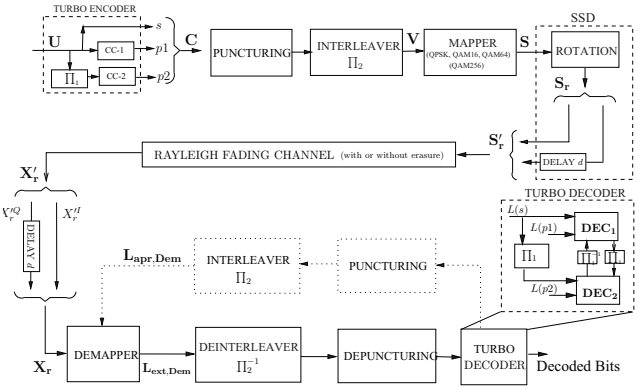


Figure 2. System model with TBICM-ID-SSD.

A. System model

Fig. 2 presents a simplified structure of the transmitter, the channel, and the receiver. On the transmitter side, information bits U which are called systematic bits are regrouped into symbols u_i consisting of k bits, and encoded with a q-binary turbo encoder. It consists of a parallel concatenation of two identical convolutional codes (PCCC). The output codeword C is then punctured to reach a desired coding rate R_c. In order to gain resilience against error bursts, resulting sequence is interleaved using an S-random interleaver Π_2. Punctured and interleaved bits denoted by v_i are then Gray mapped to complex channel symbols s_q chosen from a 2^M-ary constellation X, where M is the number of bits per modulated symbol. Applying the SSD consists of a rotation of the constellation followed by a signal space component interleaving.

At the receiver side (which is the topic of this paper), the corresponding operations to the transmitter ones are applied in reverse order. However, in order to meet the increasing requirements in terms of reduced error rates, the iterative processing is considered at two levels. The first level is at the channel decoding by adopting a turbo decoding process. The second level is between the channel decoder and the soft demapper. In fact, besides extrinsic information exchange inside the channel turbo decoder, additional extrinsic information is feedback as a priori information used by the demapper to improve the symbol to bit conversion. Thus, the receiver model, denoted as TBICM-ID-SSD, implements iterative demapping with turbo decoding.

B. Receiver configuration

A flexible software model of the whole system of the Fig.2 (transmitter, channel, and receiver) was developed. This model supports many parameters corresponding to the constellation type and modulation order, interleaving laws, turbo code type, code rate, and frame size.

Furthermore, the receiver can be configured to execute iterative or non iterative demodulation. For the case of iterative demodulation, state of the art implementations apply one turbo code iteration for each demapping iteration [8]. Thus, the number of demapping iterations (it_{dem}) is equal to the number of turbo decoding iterations (it_{dec}) in this case. This number of iterations constitutes another flexible parameter of the system model. On the other hand, for non iterative demodulation it_{dem} will be equal to 1.

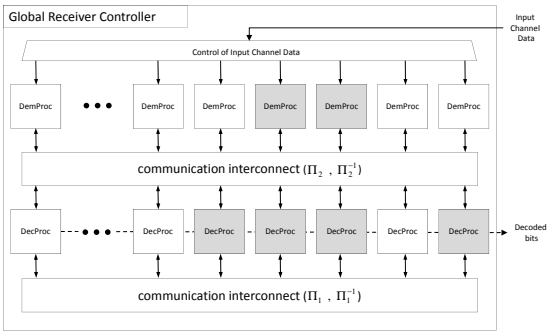


Figure 3. Generic architecture of the heterogeneous multiprocessor receiver. In this configuration, 2 DemProcs and 4 DecProcs are not used.

The configuration of these flexible parameters is generally constrained by the available communication standard, the channel condition, and the target system requirements in terms of throughput, latency, and error rate performance. The determination of their values should also take into consideration the complexity issue in order to advise the most efficient configuration (as many solutions generally exist). This task is out of the scope of this paper. However, in order to define the suitable system configurations of the usage scenario that will be considered in Section V, communication system experts were inquired and extensive simulations were conducted.

Based on the system model, the next section proposes a generic architecture model and a formal method for an efficient sizing of such platforms.

IV. Platform sizing

Multi-standards and multi-modes platforms have to be able to self-adapt when application requirements and environment evolve at run-time. A configuration is defined by the communication parameters which are chosen in accordance with the application requirements and the environment in which the communication is established. In this section we propose formal expressions which allow designers to optimize the receiver architecture by computing the required number of processors depending on each configuration. This point is essential as it enables designers to formally explore potential architectures that will meet performance constraints.

A. Generic heterogeneous multiprocessor architecture model

Fig. 3 presents the generic architecture of a flexible multiprocessor hardware platform for iterative demapping and channel decoding. The aim of this platform is to provide a flexible and dynamic solution compared to existing ones [1], [2], [3] (generally based on hardware accelerators) where designer can tune the number of resources both at design-time and at run-time. As it will be presented, such an approach allows the system meeting performance constraints without loosing its flexibility. These features will be mandatory for future communication systems. In Fig. 3, DemProc and DecProc perform demapping and decoding algorithms respectively. These two processors are characterized by their area, maximum frequency, and their performance defined by the number of cycles to demap or decode one modulated or coded symbol respectively. The platform integrates a communication interconnect that allows extrinsic information exchanges (between DecProcs themselves and between DecProcs and DemProcs). In this paper, we assume that the communication interconnect is designed for the worst case configuration in which all processors exchange data at the same time and it is congestion and conflict free.

B. Formal representation of the architectural solution space

The generic architecture of Fig. 3 can be abstracted as two components: one demapper and one decoder. Each component uses several processors in parallel to perform the frame computation exploiting sub-bloc parallelism. These two components are serially connected. The time required to process one frame (T_{syst}) corresponds to the sum of the time required by the demapper (T_{dem}) and the time required by the decoder (T_{dec}) to execute all their iterations on the frame. It can be expressed as:

$$\begin{aligned} T_{syst} &= T_{dem} + T_{dec} \\ &= N_{dem}.T_{dem/symb} + N_{dec}.T_{dec/symb} \end{aligned} \quad (1)$$

where N_{dem} and N_{dec} represent, respectively, the number of modulated and coded symbols per frame. $T_{dem/symb}$ and $T_{dec/symb}$ represent the time required by the demapper and the time required by the decoder to execute all their iterations on one modulated and coded symbol respectively. Hence, the system throughput ($D_{syst} = N_{dec}/T_{syst}$) can be expressed as below.

$$D_{syst} = \frac{D_{dem}.D_{dec}.N_{dec}}{N_{dem}.D_{dec} + N_{dec}.D_{dem}} \quad (2)$$

where D_{dem} ($= 1/T_{dem/symb}$) and D_{dec} ($= 1/T_{dec/symb}$) are the demapper and the decoder throughputs (in modulated and coded symbols, respectively). In fact, considering the code rate R_c and the number of bits per symbol M, the relation between the number of coded symbols (N_{dec}) and the corresponding number of modulated symbols (N_{dem}) can be written as follows.

$$\begin{aligned} N_{dem} &= \frac{q}{M.R_c}.N_{dec} \\ &= \alpha.N_{dec} \end{aligned} \quad (3)$$

where q depends on the coding scheme ($q = 1$ for simple binary turbo code and $q = 2$ for double binary turbo code). Introducing this expression of N_{dem} into equation (2) gives the following system throughput expression.

$$D_{syst} = \frac{D_{dem}.D_{dec}}{D_{dem} + \alpha.D_{dec}} \quad (4)$$

The throughput of the system D_{syst} is generally imposed by the application requirement. On the other hand, the throughputs of the demapper and the decoder depend on the number of processors, the number of iterations, the number of clock cycles required to process on symbol, and the clock frequency. They can be expressed as follows.

$$D_{dem} = \frac{Nb_{demProc}.F_{dem}}{it_{dem}.cycles_{dem/symb}} \quad (5)$$

where $Nb_{demProc}$ is the number of demapping processors, it_{dem} is the number of demapping iterations, $cycles_{dem/symb}$ is the number of cycles necessary to demap one symbol, and F_{dem} is the clock frequency.

$$D_{dec} = \frac{Nb_{decProc}.F_{dec}}{2.it_{dec}.cycles_{dec/symb}} \quad (6)$$

where $Nb_{decProc}$ is the number of decoding processors, it_{dec} is the number of decoding iterations, $cycles_{dec/symb}$ is the number of cycles necessary to decode one symbol, and F_{dec} is the clock frequency.

n	$Nb_{demProc}$	$Nb_{decProc}$
0.25	40	44
0.75	56	21
1	64	18
1.25	72	16
1.75	88	14

Table I
ARCHITECTURE ALTERNATIVES IN FUNCTION OF N. EXAMPLE FOR: $D_{syst} = 200$ MBPS, QPSK, $R_c = 0.5$, $it_{dem} = it_{dec} = 8, cycles_{dem/symb} = 6, cycles_{dec/symb} = 1.75$ AND 0.75 FOR THE LAST ITERATION, $F_{dec} = F_{dem} = 300MHz$

It is worth noting that the linear increase in throughput with the number of decoding processors is limited due to the sub-bloc initialization issue [9]. This limitation, which depends on the target frame size and code rate, should be considered in the platform sizing. However, this issue is not encountered in the demapping sub-bloc parallelism.

In order to establish a relation between the demapping time and the decoding time, we define the ratio n as follows.

$$n.T_{dem} = T_{dec} \tag{7}$$

From this equation we can obtain a relation between the throughputs of the demapper and the decoder:

$$\begin{aligned} n.\frac{N_{dem}}{D_{dem}} &= \frac{N_{dec}}{D_{dec}} \\ D_{dem} &= D_{dec}.n.\frac{N_{dem}}{N_{dec}} \\ D_{dem} &= D_{dec}.n.\alpha \end{aligned} \tag{8}$$

We deduce from (8) and (4) the equations which link the throughput of the system with the throughputs of the demapper and the decoder:

$$D_{dec} = \frac{n+1}{n}.D_{syst} \tag{9}$$

$$D_{dem} = \alpha.(n+1).D_{syst} \tag{10}$$

Finally, from equations (5) and (6) we can express $Nb_{demProc}$ and $Nb_{decProc}$ as follows.

$$Nb_{demProc} = C_{dem}.D_{dem} \tag{11}$$

$$Nb_{decProc} = C_{dec}.D_{dec} \tag{12}$$

where $C_{dem} = \frac{it_{dem}.cycles_{dem/symb}}{F_{dem}}$ and $C_{dec} = \frac{2.it_{dec}.cycles_{dec/symb}}{F_{dec}}$ depend on the system configuration and the processor parameters.

Replacing D_{dem} and D_{dec} by their expressions from equations (10) and (9) allows to compute the number of processors necessary for a given configuration and a given n.

$$Nb_{demProc} = C_{dem}.\alpha.(n+1).D_{syst} \tag{13}$$

$$Nb_{decProc} = C_{dec}.\frac{n+1}{n}.D_{syst} \tag{14}$$

Table I illustrates, for a given configuration, how different values of n lead to different architecture alternatives, although all of them acheiving the target throughput and supporting the target system configuration. Depending on n we observe that the architecture alternative could be quite different. For example, when n= 0,25 the architecture consists of 40 processors for demapping and 44 processors for decoding while when n=1.25, 72 processors for demapping and 16 processors for decoding are necessary. It is essential, both at *design-time* and at *run-time*, to determine the value of n which optimizes the resources use. The optimization goal depends of designers priorities and could be for example the number of processors used for each possible configuration, the total area of the chip, the clock frequency for each type of processor, etc. In this paper we extend the previous equations in order to optimize the total area of the chip at *design-time*. The same optimization can be applied at *run-time* in order to reduce the active area for the configurations performed on the platform.

C. Area optimization

Heterogeneous processors have typically different areas and performances. One main optimization objective is to determine the number of DemProcs and DecProcs in order to minimize the receiver area for a given configuration. The total area of the receiver depends on n. It can be computed using the expression below.

$$A_n = A_{dem}.Nb_{demProc} + A_{dec}.N_{decProc} \tag{15}$$

where A_{dem} and A_{dec} are the area of one DemProc and one DecProc respectively. Therefore, by putting equations (11), (12) and (8) into equation (15), A_n can be expressed as a function of C_{dem} and C_{dec}.

$$A_n = (C_{dec}.A_{dec} + C_{dem}.A_{dem}.\alpha.n)D_{dec} \tag{16}$$

On the other hand, using equation (4), D_{dem} can be expressed as:

$$D_{dem} = \frac{\alpha.D_{syst}.D_{dec}}{D_{dec} - D_{syst}} \tag{17}$$

Moreover, D_{dec} can be expressed as a function of D_{syst} and n by putting equation (8) equals to equation (17).

$$D_{dec} = \frac{D_{syst}(n+1)}{n} \tag{18}$$

Finally, A_n can be expressed as a function of n by putting the equation of D_{dec} above into equation (16).

$$A_n = \frac{a.n^2 + b.n + c}{n} \tag{19}$$

where

$$\begin{aligned} a &= C_{dem}.A_{dem}.D_{syst}.\alpha \\ c &= C_{dec}.A_{dec}.D_{syst} \\ b &= a + c \end{aligned}$$

The derivative function of the equation 19 is then computed. Only one extremum (n_{ext}) is found.

$$n_{ext} = \sqrt{\frac{c}{a}} = \sqrt{\frac{2.it_{dec}.cycles_{dec}.F_{dem}.A_{dec}}{it_{dem}.cycles_{dem}.F_{dec}.A_{dem}.\alpha}} \tag{20}$$

The second derivative function is also computed at n_{ext}. It shows a positive value corresponding to the minimum area ($A_{n_{ext}}$) of the receiver. Finally, $A_{n_{ext}}$ can be expressed as:

$$A_{n_{ext}} = a + c + 2\sqrt{a.c} \tag{21}$$

For a given configuration, n_{ext} is determined with equation (20). With the obtained value of n_{ext}, the number of DemProc and DecProc which minimizes the area can be calculated using equations (13) and (14). The number of processors is then rounded up to guarantee the throughput constraint. Note that, due to the sub-bloc initialization issue [9], the number of DecProc is limited by the maximum number of frame sub-blocs that can be extracted from the entire frame. If the number of processors determined is upper that this limit, their number is saturated in accordance to the maximum level of available parallelism and the corresponding

number of DemProc is computed with respect to the throughput requirement.

Based on the set of equations above it is now possible to analyze how the system can be tuned both at design-time and at run-time to meet performance requirements for a given configuration.

D. Design-time sizing

Platform sizing at design-time allows designers to determine the hardware configuration which minimizes the total area of the chip. This objective has been considered as it strongly impacts the cost of the chip. The design space of potential configurations is too large to allow designers to efficiently explore all possible architecture alternatives. So first, the designer needs to list the critical configurations that will be executed on the platform. These configurations will require the largest number of operations to demap and decode a frame. Then, equations (20), (13) and (14) are successively used to determine for each critical configuration the architecture alternative which minimizes the total area. Finally, The number of processors implemented on the chip is the maximum number of DemProcs and DecProcs among the different hardware configurations.

E. Run-time sizing

Platform sizing at run-time allows to determine the hardware configuration which optimizes the resources usage. Several objectives can be addressed using equations previously described. When a configuration has to be executed on the platform, the architecture alternative which optimizes the objective can be determined using the proposed equations. Configuration parameters are used to determine n_{ext} which optimizes the objective. Then, n_{ext} is used in equations (13) and (14) to compute the required number of DemProcs and DecProcs. If the number of processors required to perform the configuration is lower than the platform capacity, which is defined at design-time, unused cores can be for example switched-off as they will not be use during the execution of the current configuration.

In this section, we have proposed formal equations to explore the alternative architectures of a heterogeneous multiprocessor receiver. These equations have been applied on a particular optimization objective in order to optimize the total area used for a given configuration. We have also explained how to consider such a solution both at design-time and at run-time. The next section presents the results of this method on a typical heterogeneous multi-ASIP receiver.

V. Case study and results

A. Multi-ASIP platform

In order to apply and evaluate the proposed approach we consider the heterogeneous multi-ASIP receiver platform presented in [7]. This platform integrates two types of ASIPs which perform the main functions of iterative demodulation and turbo decoding. The first, called DemASIP [10], is dedicated to the Max-Log-MAP Demapping Algorithm. This ASIP can be used for multiple modulation schemes adopted at the transmitter side. The DemASIP provides support for BPSK to 256-QAM constellation for any mapping style with or without SSD. Depending on the modulation scheme, the time to demap one symbol evolves from 6 to 258 clock cycles. The second called DecASIP [6], performs the Max-Log-MAP Decoding Algorithm. It supports convolutional turbo codes up to eight-state double binary turbo codes or sixteen-state simple binary codes. It is able to decode a symbol in 1.75 clock cycles in turbo-demodulation scheme and in 0.75 clock cycle when only turbo-decoding is applied.

Conf.	Mod.	Throughput (Mbps)	Freq. (MHz)	it_{dem}	it_{dec}	R_c
1	QPSK	200	300	8	8	1/2
2	16-QAM	200	300	6	6	1/2
3	64-QAM	200	300	1	8	1/2
4	64-QAM	200	300	1	9	2/3

Table II
USE CASE CONFIGURATIONS

Based on this platform and in order to demonstrate the benefits of the proposed approach, a representative use case has been developed.

B. Use Case Study

The considered use case targets at the output of the channel decoder a throughput of 200 Mbps with BER performance between 10^{-5} and 10^{-6} for an SNR range from 3dB to 13dB. It can correspond to future wireless HD Media service in mobility context (e.g. during a train trip).

In order to find the suitable system parameters of the flexible communication system model of Fig. 2 with respect to this use case, extensive simulations were conducted. Results of this step (out of the scope of this paper, cf. sub-section III-B) correspond to the configurations described in Table II. The frequency of each ASIP is 300 MHz. The other parameters evolve depending on the environment conditions: Conf. 1 corresponds to severe channel conditions (i.e. lowest SNR) whereas Conf. 4 corresponds to good channel conditions (i.e. highest SNR).

C. Results

The first step of platform sizing is performed at *design-time*. For this purpose, the method described in sub-section IV-D is used to determine the best hardware sizing for the critical configurations which will be performed on the platform. For the scenario previously explained, the critical configurations are Conf. 1 and Conf. 2. They require higher number of iterations than Conf. 3 and Conf. 4 (Table II). These configurations determine the maximum number of ASIPs which needs to be implemented on the chip. Table III shows the comparison between the number of processors and the total area of the chip using the proposed method and an approach where n_{ext} is not defined. In that last case, the designer may not be able to efficiently tune the ratio between the time to demap a frame and the time to decode a frame in order to minimize the total area of the chip. A manual exploration based on the designer experience can still be performed in order to test several ratio but such an approach is time consuming and there is no guarantee to find the best one. Thus a default value of $n = 1$ is generally used. $n = 1$ means that the time to demap a frame is equal to the time to decode a frame. The last row of Table III corresponds to the number of processors that have to be implemented to support the highest requirements. It is determined by selecting the maximum number of processors needed to perform critical configurations. For this case, results show that using our formal equations allows to save 9.6% of the total area by implementing 55 DemASIP and 23 DecASIP instead of 72 DemASIP and 18 DecASIP when a default value of n is used.

Conf.	proposed method				no exploration				Gain
	n	$Nb_{demASIP}$	$Nb_{decASIP}$	Area (in mm^2)	n	$Nb_{demASIP}$	$Nb_{decASIP}$	Area (in mm^2)	(in %)
1	0.63	53	23	8.75	1	64	18	9.1	3.8
2	0.51	55	19	8.35	1	72	13	9.15	8.7
Chip	-	55	23	8.95	-	72	18	9.9	9.6

Table III
DESIGN TIME: APPLICATION OF THE PROPOSED METHOD ON THE TWO CRITICAL CONFIGURATIONS OF THE CONSIDERED CASE STUDY. $A_{dec} = 0.15mm^2$ AND $A_{dem} = 0.1mm^2$ ($90nm$ CMOS).

Conf.	n	$Nb_{demASIP}$	$Nb_{decASIP}$	Active area (in mm^2)
1	0.63	53	23	8.75
2	0.51	55	19	8.35
3	0.91	29	9	4.25
4	1.1	24	9	3.75

Table IV
RUN TIME : APPLICATION OF THE PROPOSED METHOD.$A_{dec} = 0.15mm^2$ AND $A_{dem} = 0.1mm^2$ ($90nm$ CMOS).

Once platform sizing performed, various required configurations (Table II) can be selected at run-time. The number of ASIPs that the configuration mode requires can be calculated at run-time on a GRC processor (Global Receiver Controller, Fig. 3). As the complexity of the proposed approach is very low (constant time), it allows a very efficient analysis of the best configuration at run-time. This point will be mandatory to meet real-time constraints for future adaptive communication systems. Once a new configuration has been computed the whole platform can be reconfigured.The reconfiguration mechanism itself is out of the scope of this paper but the general schedule can be sketched. The ASIPs that will be used to perform the new configuration can be loaded with appropriate parameters and program whereas the rest of the ASIPs will be idle. Once the right configuration is available the computation starts. Depending on the context the unused ASIPs can be for example powered down to reduce the total power consumption. Table IV shows the number of ASIPs necessary to perform the different configurations using our approach to reduce the active area. Results demonstrate a significant reduction of the active area when configuration corresponding to low requirement are performed. For example, in the case of Conf. 4, only 3.75 mm^2 of the chip have to be activated while 8.75 mm^2 are necessary for the highest requirements corresponding to Conf. 1. Such an approach allows to build optimization strategies for example to tune power consumption or to minimize platform aging.

VI. FINAL DISCUSSION AND CONCLUSIONS

Heterogeneous multiprocessor platforms for iterative demapping and channel decoding provide high performance and high flexibility to perform several configurations. Moreover, they provide promising solutions to be integrated in future flexible baseband receivers. Unfortunately, the first degree of flexibility of a multiprocessor system (i.e. the number of processors used for a given configuration) is currently not taken into account. The platforms are generally statically sized at design-time to reach a given maximum requirement and used, at run-time, without changing the architecture configuration. In this context, the proposed work provides an efficient method for platform sizing which could be used both at design-time and run-time. Depending on the actual requirements, this method allows a dynamic sizing at run-time which optimizes the resources management of the platform.

In this paper we propose an approach for efficient sizing of heterogeneous flexible multiprocessor for iterative demapping and channel decoding. In fact, for a given communication requirement many architecture alternatives exist and selecting the right one at design-time and at run-time is an essential issue. The proposed approach defines the mathematical expressions which exhibit the number of heterogeneous cores and their features. It has been applied on a flexible multi-ASIP hardware platform for iterative demapping and channel decoding. Results analysis demonstrates a reduction of the chip area of 9.6% compared to an approach in which alternative architectures presented in this paper are not explored. Future work targets the model extension with more functionalities, like equalization, and the application of the proposed sizing approach on a dynamic reconfigurable platform and to build optimization strategy to dynamically adapt the configuration.

REFERENCES

[1] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn, "MAGALI: A Network-on-Chip based multi-core system-on-chip for MIMO 4G SDR," in *Proc. of IEEE International Conference on IC Design and Technology (ICICDT)*, 2010, pp. 74 –77.

[2] U. Ramacher, "Software-Defined Radio Prospects for Multistandard Mobile Phones," *Computer*, vol. 40, no. 10, pp. 62 –69, 2007.

[3] J. Declerck, P. Raghavan, F. Naessens, T.V. Aa, L. Hollevoet, A. Dejonghe, and L. Van der Perre, "SDR platform for 802.11n and 3-GPP LTE," in *Proc. of International Conference on Embedded Computer Systems (SAMOS)*, 2010, pp. 318 –323.

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

[5] T. Vogt, C. Neeb, and N. Wehn, "A reconfigurable multi-processor platform for convolutional and turbo decoding," in *Proc. of International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2006, pp. 16–23.

[6] P. Murugappa, Al-Khayat R., A. Baghdadi, and M. Jézéquel, "A Flexible High Throughput Multi-ASIP Architecture for LDPC and Turbo Decoding," in *Proc. of Design, Automation and Test in Europe Conference & Exhibition (DATE)*, 2011.

[7] A. R. Jafri, A. Baghdadi, and M. Jezequel, "FPGA Prototype of Flexible Heterogeneous multi-ASIP NoC-based Unified Turbo Receiver," in *University Booth of the Design, Automation and Test in Europe Conference & Exhibition, DATE'11*, 2011.

[8] S. Haddad, A. Baghdadi, and M. Jézéquel, "Reducing the Number of Iterations in Iterative Demodulation with Turbo Decoding," in *Proc. of International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2011.

[9] O. Muller, A. Baghdadi, and M. Jézéquel, "Parallelism Efficiency in Convolutional Turbo Decoding," *EURASIP Journal on Advances in Signal Processing*, 2010.

[10] A. R. Jafri, A. Baghdadi, and M. Jezequel, "ASIP-Based Universal Demapper for Multiwireless Standards," *IEEE Embedded Systems Letters*, vol. 1, no. 1, pp. 9–13, 2009.

Résumé

Les travaux de thèse présentés dans ce manuscrit s'inscrivent dans le cadre de la conception des systèmes de communication sans fils. En effet, depuis plusieurs années, les standards de communication dans le domaine des réseaux téléphoniques mobiles, des réseaux sans fils locaux et étendus ainsi que des réseaux de diffusion de vidéo numériques ont fortement évolués. Ces évolutions ont notamment imposé une augmentation significative du débit et de la robustesse des communications vis à vis des effets de l'environnement sur les canaux de communication. Face aux nombreux standards devant être gérés par les appareils mobiles, la convergence des services au sein des terminaux devient un enjeu crucial. Par exemple, la 4ème génération (4G) de standards pour la communication sans fils à haut débit a pour objectif de fournir des solutions pour les modems d'ordinateurs portables, les smartphones, ainsi que tout autre appareil mobile communicant. Diverses fonctions comme l'accès internet haut débit, la téléphonie sur IP, les jeux en ligne, et le multimédia en streaming seront alors disponibles. De nouveaux algorithmes ont ainsi été développés et validés afin de permettre la mise en oeuvre de ces nouveaux services en vue de leur intégration dans les standards de communication sans fils émergents. Au niveau architectural, de nombreux efforts ont également été fournis pour réaliser de nouvelles plateformes offrant des débits importants et une grande flexibilité permettant notamment une configuration dynamique de la plateforme afin de sadapter aux conditions d'exécution et à la demande des utilisateurs. Pour atteindre ce niveau de performance et de flexibilité, l'équipe I.A.S (Interaction Algorithme Silicium) du laboratoire Lab-STICC a développé un Turbo-décodeur multistandard et multiprocesseur à base de processeurs ASIP (Application Specific Instruction Set Processor) nommé DecASIP. Ces précédents travaux ont démontré l'intérêt de l'utilisation d'une architecture multiprocesseur pour atteindre un haut degré de performance et de flexibilité. Toutefois, l'aspect reconfiguration dynamique de la plateforme n'avait pas été abordé. Ces travaux de thèse s'articulent donc autour de cette plateforme et ont pour but de développer un récepteur multistandard dynamiquement reconfigurable pour les futurs standards de communication. Ces travaux sont divisés en plusieurs étapes afin d'atteindre cet objectif :
La première étape a été l'étude du processeur DecASIP afin d'optimiser sa conception dans le cadre d'un système multiprocesseur reconfigurable. Cette étape a donné lieu à une nouvelle spécification intégrant une réorganisation du stockage des paramètres de configuration. Cette première contribution a permis d'optimiser les performances de reconfiguration du DecASIP. Une nouvelle implémentation du DecASIP optimisé a également été proposée.
La seconde étape a eu pour but de définir une infrastructure de communication dédiée à la reconfiguration. Cette deuxième contribution a permis d'optimiser le chargement des nouvelles configurations et le contrôle des DecASIP. Pour cela, une approche basée sur une architecture de bus unidirectionnel pipeliné de faible complexité et offrant des mécanismes de multicast et de broadcast a été proposée. Cette solution permet le transfert d'une configuration pour 128 processeurs avec une latence inférieur à la microseconde.
Enfin, la dernière étape des travaux de thèse a été l'étude d'une politique de management de la plateforme afin d'adapter ses paramètres en fonction des données recueillis sur l'environnement et sur l'application exécutée. Cette dernière contribution a abouti au développement d'une approche permettant de supporter la reconfiguration dynamique de la plateforme dans le cas de scénarios à fortes contraintes de débits et de taux d'erreur binaire où chaque trame ou groupe de trames de données est associé à une configuration particulière.

Les résultats de ces travaux permettront au laboratoire de proposer un démonstrateur de Turbo-décodeur dynamiquement reconfigurable respectant les besoins des futurs standards de communication en termes de débit, de correction d'erreurs, et de flexibilité. Un tel démonstrateur permettra de tirer profit du savoir-faire du Lab-STICC au niveau des algorithmes de décodage, des architectures multiprocesseurs, de la reconfiguration dynamique et de l'auto-adaptation.

Abstract

Recent years have seen a huge evolution of wireless communication standards in the domains of mobile phone, local and wide area networks and video broadcasting. These evolutions aim at increasing the requirements in terms of throughput, robustness against destructive channel effects and convergence of services in a smart terminal. As an example, the fourth generation (4G) of cellular wireless standards aims at providing mobile broadband solution to laptop computer wireless modems, smartphones, and other mobile devices. Diverse features such as ultra-broadband Internet access, IP telephony, gaming services, and streamed multimedia are provided. In order to enable such advanced services at the algorithmic level, new state of the art data processing techniques have been developed and adopted in the emerging wireless communication standards. At the architecture level, many efforts are being conducted towards the design of flexible high throughput hardware platforms which can be configured to the required configuration. In order to reach high flexibility, the I.A.S. (Algorithm Silicon Interaction) team of the Lab-STICC laboratory has developed an Application Specific Instruction Set Processor (ASIP) based multi-standard multiprocessor Turbo decoder. This architecture is based on the DecASIP processor. Previous work provides an efficient way to reach the high performance and high flexibility requirements of emergent standards. However, dynamic reconfiguration aspect of the architecture has not been addressed. In this context, this Ph.D work targets the development of a dynamically reconfigurable multiprocessor Turbo decoder for future communication standards. For that purpose, this thesis work is divided in several steps:
The first step consists on the study of the initial processor architecture in order to propose optimizations in a multiprocessor context. This step leads to a new implementation of the DecASIP processor integrating a new configuration memory organization in order to reduce the configuration transfer latency.
The second step leads to the development of a configuration infrastructure allowing an efficient and high speed configuration transfer for the ASIPs and the controller of the platform. The proposed approach is based on a low complexity unidirectional pipeline bus implementing optimized transfer mechanisms such as multicast and broadcast. This configuration infrastructure provides an efficient solution in order to transfer an entire configuration for 128 processors in less than one microsecond.
Finally, the last step of this thesis work concerns the development of a configuration management of the proposed platform in order to adapt the configuration parameters regarding the environment evolution and the application requirements. This step leads on an approach allowing the support of dynamic configuration of the platform in the context of highly constrained scenario in terms of throughput and error rate performances where each frame or group of frames is associated to a specific configuration.

This thesis work will allow the laboratory to present a prototype of a dynamically reconfigurable Turbo decoder respecting future communication standards requirements in terms of flexibility, throughput and error rate performances. Such a contribution gathers the skills present in the Lab-STICC laboratory at the decoding algorithm, multiprocessor architecture, dynamic reconfiguration and self-adaptation levels in a single prototype.

n d'ordre : 000000000

Université de Bretagne Sud

Université de Bretagne Sud