Université d'Ottawa · University of Ottawa

PERMISSION DE REPRODUIRE ET DE DISTRIBUER LA THÈSE

PERMISSION TO REPRODUCE AND DISTRIBUTE THE THESIS

NOM DE L'AUTEUR / NAME OF AUTHOR:	TONG, Yanhui
ADRESSE POSTALE / MAILING ADDRESS:	48 KEIGHLEY CIRCLE OTTAWA ON K2K3H8
GRADE / DEGREE:	**ANNÉE D'OBTENTION / YEAR GRANTED**
M.A.Sc. (Electrical Engineering)	2003
TITRE DE LA THÈSE / TITLE OF THESIS:	VHDL IMPLEMENTATION OF TURBO CODES

L'auteur permet, par la présente, la consultation et le prêt de cette thèse en conformité avec les règlements établis par le bibliothécaire en chef de l'Université d'Ottawa. L'auteur autorise aussi l'Université d'Ottawa, ses successeurs et cessionnaires, à reproduire cet exemplaire par photograhie ou photocopie pour fins de prêt ou de vente au prix coûtant aux bibliothèques ou aux chercheurs qui en feront la demande.

Les droits de publication par tout autre moyen et pour vente au public demeureront la propriété de l'auteur de la thèse sous réserve des règlements de l'Université d'Ottawa en matière de publication de thèses.

The author hereby permits the consultation and the lending of this thesis pursuant to the regulations established by the Chief Librarian of the University of Ottawa. The author also authorizes the University of Ottawa, its successors and assignees, to make reproductions of this copy by photographic means or by photocopying and to lend or sell such reproductions at cost to libraries and to scholars requesting them.

The right to publish the thesis by other means and to sell it to the public is reserved to the author, subject to the regulations of the University of Ottawa governing the publication of theses.

N.B. LE MASCULIN COMPREND ÉGALEMENT LE FÉMININ

May 8, 2003

DATE

Yanhui Tong

(AUTEUR) SIGNATURE (AUTHOR)

Université d'Ottawa · University of Ottawa

Université d'Ottawa • University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

TONG, Yanhui

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

VHDL Implemetiation of Turbo Codec

Jean-Yves Chouinard and Tet Yeap

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

C. D'Amours

Q.J. Zhang

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

VHDL IMPLEMENTATION OF TURBO CODEC

by

Yanhui Tong

-- A thesis submitted to the
School of Graduate Studies and Research in
Partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa
Ottawa, Ontario

April, 2003

Abstract

Turbo coding is one of the most significant achievements in coding theory during the last decade. It has been shown in the literature that transmission systems employing turbo codes could achieve a performance close to the Shannon limit. Turbo decoding is the major contributor to the overall complexity of turbo coding. Therefore, the challenge is to implement turbo coding in various communications systems at affordable decoding complexity using current VLSI technology.

Four different turbo decoding algorithms were investigated in this thesis. Comparisons on both their performances and implementation complexities were performed. Log-MAP based turbo decoding offers the best compromise among the different turbo decoding algorithms. A Register-Transfer-Level (RTL) fixed-point turbo decoder model based on Log-MAP algorithm was designed and simulated using VHDL as the hardware description language. The RTL model was verified by comparing its simulation results with those obtained from a behavioral model of the same turbo decoder written in C language.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete the thesis. First of all, I would like to thank my supervisors, Dr. Tet Hin Yeap and my co-supervisor Dr. Jean-Yves Chouinard, who gave me invaluable support, constant guidance and encouragement in all the time of research and writing of the thesis. They pointed out areas that needed more explanation, suggesting alternative presentations, wording in the thesis, which improved the accuracy and presentation of the text. It was a great pleasure to me to conduct this thesis under their supervision.

I would also like to thank Dr. Qijun Zhang and Dr. Claude D'Amours for their inputs as part of my committee.

I am grateful to my family for their support and understanding when I worked on the thesis. Especially to my husband Liang, not only for his love, support and encouragement, but also for spending so much time in reviewing the thesis and providing helpful feedback and many interesting technical discussions.

Table of Contents

List of Figures

List of Tables

Table of Acronyms

ACS	Add-Compare-Select
ACSO	Add-Compare-Select-Offset
APP	*A Posteriori* Probability
ASIC	Application Specific Integrated Circuits
AWGN	Additive White Gaussian Noise
BCJR	Bahl, Cocke, Jelinek, Raviv Algorithm
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
CC	Constituent Code
CPU	Central Processing Unit
FEC	Forward Error Correction
FPGA	Field-Programmable Gate Array
LLR	Log Likelihood Ratio
MAP	Maximum *A-Posteriori*
ML	Maximum Likelihood
PC	Personal Computer
PCCC	Parallel Concatenated Convolutional Code
PSD	Power Spectral Density
RSCC	Recursive Systematic Convolutional Code
RTL	Register Transfer Level
SISO	Soft-Input/ Soft-Output
SNR	Signal to Noise Ratio
SOVA	Soft Output Viterbi Algorithm

TCM	Trellis-Coded-Modulation
VA	Viterbi Algorithm
VHDL	VHSIC Hardware Description Language
VSS	VHDL System Simulation (Synopsys)

List of Symbols

C	Channel capacity
d	Information bit
E_b	Energy per information bit
E_s	Energy per information symbol
f_c	Correction function
$g(D)$	Generator polynomial
$G_R(D)$	Generator matrix for recursive convolutional code
K	Number of information bits transmitted per codeword
L	Log-likelihood ratio
L_{apri}	A priori information
L_c	Channel information
L_e	*Extrinsic* information
M	Memory
N	Total number of bits transmitted per codeword
N_0	Power spectral density of AWGN channel
R	Transmission rate in information bits per channel use
R_b	Data transmission rate
R_c	Code rate
W	Channel bandwidth in Hz
x^{1p}	Parity information generated by encoder 1
x^{2p}	Parity information generated by encoder 2
x^s	Systematic information for encoder
Y	Received symbol sequence

y^{1p}	Received parity information for decoder 1
y^{2p}	Received parity information for decoder 2
y^{s}	Received systematic information
α	Forward path metric
β	Reverse path metric
γ	Branch metric
γ_b	Signal to noise ratio per information bit (E_b/N_0)
γ_c	Signal to noise ratio per channel bit (E_b/N_0)
π	Interleaver
π^{-1}	De-interleaver
τ	Interleaver delay

Chapter 1

Introduction

1.1 Motivation

In 1948, Claude E. Shannon proved that it is possible to transmit information with arbitrarily high reliability provided that the rate of transmission, R, does not exceed a certain value, C, known as Shannon capacity or Shannon limit [1]. For an Additive White Gaussian Noise (AWGN) channel with a single-sided noise Power Spectral Density (PSD), N_0 (W/Hz), the channel capacity is calculated as,

$$C = \frac{1}{2}\log_2[1+2R\cdot\frac{E_b}{N_0}] \quad \text{(bits /channel use)}. \tag{1.1}$$

where E_b is the average transmission energy per information bit and R is the transmission rate in information bits per channel use. The following equation is derived from (1.1),

$$\frac{E_b}{N_0} \geq \frac{2^{2R}-1}{2R} \tag{1.2}$$

This equation tells that, for a transmission with R information bits per channel use, reliable transmission is only possible when the signal-to-noise ratio per information bit is larger than $(2^{2R}-1)/2R$. This value is termed as Shannon limit in this thesis and is used as the indication of how far the turbo coded system performs from the Shannon limit.

To approach the theoretical limit of the transmission rate, Forward-Error-Correction (FEC) codes are used to mitigate the adverse effects on transmission performance

introduced by noise and imperfect channel response, by adding redundancy and memory to the transmission sequence. For block code, it was proved that reliable transmission could be achieved by using FEC with unlimited code word length. However, the decoding complexity increases rapidly with code word length too, which puts a strict limit on the feasible channel coding schemes. Serial concatenation was introduced to efficiently increase the code word length of the channel coding by concatenating two or more short component codes into one long code. The effective code word is the multiplication of the component codes. By applying the suboptimal multiple-stage decoding, the overall decoding complexity is only the sum of the component codes. Even with this serial concatenation, system design with channel coding usually aimed at achieving cutoff rate instead of Shannon limit [2], before the introduction of turbo code in 1993.

Turbo code is one of the most significant achievements in coding theory during the last decade. By concatenating two simple convolutional (or block) codes in parallel, it can achieve the transmission performance a few tenths of a dB from Shannon limit in an AWGN channel [3], [4]. More importantly, by employing a sub-optimal iterative decoding structure and soft-in/soft-out (SISO) *a posterior* probability (APP) decoding algorithm, the near-Shannon limit performance is achieved at a feasible decoding complexity. Due to its exceptional performance advantage, turbo code has been employed or proposed in several transmission systems, such as CDMA2000 and next generation ADSL systems.

While research on applying turbo coding techniques to various applications is being carried out all over the world, implementation of turbo encoder/decoders becomes a more and more interesting topic. Although the complexity of turbo decoding is not so high as to make the implementation impossible, the SISO decoding algorithms are still very complex compared with other FEC decoding algorithms, such as Viterbi Algorithm (VA). Besides, turbo codes with good performance normally introduce a long encoding/decoding delay because of the long interleaver in both encoders and decoders. For delay sensitive applications, i.e. real-time applications, this delay must be kept very low. However, the delay is usually reduced at the cost of the performance degradation. Therefore, to implement an efficient turbo encoder/decoder with low complexity, short

delay and insignificant coding performance degradation is currently a very challenging issue.

To implement an efficient turbo decoder, a proper APP decoding algorithm has to be selected first. The original turbo decoder consists of multiple *maximum a posteriori* decoders (MAP), which involves a large amount of multiplications, exponentials and logarithm calculations. These mathematical operations are not expected in a practical turbo decoder design. Several computation-efficient suboptimal APP decoding algorithms were proposed, which greatly reduce the decoding complexity with slight performance degradation. These include Max-Log-MAP, Log-MAP and Soft-Output Viterbi Algorithm (SOVA). The Max-Log-MAP and Log-MAP algorithms are simplified MAP decoding algorithms performed in logarithm domain. SOVA is a variation of traditional Viterbi algorithm that can output the soft reliability information as the decoding results. In this thesis, performance of turbo decoders with these different SISO decoding algorithms are investigated and verified by computer simulations. Simulation of turbo decoding with MAP algorithm is also performed, and these results serve as the reference of the best achievable performance. The implementation complexities of these SISO algorithms are first analyzed and compared in terms of the number of required mathematical operations. Later, the average execution time of each SISO decoding on the same simulation platform is obtained from simulation as another benchmark for complexity comparison. Although the second comparison can only be used as a good indication since no implementation optimizations were taken into account, it gives a pretty good indication of the calculation complexities of the SISO algorithms. With the performance and complexity comparisons obtained, the best APP algorithm among the most popular decoding algorithms can be selected for our turbo decoder implementation.

Another major objective of this thesis is to investigate various issues in practical turbo decoder implementation and to find the best solutions (or tradeoffs). The fundamental issues include:

- Since turbo decoding involves large amount of mathematical computations, what is the desired data type for these computations, fixed-point or floating-point?
- Once a data type is selected, what is the minimum data word length that guarantees a satisfactory performance with a minimum complexity?

- How to avoid the computation overflow during the decoding process?
- What are the tradeoff options in a turbo decoder design? How do these tradeoffs affect the implementation?

In this thesis, we investigate these issues and use the conclusions to help determine a proper strategy of the turbo decoder implementation.

The complicated turbo decoding procedure makes it necessary to perform custom hardware design to meet a specified decoding speed. The custom design can be implemented in both Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA), where the first one provides the highest customization and the latter provides high flexibility and requires less design effort. No matter what hardware implementation solution we choose, a hardware description language is used to performance design description, design simulation, design verification, and design netlist generation. Very-High-Speed-Integrated-Circuit Hardware-Description-Language (VHDL) is now becoming increasingly popular as a way to design complex digital electronic circuits. In this thesis, we perform the turbo decoder implementation using VHDL as the design and simulation tool.

1.2 Thesis Contributions

This thesis presents the hardware implementation of a turbo codec. The design is performed and simulated using VHDL, and is verified by comparing the VHDL simulation results to C-program simulation results.

- In this thesis, C language simulations of turbo coding with MAP, Max-Log-MAP, Log-MAP and SOVA decoding algorithms in a BPSK digital transmission were performed to validate the literature research results. The performance comparison was given based on these simulation results. The computational/time complexity comparison of different decoding algorithm was also performed. These analyses were used to benchmark different decoding algorithms, so that a practical compromise can be made to select the best SISO decoding algorithm in the turbo decoder implementation.
- Some of the critical design issues in turbo decoder implementation were pointed out as (but not limited to) the internal computation data type, the data word length, the

quantization level requirement and the path metric normalization. A modified path metric normalization method was proposed to minimize the additional hardware complexity and additional delay in the Log-MAP decoder. Discussions on these design issues were presented in details and the conclusions were applied to the later turbo decoder implementation.

- An area-efficient turbo decoder was implemented and verified using VHDL as the design entry and design simulation tool. Portability and flexibility design are two major considerations in the turbo decoder implementation. The turbo decoder could be tuned to fulfill different requirements with minor modifications in the VHDL code.
- The C simulation with dynamic range limitation and the same path metric normalization method was performed to verify the VHDL turbo code implementation.

1.3 Thesis Organization

This thesis is organized as follows. In Chapter 2 we present a general introduction to turbo code and its important components, including turbo encoder, interleaver and turbo decoder. Theoretical backgrounds of several SISO decoding algorithms are discussed. Based on these discussions, their implementation complexities (in the number of various computation operations per decoded bit) are estimated and compared. Chapter 3 presents the floating-point simulation results. Simulations of turbo codes with these SISO algorithms introduced in Chapter 3 are all performed and their performances are compared. The computation complexity of turbo codes with these different SISO algorithms are compared using their simulation program execution durations on the same platform. This gives a further indication of their relative implementation complexity. A turbo codec (turbo encoder and turbo decoder) is designed in Chapter 4. In this chapter, the critical issues of turbo decoder implementation are discussed and their conclusions are presented. Based on these conclusions, an area-efficient turbo decoder is designed and implemented using VHDL. Register-Transfer Level (RTL) simulation and verification are performed on both turbo encoder and turbo decoder. A floating-point simulation of turbo decoding using Log-MAP algorithm is performed again with internal data dynamic range limits posed by the turbo decoder implementation. The simulation results are used to verify the VHDL implementation of the turbo decoder. Further considerations of turbo

codec implementation are also discussed in this chapter. Chapter 5 gives the conclusions and presents suggestions for future work.

Chapter 2

Turbo Codes

2.1 Turbo Codes Overview

This section gives a general introduction of turbo codes and the most important components, including turbo encoder, interleaver and turbo decoder.

2.1.1 Turbo Code Encoder

A typical turbo code, also known as Parallel Concatenated Convolutional Codes (PCCC), consists of two recursive convolutional codes separated by an interleaver. A generic encoding structure of a binary turbo code with two identical rate-1/2 Constituent Codes (CCs) is shown in Figure 2-1. In this diagram, the bit stream is divided into blocks in a turbo coded transmission system. The turbo code encoder processes a block of K bits each time. The first CC takes the current block as input and produces K parity bits. An interleaving operation, however, is applied to the original information bit block before it is input into the second CC, which then produces another block of K parity bits from the interleaved version of input information bits. Typically, the output of turbo encoder is the information bits followed by their parity bits from the first CC and the parity bits generated by the second CC. Hence, there are $3K$ output bits from the turbo encoder, which gives a code rate of 1/3. The parity bits from the two CC's can be punctured alternatively to obtain a code rate of 1/2 at some performance loss. High density

puncturing schemes could further be applied to achieve higher code rates, at even more performance degradation.

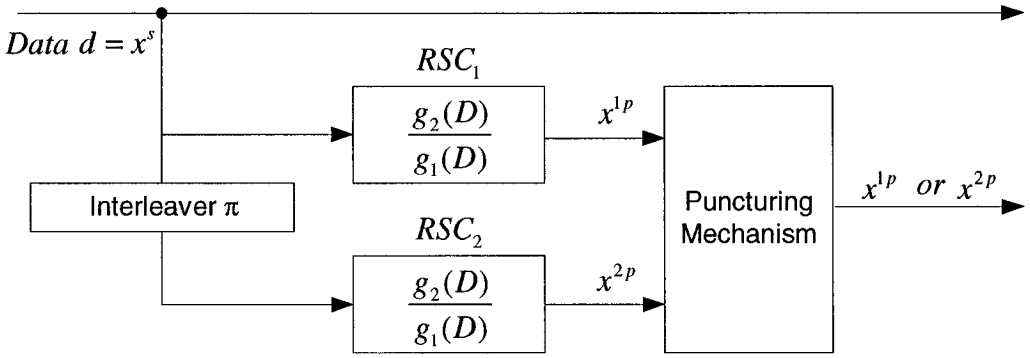


Figure 2-1 – Typical structure of turbo encoder.

The most important components in a turbo encoder are the CC's and the interleaver. Puncturing can be implemented quite easily, which will not be further discussed.

It has been shown in the literature [5] that using Recursive Systematic Convolutional Code (RSCC) provided with better distance properties. The basic structure of an RSCC with generator polynomials $G_R(D) = [1, g_2(D)/g_1(D)]$ is shown in Figure 2-2. It is also pointed out in [5] that the better is the RSCC, the better the turbo code performance is. The RSCC used in [3] has been proved by Benedetto to be optimal among rate 1/2, 16-state convolutional codes used in turbo codes [6]. Therefore, it is chosen in the turbo encoder and decoder implementations in this thesis.

2.1.2 Interleavers in Turbo Codes

Interleavers are used to both increase the overall minimum distance of turbo code to provide an interleaving gain and de-correlate the *extrinsic* information at the decoder [7]. The interleaver length is the main factor to determine the interleaver gain. It is shown in [5] that the BER of turbo-coded system is inversely proportion to the interleaver length. The second function of the interleaver requires a randomness of interleave pattern, which is also very important to the turbo codes' performance. Various interleaver design methods were proposed in the literature. In this thesis, the popular *S*-random interleavers

are used. *S*-random interleavers are actually pseudo-random interleavers with distance constraints [7]. The generation of these interleavers mainly depends on computer searching.

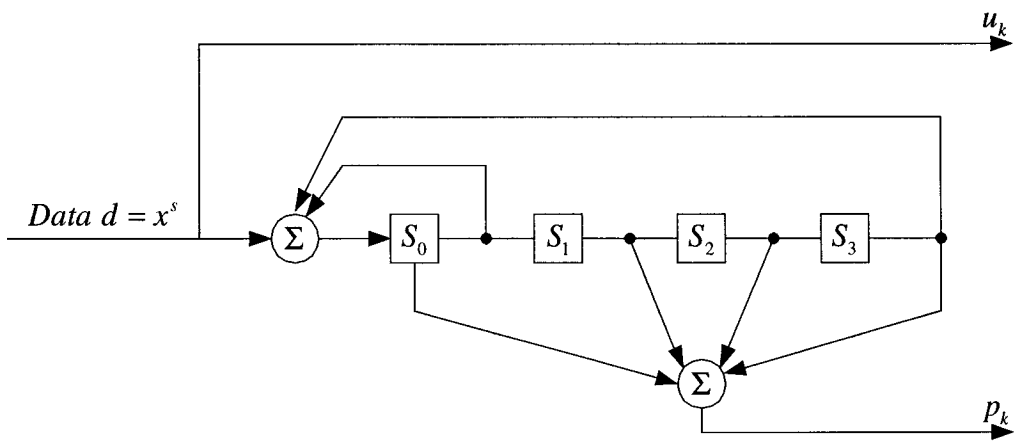


Figure 2-2 – Recursive 16-state systematic encoder with code generators, (g_0, g_1) = (31, 27).

While the interleaver is critical for the performance, it introduces end-to-end system delay. The delay introduced by the interleaving process is usually proportional to the interleaver length. Since the turbo decoding performance (BER) is inversely proportional to the interleaver length, there is a compromise between the performance and interleaving delay. Assuming that the interleaver introduces a delay of $K/2$ bits, for a transmission of R bits/second, the delay caused by this interleaving process is calculated as $\tau = K/2R$ seconds.

2.1.3 Turbo Code Decoder

Theoretical performance analysis of turbo codes always assumes using a Maximum-Likelihood (ML) decoder at the receiver. However, ML decoders are often too complex to be implemented for turbo decoding. Therefore, iterative decoding is performed instead for turbo codes. Although iterative decoding is sub-optimum in the sense of achieving maximum likelihood decoding results, it has been shown by simulation to approach ML performance after a large enough number of iterations.

In an iterative decoder, the several constituent decoders successively generates an improved decision of transmitted data. The general decoding procedure can be summarized as a three-phase process:

- Use the received channel data to compute the common and private soft channel information. Initialize the iterative information to the *a priori* values.
- Repeatedly update, for the first and second decoder, the iterative information by updating the *a priori* values. The computation involves branch path metrics (γ), forward path metric (α) calculation, and backward path metric (β) calculation.
- The APP measure for the message symbol is computed from α, β and γ.

The structure of turbo code decoder corresponding to the turbo code encoder shown in Figure 2-1 is plotted in Figure 2-3.

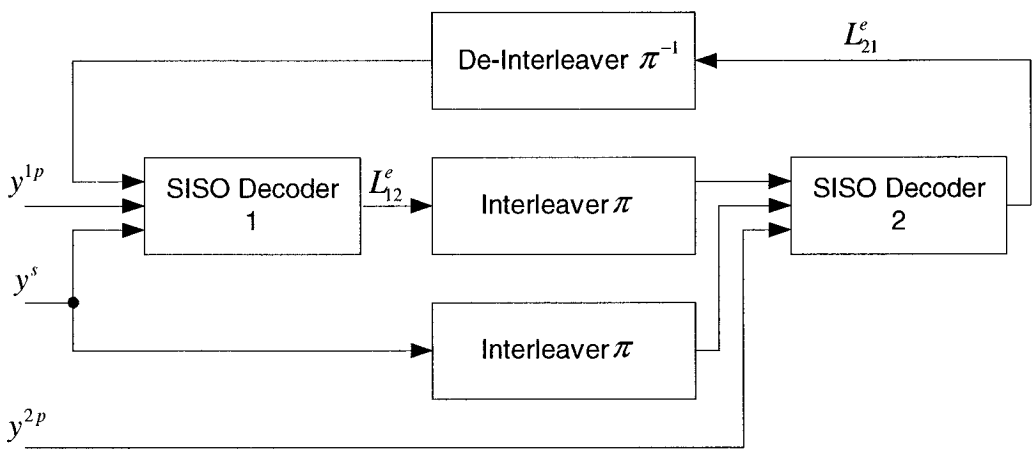


Figure 2-3 – Diagram of iterative (turbo) decoder

As shown in Figure 2-3, two soft-in/soft-out (SISO) decoders are used in the turbo decoder in an iterative manner. The input, noise corrupted binary symbol sequence, is de-multiplexed into three sequences, corresponding to the systematic sequence, y^s, and the two parity sequences, y^{1p} and y^{2p}, respectively. A SISO decoder takes y^s and y^{1p} (or y^{2p} for the second SISO decoder) as input and calculates the reliability information, i.e., the Log-Likelihood Ratio (LLR), of each information bit based on the trellis coding structure. The LLR of the k^{th} information bit, d_k, is defined as:

$$L(d_k) = \log \frac{\Pr\{d_k = 1/Y\}}{\Pr\{d_k = 0/Y\}}. \tag{2.1}$$

where Y is the received symbol sequence. A decision "1" is made for positive LLR and "0" for negative LLR. The absolute value of LLR provides the reliability of this decision, the larger the absolute value, the more reliable the decision is.

There are several optional algorithms for the SISO decoders. *Maximum A Posteriori* (MAP) decoding is the optimal SISO decoding algorithm in the sense of minimizing the symbol error probability, with large computation load [8]. A simplification version of the MAP algorithm, Max-Log-MAP, can achieve slightly worse performance with a great deal of complexity reduction [9]. By adding a modification to Max-Log-MAP, Log-MAP algorithm is obtained that provides nearly optimum performance. Another very popular SISO decoding algorithm is the Soft-Output Viterbi Algorithm (SOVA), based on some modification on conventional Viterbi Algorithm (VA) to generate the soft reliability information [10]. Using any of these decoding algorithms, for binary turbo code, it has been proved that the output LLR of the k^{th} bit, d_k, can be divided into three approximately independent terms,

$$L(d_k) = L_c(d_k) + L_{apri}(d_k) + L_e(d_k). \tag{2.2}$$

where $L_c(d_k)$ is the channel information, $L_{apri}(d_k)$ is the *a priori* information of d_k, and $L_e(d_k)$ is the *extrinsic* information, which is represented by L_{12}^e and L_{21}^e in Figure 2-3. The channel information depends only on the noise corrupted systematic symbol, while the *extrinsic* information is obtained based on the trellis structure together with the other systematic symbols and the received parity symbols. It is important that the *extrinsic* information, $L_e(d_k)$, be uncorrelated of $L_c(d_k)$ and $L_{apri}(d_k)$ since it will be used as *a priori* information by the other SISO decoder. However, with the number of iterations increasing, the correlation increases and the performance gained from additional iteration becomes less. Until certain stage, further iteration can no longer give any performance improvement, and then the iterative decoding can stop.

2.2 SISO Decoding Algorithms

The attractive near-capacity performance of Turbo codes is achieved by using decoding algorithms that generate soft-decisions, and by exchanging the soft-decision between two individual decoders. In this section, we present several SISO decoding algorithms.

The symbol-by-symbol Maximum A Posteriori (MAP) algorithm by Bahl, et al., which is often called BCJR algorithm, is optimal in estimating the states or outputs of a Markov process observed in white noise [8]. However, it is too complex to be implemented in practice, basically because of the representation of probabilities, non-linear functions and the mixed multiplications and additions of these values. Berrou, et al. [3] utilized a modified version of the BCJR algorithm to perform SISO decoding in the iterative turbo decoding process.

Some approximations of the MAP algorithm have been derived, such as the Max-Log-MAP algorithm [9], [12], and the Soft Output Viterbi algorithm (SOVA) [10], [11]. In both cases, processing is performed exclusively in the logarithmic domain so that values and operations (addition and max-function) are easier to be handled. However, both algorithms are sub-optimal. In [9], a Log-MAP algorithm was proposed to decrease the complexity dramatically with almost no performance loss.

2.2.1 MAP Decoding Algorithm and Its Variations

The MAP algorithm and its variations, Max-Log-MAP and Log-MAP algorithms, are summarized in the following.

2.2.1.1 MAP Algorithm

The MAP algorithm is designed to produce the LLR of each information bit, which is defined as the logarithm of the likelihood ratio, i.e., the *A Posteriori* Probability (APP) of the bit d_k being 1 to the APP of it being 0, based on the data received from the channel. The LLR can be calculated as,

$$L(d_k) = \ln \frac{\sum_{Sk} \sum_{S_{k-1}} \gamma_1(y_k, S_{k-1}, S_k) \cdot \alpha_{k-1}(S_{k-1}) \cdot \beta_k(S_k)}{\sum_{Sk} \sum_{S_{k-1}} \gamma_0(y_k, S_{k-1}, S_k) \cdot \alpha_{k-1}(S_{k-1}) \cdot \beta_k(S_k)} \tag{2.3}$$

where the α is the forward recursion path metrics, β is the backward recursion path metrics and γ is the branch metrics. The forward and backward path metrics can be calculated recursively as,

$$\alpha_k(S_k) = \frac{\sum_{S_{k-1}} \sum_{i=0}^{1} \gamma_i(y_k, S_{k-1}, S_k) \cdot \alpha_{k-1}(S_{k-1})}{\sum_{S_k} \sum_{S_{k-1}} \sum_{i=0}^{1} \gamma_i(y_k, S_{k-1}, S_k) \cdot \alpha_{k-1}(S_{k-1})} \tag{2.4}$$

and

$$\beta_{k-1}(s_{k-1}) = \frac{\sum_{S_k} \sum_{i=0}^{1} \gamma_i(y_k, S_{k-1}, S_k) \cdot \beta_k(S_k)}{\sum_{S_{k-1}} \sum_{S_k} \sum_{i=0}^{1} \gamma_i(y_k, S_{k-1}, S_k) \cdot \alpha_{k-1}(S_{k-1})} \tag{2.5}$$

where α_0 and β_N are initialized as,

$$\alpha_0(S_0) = \begin{cases} 1 & \text{for } S_0 = 0 \\ 0 & \text{elsewhere} \end{cases} \tag{2.6}$$

and

$$\beta_N(S_N) = \begin{cases} 1 & \text{for } S_N = 0 \\ 0 & \text{elsewhere} \end{cases} \tag{2.7}$$

when both CCs in the turbo encoder are terminated [3].

The branch transition probabilities are calculated as,

$$\begin{aligned} \gamma_i((y_k^s, y_k^p), S_{k-1}, S_k) = {} & q(d_k = i \mid S_k, S_{k-1}) \cdot p(y_k^s \mid d_k = i) \cdot \\ & p(y_k^p \mid d_k = i, S_k, S_{k-1}) \cdot \Pr\{S_k \mid S_{k-1}\} \end{aligned} \tag{2.8}$$

The value of $q(d_k=i \mid S_k, S_{k-1})$ is either one or zero depending on whether the transition from state S_{k-1} to S_k is valid or not. The *a priori* is used to calculate $Pr\{S_k|S_{k-1}\}$.

2.2.1.2 The Max-Log-MAP Algorithm

The Max-Log-MAP is a simplification of the MAP algorithm. As mentioned above, the MAP algorithm is likely to be too complex for implementation in a real system [9]. To reduce the number of complicated operations and also avoid number representation problems, the LLR calculation process described above can be implemented in the logarithm domain [13] – [16]. In this case, we no longer work with α, β and γ anymore, but with their logarithms.

By taking the logarithm of $\gamma_i((y_k^s, y_k^p), S_{k-1}, S_k)$ derived in (2.8) and inserting

$$p(y_k^p \mid d_k = i, S_k, S_{k-1}) = \frac{1}{\sqrt{\pi N_0}} \cdot e^{-\frac{1}{N_0}(y_k^p - x_k^p(i, S_k, S_{k-1}))^2} \tag{2.9}$$

we obtain the following expression for $q(.) = 1$:

$$\ln \gamma_i((y_k^s, y_k^p), S_{k-1}, S_k) = \frac{2 y_k^s x_k^s(i)}{N_0} + \frac{2 y_k^p x_k^p(i, S_k, S_{k-1})}{N_0} + \ln \Pr\{S_k \mid S_{k-1}\} + K \tag{2.10}$$

We can ignore the constant K, since it will be cancelled out in the calculation of $\ln \alpha_k(S_k)$ and $\ln \beta_k(S_k)$.

The $\ln(\alpha_k(S_k))$ is calculated as,

$$\begin{aligned} \ln \alpha_k(S_k) = & \ln\Big(\sum_{S_{k-1}} \sum_{i=0}^{1} e^{\ln \gamma_i((y_k^s, y_k^p), S_{k-1}, S_k) + \ln \alpha_{k-1}(S_{k-1})}\Big) \\ & - \ln\Big(\sum_{S_k} \sum_{S_{k-1}} \sum_{i=0}^{1} e^{\ln \gamma_i((y_k^s, y_k^p), S_{k-1}, S_k) + \ln \alpha_{k-1}(S_{k-1})}\Big) \end{aligned} \tag{2.11}$$

The following approximation is used to simplify the calculation,

$$\ln(e^{\delta_1} + \ldots + e^{\delta_n}) \approx \max_{i \in \{1,\ldots,n\}} \delta_i; \tag{2.12}$$

The term $\max_{i \in \{1,\ldots,n\}} \{\delta_i\}$ can be calculated by successively using n-1 maximum functions over two inputs.

If we define,

$$\overline{\gamma}_i((y_k^s, y_k^p), S_{k-1}, S_k) = \ln \gamma_i((y_k^s, y_k^p), S_{k-1}, S_k); \tag{2.13}$$

$$\overline{\alpha}_k(S_k) = \ln \alpha_k(S_k); \tag{2.14}$$

and

$$\overline{\beta}_k(S_k) = \ln \beta_k(S_k); \tag{2.15}$$

we replace the expressions in (2.4) and (2.5) as,

$$\overline{\alpha}_k(S_k) \approx \max_{(S_{k-1},i)}(\overline{\gamma}_i((y_k^s, y_k^p), S_{k-1}, S_k) + \overline{\alpha}_{k-1}(S_{k-1})) \tag{2.16}$$

and

$$\overline{\beta}_{k-1}(S_{k-1}) \approx \max_{(S_k,i)}(\overline{\gamma}_i((y_k^s, y_k^p), S_{k-1}, S_k) + \overline{\beta}_k(S_k)) \tag{2.17}$$

The LLR of each bit d_k is then calculated as,

$$\begin{aligned} L(d_k) \approx & \max_{(S_k,S_{k-1})}(\overline{\gamma}_1(y_k, S_{k-1}, S_k) + \overline{\alpha}_{k-1}(S_{k-1}) + \beta_k(\overline{S}_k)) - \\ & \max_{(S_k,S_{k-1})}(\overline{\gamma}_0(y_k, S_{k-1}, S_k) + \overline{\alpha}_{k-1}(S_{k-1}) + \beta_k(\overline{S}_k)). \end{aligned} \tag{2.18}$$

Because of the approximation shown in (2.12), the Max-Log-MAP algorithm is sub-optimal and yields an inferior soft-output than the MAP algorithm.

2.2.1.3 Log-MAP

The inferior performance of the Max-Log-MAP algorithms comes from the simplification made in (2.12). The logarithm of the sum of two exponents can be accurately calculated as,

$$\ln(e^{\delta_1} + e^{\delta_2}) = \max(\delta_1, \delta_2) + \ln(1 + e^{|\delta_1 - \delta_2|}); \tag{2.19}$$

The two terms on the right side are the maximum exponent and a correction function of the absolute difference between the two exponents.

Let $\Delta = e^{\delta_1} + ... + e^{\delta_{n-1}} = e^{\delta}$, then $\ln(e^{\delta_1} + ... + e^{\delta_n})$ can be computed recursively as follows:

$$\begin{aligned} \ln(e^{\delta_1} + ... + e^{\delta_n}) &= \ln(\Delta + e^{\delta_n}) \\ &= \max(\ln \Delta, \delta_n) + f_c(|\ln \Delta - \delta_n|) \\ &= \max(\delta, \delta_n) + f_c(|\delta - \delta_n|) \end{aligned} \tag{2.20}$$

Thus, the original MAP algorithm can be preserved by correcting the approximation at each step when deriving the log-MAP algorithm.

To avoid the exponential and logarithm computations, the f_c can be approximated by a pre-calculated look-up table (LUT) of $\log(1 + e^{-x})$. It has been shown that 3-bit

quantization of the f_c provides nearly no performance degradation. This means a LUT with only 8 different values is needed.

2.2.2 SOVA Decoding Algorithm

Soft-Output Viterbi Algorithm (SOVA) was proposed by Hagenauer et al. in [10] as another SISO decoding algorithm. In SOVA, the path metric difference between the survivor and the discarded paths is used to update the soft output of each information bit.

The LLR of a binary path decision is,

$$\hat{L}_j = \frac{\Pr(correct)}{1-\Pr(correct)} \tag{2.21}$$

Let's assume that a SOVA decoder makes a decision with delay d_δ, d_δ being large enough so that all 2^M surviving paths have been merged with sufficiently high probability, where 2^M is the number of states in the trellis. The decoder selects a survivor for state s_k like in a conventional Viterbi algorithm (VA), $0 \le s_k \le S = 2^M - 1$ at time instance k, by selecting the path with the smallest ML metric, which, for the AWGN channel, is,

$$M_m = \frac{E_s}{N_0} \sum_{j=k-\delta}^{k} \sum_{n=1}^{N_b} (y_{jn} - x_{jn}^{(m)})^2, \quad m = 1,2 \tag{2.22}$$

where $x_{jn}^{(m)}$ is the n^{th} bit of the N_b bits on the branch of the m^{th} path at time j, y_{jn} is the received value at the same position, and E_s/N_0 is the SNR.

If at any state, the path with the smaller metric is labeled by $m = 1$, which means that $M_1 \le M_2$, the VA selects path 1 as the survivor. The probability of selecting the wrong surviving path can be calculated as,

$$p_{sk} = \frac{e^{-M_2}}{e^{-M_2} + e^{-M_1}} = \frac{1}{1 + e^{M_2 - M_1}} = \frac{1}{1 + e^{\Delta}} \tag{2.23}$$

with $\Delta = M_2 - M_1 \ge 0$. The p_{sk} is actually the probability that the VA has made errors in all the δ_e positions where the information bits of path 2 differ from path 1,

$$u_j^{(1)} \ne u_j^{(2)}, \qquad j = j_1, j_2, \cdots, j_{\delta_e} \tag{2.24}$$

Positions where $u_j^{(1)} = u_j^{(2)}$ are not affected by the survivor decision. Let δ_m be the length of those two paths until they merge. There are δ_e different information values, and

$\delta_m - \delta_e$ nondifferent values. Assuming that the probabilities $\hat{p}_j$ of previous erroneous decisions with path 1 have been stored, under the assumption that path 1 has been selected, the erroneous probabilities for the δ_e differing decisions on this path are modified according to,

$$\hat{p}_j \leftarrow \hat{p}_j \cdot (1 - p_{sk}) + (1 - \hat{p}_j) \cdot p_{sk}, \qquad j = j_1, \cdots, j_{\delta_e} \tag{2.25}$$

where $0 \le \hat{p}_j \le 0.5$.

This formula requires statistical independence between the random variables $\hat{p}_j$ and p_{sk}, which is approximately true when a proper interleaver is used. The soft information updating could be directly performed on the LLR as,

$$\hat{L}_j = \frac{1}{\alpha} \log \frac{1 - \hat{p}_j}{\hat{p}_j}, \qquad 0 \le \hat{L}_j \le \infty \tag{2.26}$$

Using (2.23), (2.25) and (2.26), the following LLR updating equation is obtained,

$$\hat{L}_j \leftarrow f(\hat{L}_j, \Delta) = \frac{1}{\alpha} \log \frac{1 + e^{(\alpha \hat{L}_j + \Delta)}}{e^{\Delta} + e^{\alpha \hat{L}_j}} \tag{2.27}$$

with $\Delta = M_2 - M_1 \ge 0$, $j = j_1, \ldots, j_e$. The function $f(\hat{L}_j, \Delta)$ should be tabulated with $\hat{L}_j$ and Δ as input variables and need not to be calculated at each step. The factor α prevents overflow with increasing SNR.

A good approximation of (2.27) is

$$f(\hat{L}_j, \Delta) = \min(\hat{L}_j, \Delta / \alpha) \tag{2.28}$$

which requires neither a table nor the SNR.

In [15], an additional modification is proposed to improve the decoding performance. The soft decisions of the nondifferent bits along the two paths entering one state node are also changed after the metric comparison, according to the following:

$$\hat{L}_{j-new} = \min\left\{\Delta + \hat{L}_j^2, \hat{L}_j^1\right\} \tag{2.29}$$

This equation is based on the fact that, for the nondifferent bits, $\hat{L}_j^2$ is the metric difference between path 2 and a previously discarded path in the trellis, which has a different bit decision corresponding to the j^{th} position in path 2. Therefore, it also has different bit decision at this position with path 1. This path, now, should be taken into

account as the most likely path with the different decision on this bit. Therefore, the soft decision of this bit should be updated again in a similar way presented in (2.27).

Implementation of SOVA consists of a conventional VA and the additional soft information updating operations. The additional operations consists of calculating the path metric difference, performing path trace-back of both the surviving path and the discarded path, locating the different decision positions along the track-back, the LLR updating of the bits given different decisions by the two paths according to (2.28), and possibly the LLR updating of the bits given same decisions by the two paths according to (2.29). Furthermore, additional memory is required to store the soft information.

This implementation is pretty complex since at each time instance, the SOVA performs LLR-updating related operations for each state, and there are 2^M states. A simplified implementation is proposed in [11]. In this implementation, a first VA is performed to decide the optimal path. Since all the 2^M survivors at time instance k have a high probability to merge at time $k-L$, provided L is large enough, the maximum-likelihood path before $k-L$ can be decided at time k. This first VA performs a single trace back for each of the 2^M survivors. A SOVA, with soft information updating, is carried out L branches later than the first VA. This SOVA performs double trace-back, along the survivor and the discarded path, only on the states along the maximum-likely path decided by the first VA. The complexity in performing the LLR updating of this implementation is therefore only $1/2^M$ of the original SOVA. It is shown that the performance degradation of this simplified implementation is negligible [11]. We will use this simplified SOVA implementation in our simulation in Chapter 3 to compare the performances of turbo decoders using different SISO algorithms.

It is shown in [9] that the complexity of SOVA is less than one half of the Max-log-MAP algorithm. It is also shown in [9] that the performance of SOVA is approximately 0.6 dB less than MAP algorithm, slightly less than Max-Log-MAP. However, it is shown in [12] that after the modification in (2.29), SOVA provides exactly the same soft information as the Max-Log-MAP algorithm.

2.2.3 Comparison Between MAP Based vs. SOVA Based Turbo Decoding

The implementation of MAP decoder is very complex mainly due to the required number of multiplications and exponential operations. As illustrated in [9], the Log-MAP algorithm is equivalent to the optimum MAP algorithm in estimating the outputs of a Markov process, whereas the multiplications become additions and the exponentials disappear.

2.2.3.1 Comparison of the (Max-) Log-MAP and SOVA Algorithms

In MAP decoding, all the paths are used to calculate the LLR of each information bit. The paths are divided into two groups: one group with a decision of this bit being "1" and the other with a decision of this bit being "0". The LLR is calculated as the difference between the two total APPs (sum of all the APPs provided by the paths in each group) of this two set.

In contrast, the Max-Log-MAP considers only two paths at each transition: the best with bit zero and the best with bit one. The difference between the two APPs provided by these two best paths are used to calculate the LLR. However, from step to step one of these paths can change, but one will always be the Maximum-Likelihood (ML) path.

The SOVA will always correctly find one of these two paths (the ML path), but not necessarily the other, since it may have been eliminated before merging with the ML path. There is no bias on the SOVA output when compared to that of the Max-Log-MAP algorithm, only the former will be noisier [9].

2.2.3.2 Computational Complexity Analysis and Comparison

In this section, based on the calculation procedures of the different SISO algorithms described above, we perform a decoding complexity analysis and comparison. Please refer to Appendix A.1 for detailed derivations of the results presented in this section.

As presented in section 2.2.1.3, the Log-MAP algorithm has four major operations:

- branch metric calculation (γ),
- forward path metric calculation (α) and backward path metrics calculation (β),
- soft output LLR/*extrinsic* information calculation.

The Max operation and lookup table are used widely in each operation. The subtraction is represented as one inverter (multiply ±1) plus one adder. The computational complexity of Log-MAP algorithm is analyzed and presented in Table 2-1. The entries are the numbers of corresponding operations required to decode each information bit.

Table 2-1 – Log-MAP computational complexity.

Operation	Addition	Max Operations	Lookup Table	Inversion (Subtraction)
α calculation	$3 \cdot 2^M$	2^M	2^M	
β calculation	$3 \cdot 2^M$	2^M	2^M	
γ calculation	8	0	0	2
LLR calculation	$2 \cdot (3 \cdot 2^M-1)+1$	$2 \cdot (2^M-1)$	$2 \cdot (2^M-1)$	1
Total	$12 \cdot 2^M+7$	$4 \cdot 2^M-2$	$4 \cdot 2^M-2$	3

Comparing with the Log-MAP algorithm, the Max-Log-MAP algorithm is simplified by ignoring the correction term. The complexity of Max-Log-MAP algorithm is decreased at the price of BER performance degradation. The computational complexity of Max-Log-MAP algorithm is given in Table 2-2. Same as Table 2-1, the entries are the numbers of corresponding operations per decoded information bit.

Table 2-2 – Max-Log-MAP computational complexity.

Operation	Addition	Max Operations	Lookup Table	Inversion (Subtraction)
α calculation	$2 \cdot 2^M$	2^M		
β calculation	$2 \cdot 2^M$	2^M		
γ calculation	8	0		2
LLR calculation	$4 \cdot 2^M+1$	$2 \cdot (2^M-1)$		1
Total	$8 \cdot 2^M+9$	$4 \cdot 2^M-2$		3

The SOVA algorithm can be considered as having two calculation stages: best path selection classical Viterbi Algorithm and soft output LLR/*extrinsic* information generation (including the double-trace-back operation and software LLR updating). The numbers of various mathematic operations per decoded bit are shown in Table 2-3.

Table 2-3 – SOVA computational complexity.

Operation	Addition	Max Operations	Inversion (Subtraction)
Best path selection	Up to $2 \cdot 2^M$	2^M	
2nd Viterbi decoding	$2 \cdot 2^M$	2^M	
γcalculation	8		2
LLR calculation	$6 \cdot (M+1)$	$6 \cdot (M+1)$	0
Total	$4 \cdot 2^M+6 \cdot M+14$	$2 \cdot 2^M+6 \cdot M+6$	2

It is observed that the number of operations of the Log-MAP is about twice that of the SOVA. However, it will be shown in next chapter that the performance of turbo decoding with Log-MAP algorithm is about 0.5 to 1 dB better than that of turbo decoding with SOVA as component decoders.

2.3 Conclusions

In this chapter, we performed a brief introduction to the turbo codes and their major components: encoder, interleaver and decoder.

- A turbo encoder mainly consists of two (or more) constituent codes (CCs) separated by interleavers.
- The two major characteristics of the turbo code interleavers are the length and the randomness.
- A practical turbo decoding is performed using iterative decoding structure with several SISO decoders.
- There are four popular SISO decoding algorithms proposed in the literature, MAP, Max-Log-MAP, Log-MAP and SOVA.
- MAP is too complex and not feasible for a practical implementation.
- Max-Log-MAP and SOVA are relatively simple in terms of implementation complexity but at the cost of a performance loss.
- The number of operations of the Log-MAP is about twice that of the SOVA but with only a 0.5 dB performance loss compared with MAP. Furthermore, the Log-MAP is more suitable for parallel processing which could reduce the overall delay.

Chapter 3

Simulation Of Turbo Coded Communication Systems

In this chapter, we present the simulation results of turbo decoding using various SISO decoding algorithms introduced in Chapter 2. Although MAP algorithm is known to be too complex to implement in practice, simulation of turbo decoding with MAP algorithm is also performed, mainly as the reference to indicate how close the other sub-optimal SISO decoding can approach the optimal performance. With the simulation results, a performance comparison is performed among all these SISO algorithms.

Following the performance comparison, the complexities of the SISO algorithms are compared using the execution time on the same simulation platform as the benchmark. This is not the best benchmark to perform the complexity comparison since the specific structure of each SISO decoder cannot be optimized in this common simulation platform. However, it does shed some light on the relative decoding complexities of these SISO algorithms.

With the performance comparison, the complexity comparison presented in Chapter 2 and this chapter, the one that has the best performance-complexity compromise is chosen as the candidate to be used in the VLSI turbo decoder implementation in Chapter 4. The simulation results of turbo decoding with the SISO algorithms are validated by comparing with the results reported in literature [3], [9]. This is necessary because the

VLSI design of turbo decoder is carried out using VHDL based on the C-program algorithm implementation in the simulation.

The simulation platform is a Personal Computer (PC) with Pentium 800 Central Processing Unit (CPU) with 512Mbyte memory. For fair comparison, the same assumptions are made in all simulations with these different algorithms, including the channel model, number of iterations, etc. The simulation assumptions are summarized in Table 3-1.

Table 3-1 – Simulation assumptions.

Channel	AWGN
Modulation	BPSK
Block Length	1024 (1020 info bits + 4 tail bits)
Turbo Coding Rate	1/3
Puncture	No
RSCC (2)	Rate ½, (31, 27), 16 state
Interleaver	S-random, spreading factor 20
Number of Iterations	1-10
Signal to Noise Ratio	Exact SNR value available at the decoder
RSCC termination	Both CCs are terminated

The simulations are performed over turbo-coded Binary Phase Shift Keying (BPSK) transmissions in Additive White Gaussian Noise (AWGN) channels. In the simulation, the Box-Muller method is used to generate the Gaussian random noise samples. Detailed information could be found in Appendix A.2.

The turbo encoder in the simulations uses two identical rate-1/2 RSCCs with the generator polynomial of [31,27]. The generator polynomials are one of the optimum in terms of maximizing the effective free distance of the component code. The output of the turbo encoder is the multiplexed sequence of the systematic bit sequence and the two parity bit sequences output by the two RSCCs. This results in an overall code rate of 1/3. The interleaver used between two RSCCs is a 1020-bit s-random interleaver. At the decoder, 10 iterations are performed in the iterative decoding process. In this simulation, we assume that both RSCCs are terminated. A detailed discussion of RSCC termination is presented in section 4.1.1.

The simulation results for turbo decoding with one to ten iterations are obtained and presented. Since only very little performance improvement can be obtained with further decoding iterations, the decoding stops at ten iterations

Since the overall code rate is about 1/3 (slightly less than 1/3 with RSCC termination), the Shannon limit can be calculated by (1.2) as $C_{R=1/3} = -0.55$ dB.

3.1 Simulation results analyses

This section presents the simulation results of turbo coding employing MAP (Modified BCJR), Log-MAP, Max-Log-MAP and SOVA decoding algorithms. The performance of each simulated decoding algorithm is compared with the results published in literature.

3.1.1 Simulation with MAP

Simulation results of turbo coded BPSK transmission in AWGN channels using MAP decoding algorithm are shown in Figure 3-1. This performance is about 1.55 dB from the Shannon limit. A rate ½ turbo code is reported to be only 0.7 dB away from Shannon limit in [3]. This larger gap resulted from our simulation can be explained by the small interleaver size we use in the simulation, which is 1020-bit, where in [3], an interleaver of size 65536-bit is used.

It is shown in Figure 3-1 that turbo codes are capable of achieving very low Bit Error Rates (BER) in regions of low SNR values.

For comparison purpose, the BER of a uncoded BPSK transmission and the BER obtained using a standard (2,1,3) non-recursive convolutional code are also shown. The performance improvement introduced by turbo coding at low signal-to-noise ratio range is significant.

From theoretical uncoded curve, we find the BER is around 10^{-5} at SNR=9.6dB. This simulated uncoded BPSK BER curve is in accordance with the theoretic results.

The coding gain performance is calculated as:

$$G = SNR_{un} - SNR_{co} \tag{3.1}$$

where SNR_{un} and SNR_{co} are the required SNR per channel bit in uncoded and coded systems. About 8.6 dB coding gain is achieved at BER of 10^{-5}.

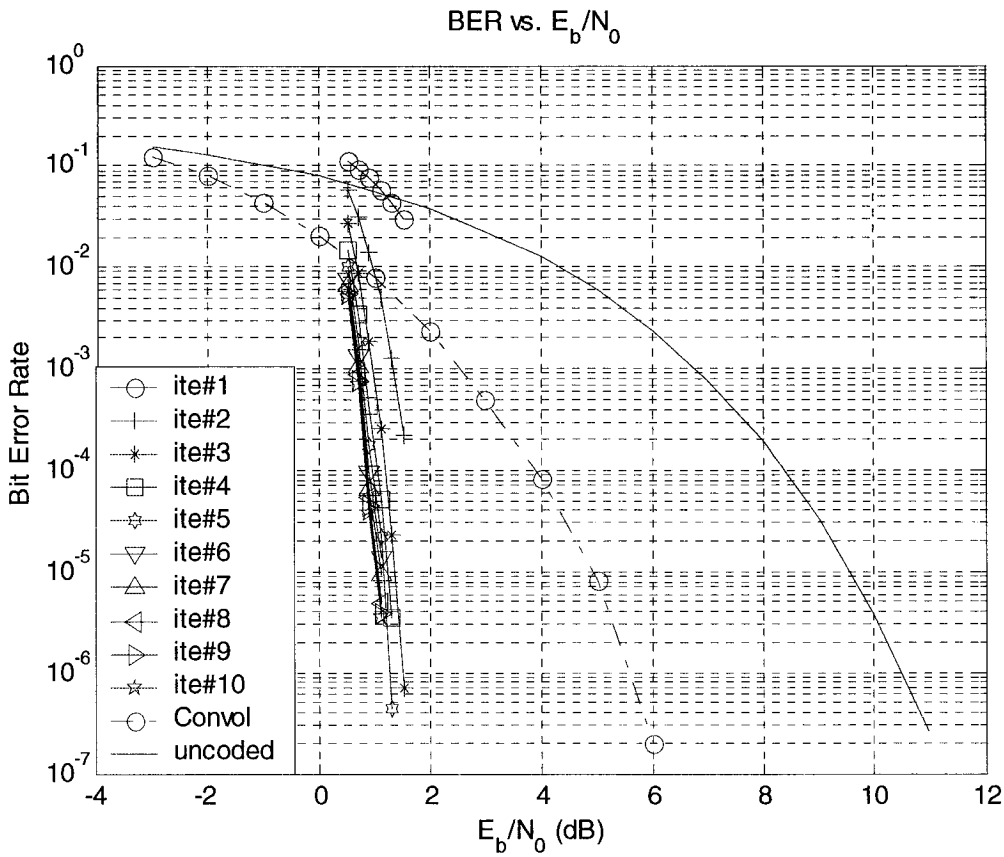


Figure 3-1 – Performance comparison among a) simulated turbo coding with MAP decoding algorithm; b) rate ½, 4-state convolutional code in AWGN; c) uncoded BPSK Transmission.

The convolutional code used for comparison is rate 1/2 code with constraint length k=3. The generator polynomial of this convolutional code is $g = [1\ 1\ 1;\ 1\ 0\ 1]$. To achieve the same 10^{-5} BER, an SNR of 4.4dB is required. Therefore, the coding gain of this convolutional code is 5.2dB. Using turbo code provides additional 3.3 dB coding gain over this convolutional code.

The performance for the BCJR algorithm is shown for various numbers of iterations 1 – 10. It is observed that with only one iteration, performance of the iterative decoding is very poor. With two iterations, coding gain becomes significant. After four iterations, the performance improvement by adding more iteration becomes insignificant. A zoom out view of is shown in Figure 3-2.

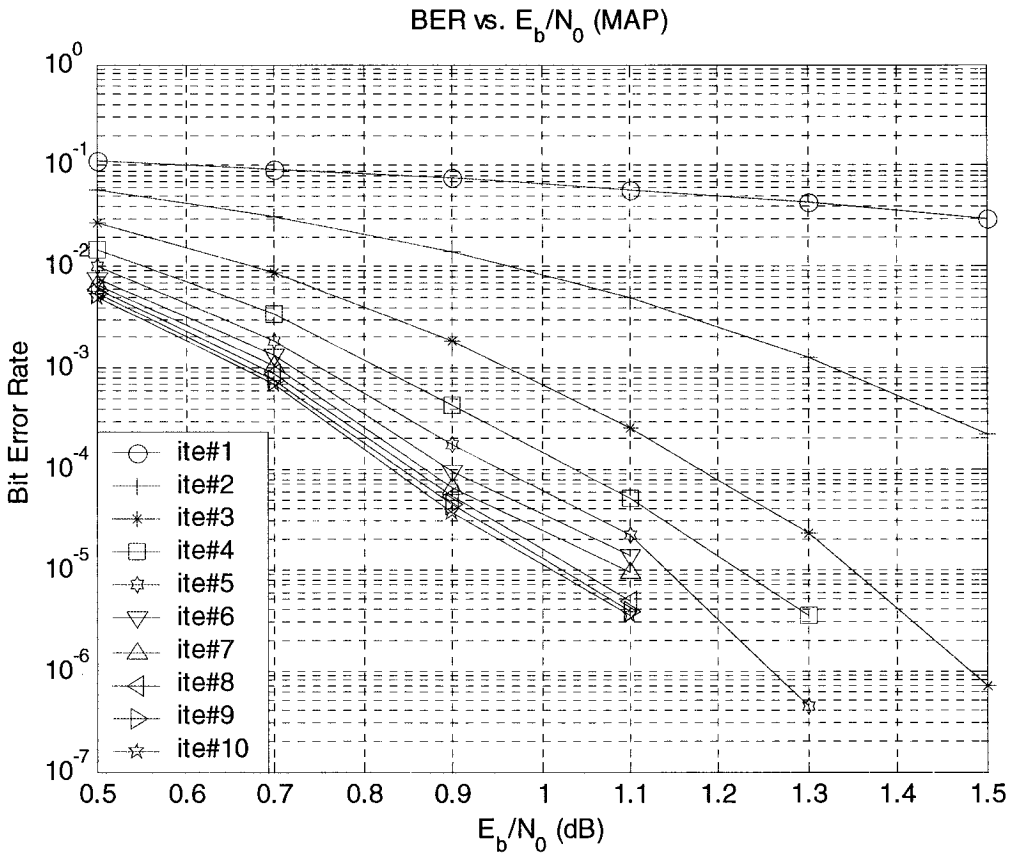


Figure 3-2 – Simulation of (31, 27), rate 1/3, 16-state, turbo code employing MAP decoding algorithm with 1020 *s*-random interleaver (S=20), over 40000 simulation blocks, 1 to 10 iterations, BPSK transmissions.

3.1.2 Simulation with Max-Log-MAP

The performances of turbo decoding with Max-Log-MAP algorithm with similar simulation assumptions are plotted in Figure 3-3 and Figure 3-4. With four iterations, 8 dB coding gain is achieved at a BER of 10^{-5}. Simulations with more than four iterations gives no error after E_b/N_0 or 1.5 dB because of insufficient simulation data. However, it is expected that the performance is very close to the performance obtained with four iterations.

Since some applications are more concerned with block error rate instead of the BER, the block error rate resulted from using turbo coding with Max-Log-MAP decoding algorithm is plotted in Figure 3-4 as another measure of the decoding performance.

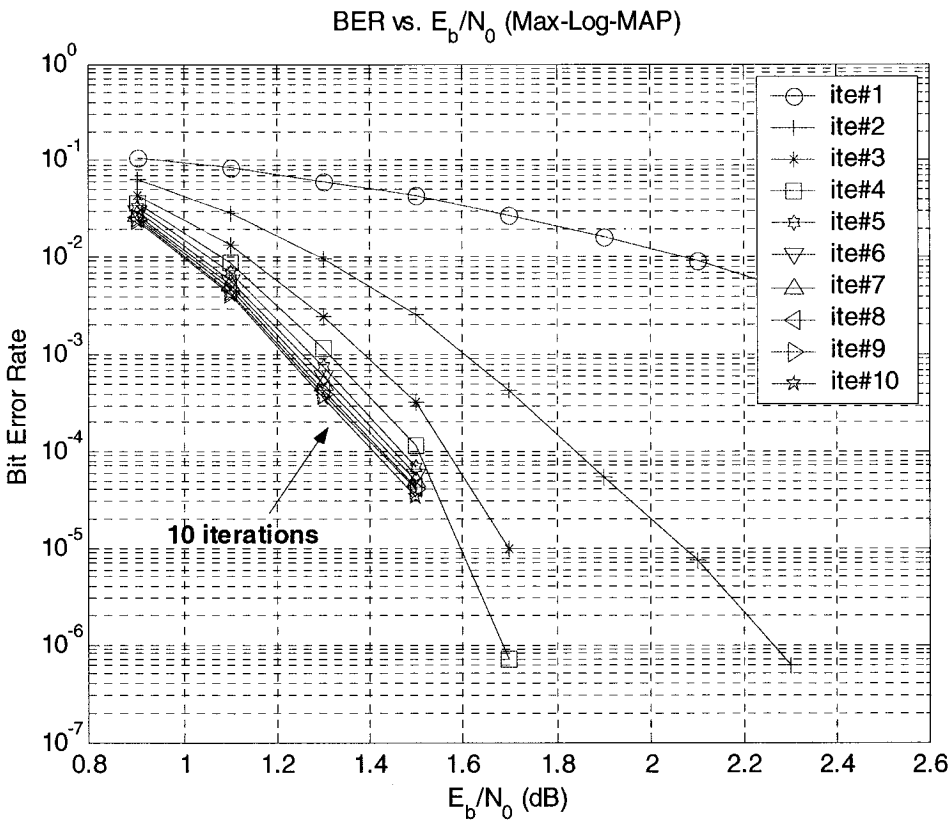


Figure 3-3 – Simulation of (31, 27), rate 1/3, 16-state turbo code employing Max-Log-MAP decoding algorithm with 1020 *s*-random interleaver (S=20), over 30000 simulation blocks, 1 to 10 iterations, BPSK transmissions.

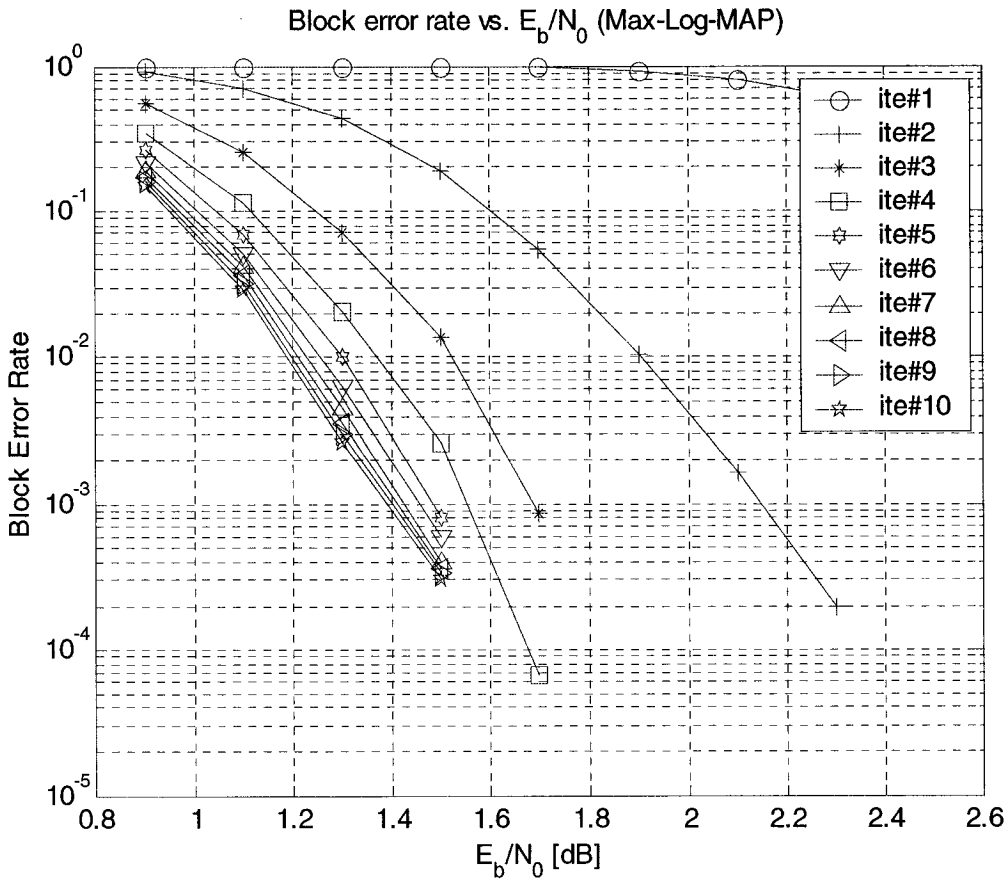


Figure 3-4 – Simulation of (31, 27), rate 1/3, 16-state turbo code employing Max-Log-MAP decoding algorithm with 1020 *s*-random interleaver (S=20), over 30000 simulation blocks, 1 to 10 iterations, BPSK transmissions.

The simulation curves of turbo code using soft decision Max-Log-MAP decoding algorithm with ten iterations shows about 8.0 dB coding gain at BER of 10^{-5}. The coding gain achieved using Max-Log-MAP SISO algorithm is 0.6 dB less than that of using MAP algorithm.

3.1.3 Simulation with Log-MAP

The performance of using Log-MAP as the component decoder with similar simulation consideration is given in Figure 3-5. The results for BER vs E_b/N_0 and block error rate vs E_b/N_0 with one to ten iterations are plotted in Figure 3-6.

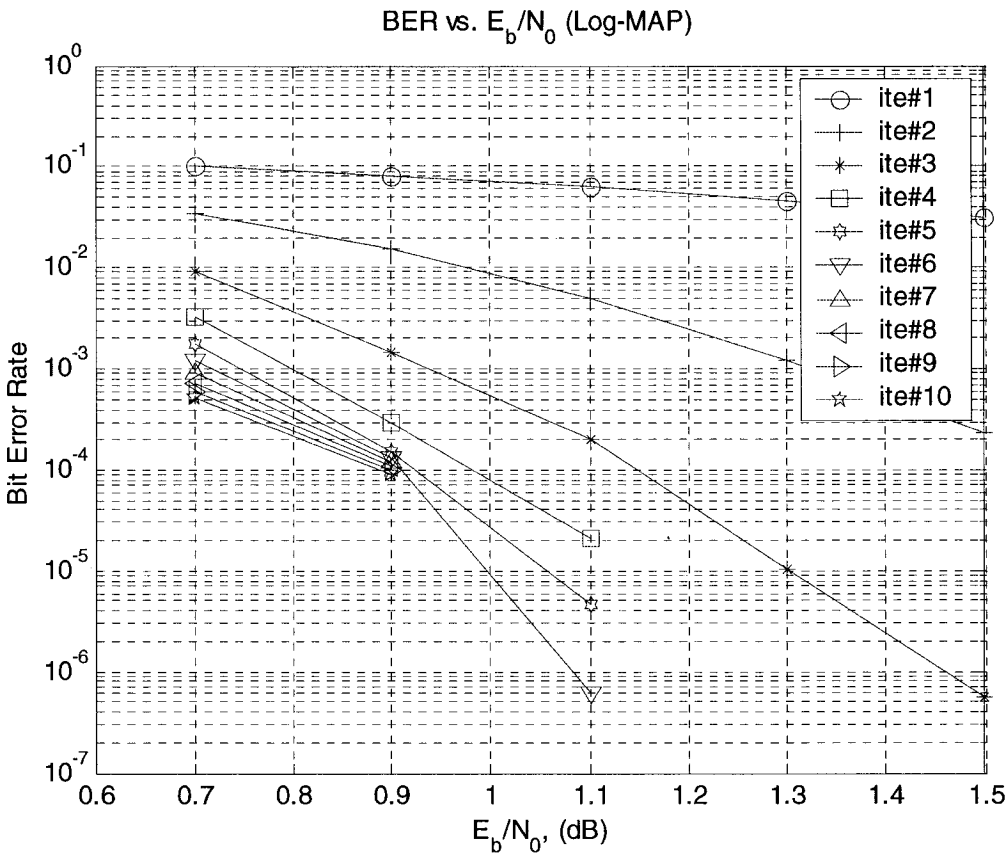


Figure 3-5 – Simulation of (31, 27), rate 1/3, 16-state turbo code employing Log-MAP decoding algorithm with 1020 *s*-random interleaver (S=20), over 30000 simulation blocks, 1 to 10 iterations, BPSK transmissions.

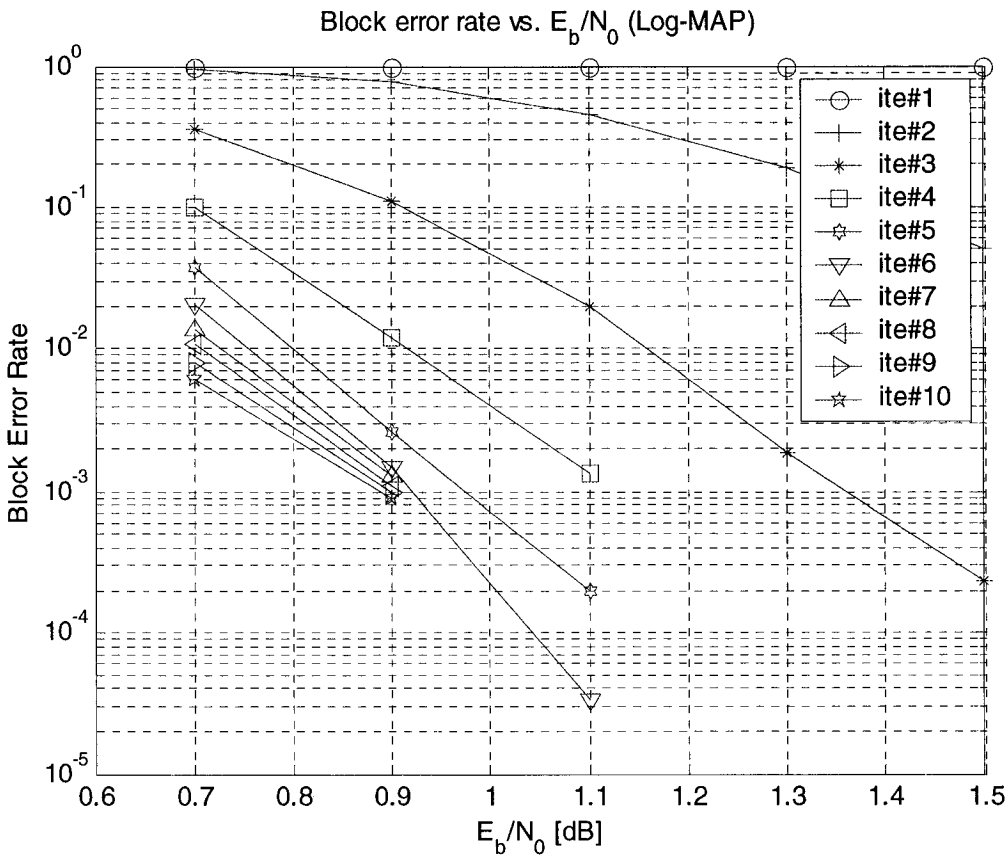


Figure 3-6 – Simulation of (31, 27), rate 1/3, 16-state turbo code employing Log-MAP decoding algorithm with 1020 *s*-random interleaver (*S*=20), over 30000 simulation blocks, 1 to 10 iterations, BPSK transmissions.

In this simulation, an 8-values lookup table is used to perform the modification shown in (2.19). This algorithm is therefore sub-optimal compared to MAP decoding algorithm since it has a quantization error when performing the modification.

The simulation curves of turbo code using soft decision Log-MAP decoding algorithm with ten iterations shows about 8.6 dB advantage in signal-to-noise ratio at BER of 10^{-5}, which is almost the same as that obtained using MAP algorithm. This is compliant to what was reported in the literature.

In the Log-MAP simulation result, the BER at SNR 1.1 dB is even better that MAP. Because the number of error events occur is very low, the confidence interval is quite loose here. Therefore, the BER at this point is not very reliable.

3.1.4 Simulation with SOVA

The BER performance of turbo coded BPSK transmission using SOVA as the SISO decoder is shown in Figure 3-7 and the block error rate performance is shown in Figure 3-8.

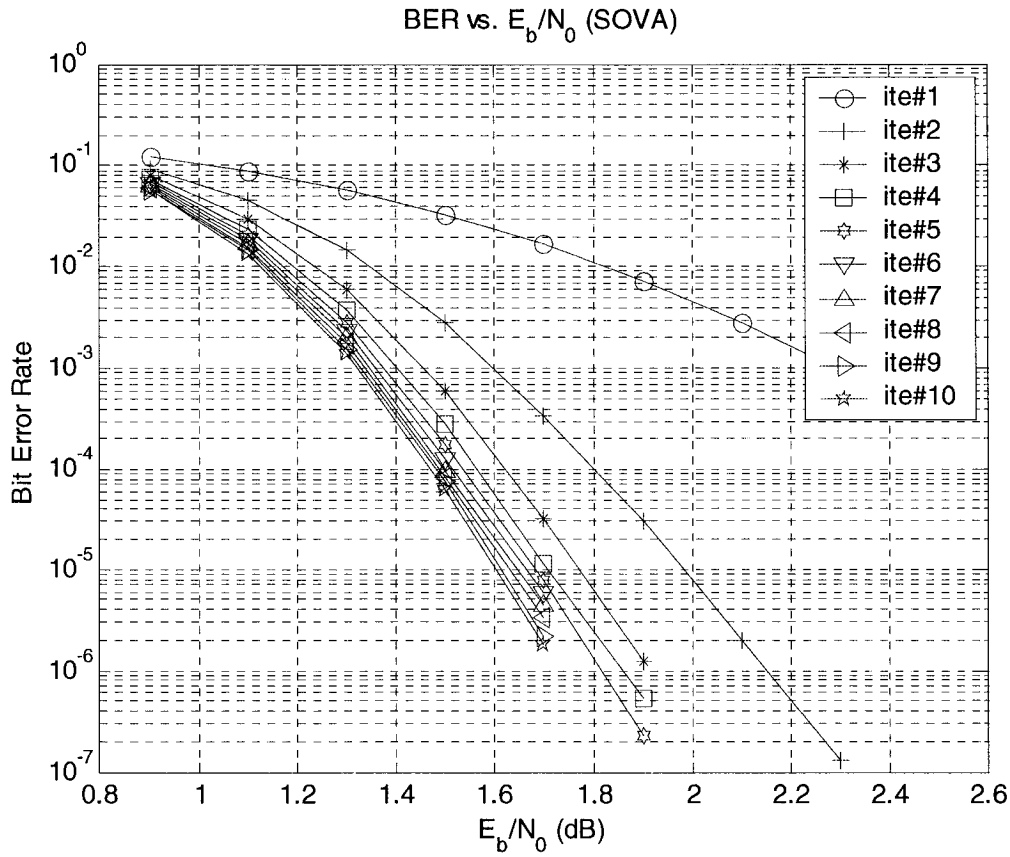


Figure 3-7 – Simulation of (31, 27), rate 1/3 turbo code employing SOVA decoding algorithm with 1020 *s*-random interleaver (S=20), over 30000 simulation blocks, 1 to 10 iterations, BPSK transmissions.

The simulation curves of turbo code using soft decision SOVA decoding algorithm with 10 iterations shows about 8.0 dB coding gain at BER of 10^{-5}. This is about the same as the Max-Log-MAP algorithm. This result is a little bit different from those reported in the literature, where turbo decoding with Max-Log-MAP should have a better coding gain performance than that using SOVA. However, their difference is usually very small,

i.e., at the range of 0.2 dB or so. Therefore, it might require much larger amount of simulation data to show their performance difference.

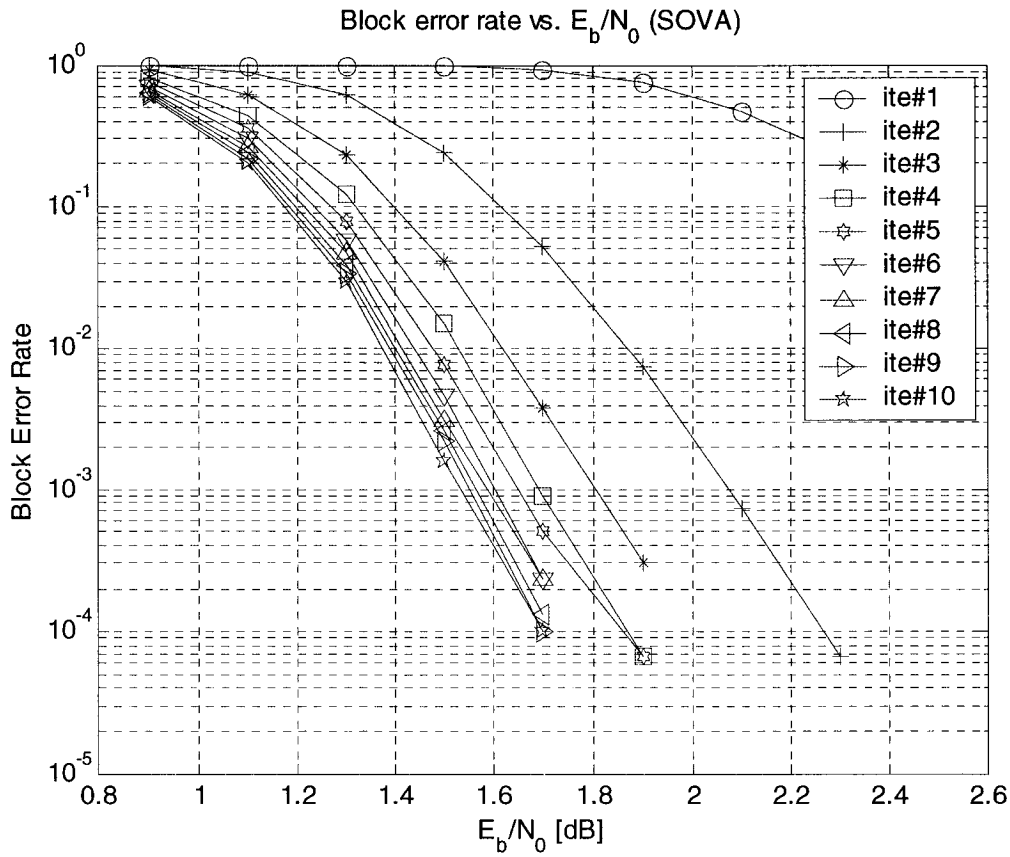


Figure 3-8 – Simulation of (31, 27), rate 1/3 turbo code employing SOVA decoding algorithm with 1020 *s*-random interleaver (S=20), over 30000 simulation blocks, 1 to 10 iterations, BPSK transmissions.

3.1.5 Confidence Interval

Since only finite bits are tested in our simulations, the reliability of the BER estimation is quantified in terms of confidence interval [17]. Please refer to Appendix A.3 for more discussions on the BER estimation.

In turbo decoding, an error event generates multiple bit errors with a complicated distribution, therefore, it is difficult to obtained an accurate value of the estimation

variance. However, assuming an average number of bit errors in each error event, the variance is easily estimated as (6.11) and the confidence interval is estimated with (6.10).

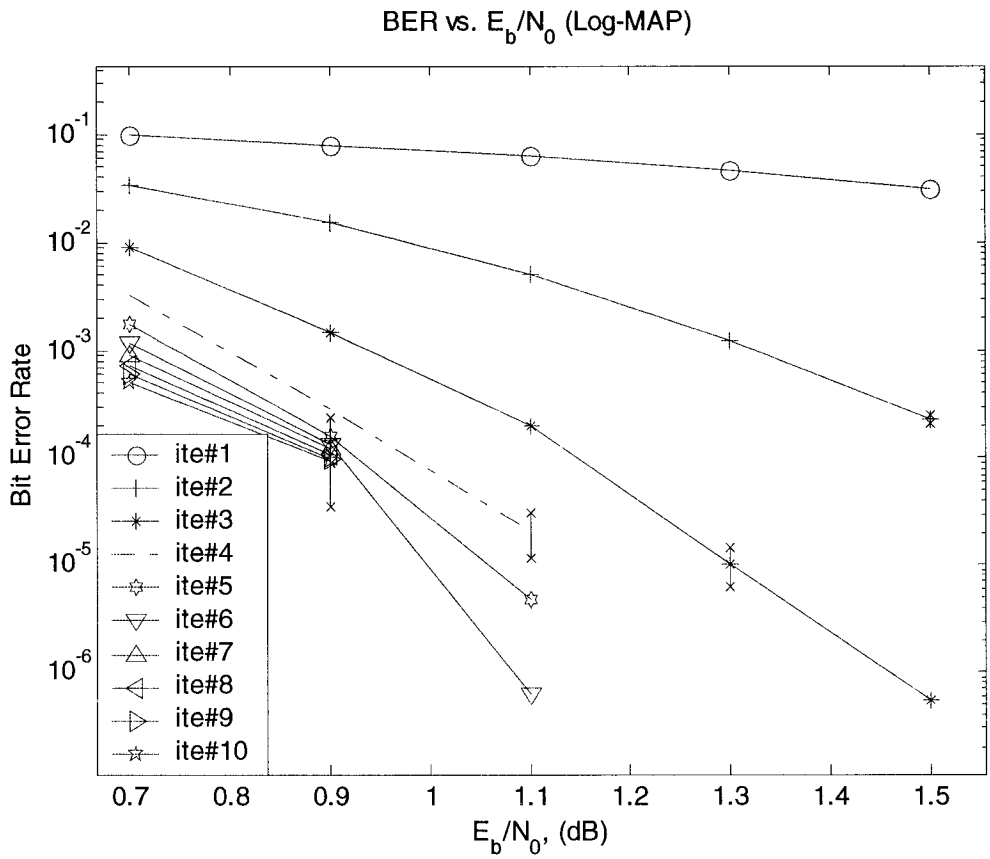


Figure 3-9 – Confidence interval (α = 0.05) of the Log-MAP turbo coding simulation.

Figure 3-9 shows the confidence intervals (α = 0.05) of the simulated Log-MAP decoding at different SNRs. The simulations can thus be considered reliable and valid based on the confidence intervals observed.

3.2 Comparison between turbo codes using various SISO decoders

In this section, we compare the decoding performances and complexities of employing different SISO decoding algorithms in the turbo coded BPSK transmissions, according to the simulation results presented above. Based on this comparison, the SISO algorithm that provides the best compromise between the performance and complexity will be our choice to be implemented with VHDL in next chapter.

3.2.1 Performance comparison

Turbo decoding performances using the four SISO decoding algorithms are shown in Figure 3-10.

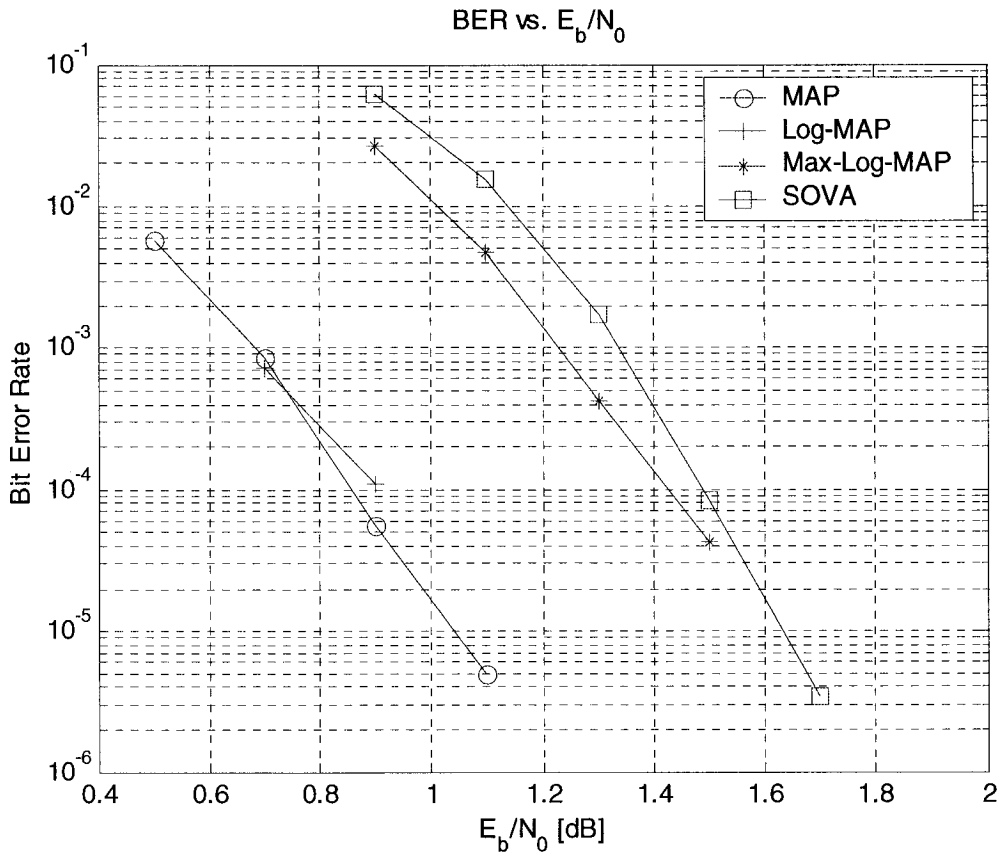


Figure 3-10 – Turbo decoding performance comparison among different SISO decoding algorithms with 8 iterations. Simulation of (31, 27), rate 1/3 turbo code with 1020 *s*-random interleaver, AWGN channel, BPSK transmissions.

From the Figure 3-10, it could be observed that the MAP algorithm has the best performance out of all the simulated decoding algorithms. The Log-MAP algorithm SNR at BER of $10^{-3.5}$ has very slight difference from that of the MAP algorithm and is approximately 0.6dB better than that of SOVA at BER of 10^{-5}. The Max-Log-MAP is about 0.1dB better than the SOVA at BER of 10^{-5}. The simulation results are in concordance with those reported by Robertson in [9].

3.2.2 Complexity comparison

An approximate complexity comparison was given in last chapter based on the number of the different types of computations the SISO decoders performs to process each information bit. In this section, we take the execution time of each SISO algorithm in the same simulation platform as a measure to compare the computation complexities of the four SISO algorithms. The simulation platform is a PC with a CPU Pentium 800 and 512Mbyte memory. The average CPU times taken by the four SISO decoders to process a block of data are reported in Figure 3-11.

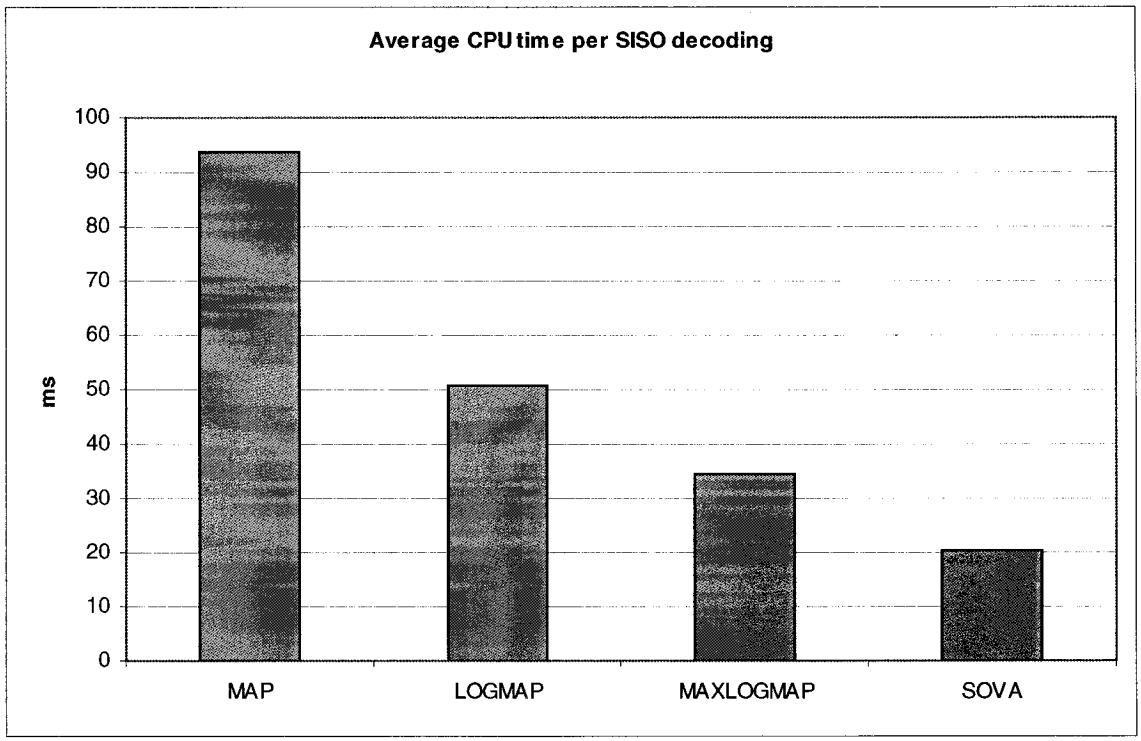


Figure 3-11 – Execution time comparison.

In Figure 3-11, it is shown that the simulated system model employing the MAP algorithm has the highest computational complexity. The complexity of Log-MAP is approximately 1.5 times of that of the Max-Log-MAP algorithm and about 2.5 times of that of SOVA. Taking into account that the simulations were performed on PC that uses a general purpose CPU, this diagram mainly intends to provide an illustrative comparison

rather than strict complexity comparison from hardware implementation perspective. This chart should not be used to benchmarking the complexity of these decoding algorithms implemented using customized VLSI design.

3.2.3 Implementation decision

From the performance comparison and complexity comparison, it could be observed that the Log-MAP decoder complexity is no more than twice that of Max-Log-MAP decoder. However, the coding gain is approximately 0.6 dB better that Max-Log-MAP.

From the complexity analysis presented in Chapter 2 and [9], Log-MAP is about twice as complexity as SOVA. Although in our simulation, performance of SOVA is very close to that of Max-Log-MAP, it was reported in the literature to be worse than that of Max-Log-MAP. A performance degradation of 1 dB was reported in [9] of SOVA compared to Log-MAP. Furthermore, it was shown in [9] that Log-MAP is more suited to parallel processing, which suggests some advantages in VLSI design over the SOVA. Therefore, the final complexity of the Log-MAP decoding will not be so much larger than that of SOVA. Therefore, it is concluded that the Log-MAP decoding algorithm is the best compromise concerning both complexity and performance. Please be advised that this conclusion is not valid all the time. In situations where the complexity is the critical issue and performance could be sacrificed, Max-Log-MAP or SOVA could be the best solution.

3.3 Conclusions

In this chapter, we presented the turbo decoding simulation results using the four SISO decoding algorithms.

- Simulation of turbo coded BPSK transmissions in AWGN channel were performed to evaluate the coding gain performance of turbo decoding using the four SISO decoding algorithms.
- It is found that MAP and Log-MAP provides the best performance, which is about 8.6 dB coding gain at a BER of 10^{-5}. Turbo decoding using Max-Log-MAP and SOVA algorithms provides only about 8.0 dB coding gain at BER of 10^{-5}.

- With an interleaver of 1020-bit long, the turbo-coded system achieves performance at 1.55 dB away from Shannon limit with MAP decoding algorithm.
- Decoding complexity comparison was performed using the execution time of each SISO algorithm on the same simulation platform as the measure. It clearly indicated that the MAP decoding algorithm is much more complex than other SISO algorithms. Log-MAP is more complexity than Max-Log-MAP and SOVA has the smallest complexity. This is consistent with the results reported in various literatures.
- To obtain the optimal performance with feasible implementation complexity, we concluded that Log-MAP is the best decoding algorithm. It is therefore our choice in designing the VLSI turbo decoder in Chapter 4.

Chapter 4

Log-MAP Based Turbo Codec Implementation In VHDL

Hardware Description Languages (HDL) are used to design and document electronic systems. The advantages of using HDLs include:

- efficient description of large systems,
- automation of synthesis algorithms, and
- simulation of designs.

There are now two industry standard hardware description languages, VHDL and Verilog. VHDL (Very high speed integrated circuit Hardware Description Language) became IEEE standard 1076 in 1987. It was updated in 1993 and is known today as "IEEE standard 1076 1993". The Verilog hardware description language has been used far longer than VHDL and has been used extensively since it was launched by Gateway in 1983. Cadence bought Gateway in 1989 and opened Verilog to the public domain in 1990. Verilog became IEEE standard 1364 in December 1995.

It was shown in Chapter 3 that turbo decoder with Log-MAP algorithm is the best compromise between the implementation complexity and the decoding performance. In this chapter, we investigate how to implement the turbo decoder into an Application Specific Integration Circuit (ASIC) chipset, or a Field-Programmable Gate Array (FPGA). VHDL is used as the Hardware Design Language for design entry and functional simulation. A basic ASIC/FPGA design process [18] is shown in Figure 4-1.

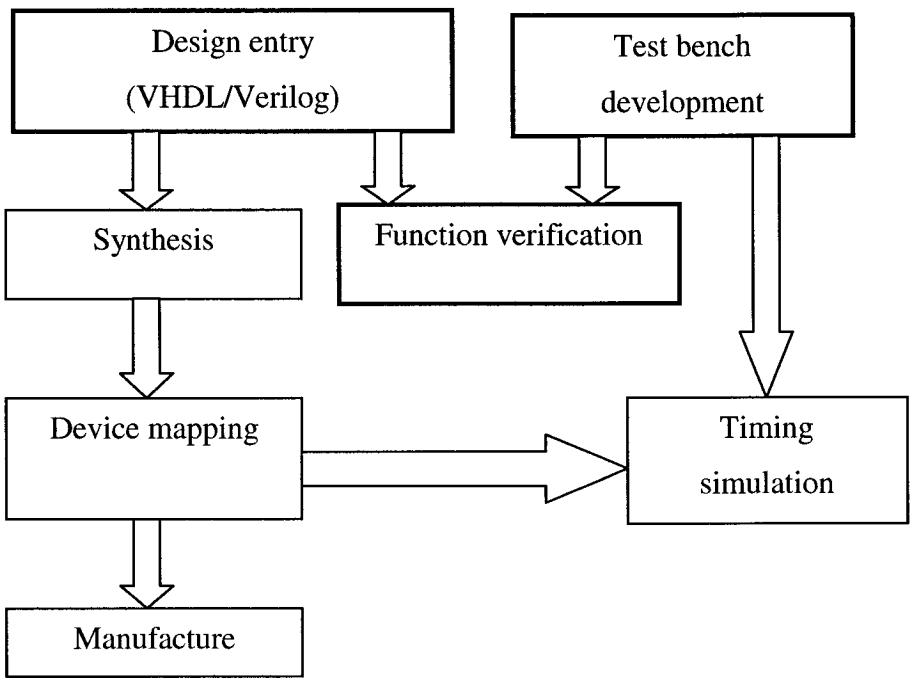


Figure 4-1 – ASIC/FPGA design process.

1) **Design entry**

In this step, the system interfaces and functionalities are defined. The detailed design is captured in VHDL, which provides useful programming features for structured design techniques and offers a rich set of control and data representation features. With these features, a complex design can be analyzed and partitioned into simpler implementation modules. Each module has its own definition of functionality and interface. Note that the higher-level implementation only depends on the interfaces of all the modules and is independent of each module's specific implementation. The modularity and hierarchy of VHDL makes it very similar to a high-level hierarchical programming language, such as C, and therefore very easy to manage complex design structure.

However, in VHDL, not all the program features are "synthesizable". To use the synthesis technology, the VHDL description has to be written in a style that is appropriate for synthesis.

2) **Test bench development**

The functionality of the VHDL design should be verified before going further into synthesis. Test bench software is developed for this purpose, which is also programmed in VHDL that provides the design entity with the stimulus and verifies the outputs. VHDL provides rich high-level programming features, which, although not useful for a synthesizable design, are very convenient for test bench programming.

3) **Functionality verification by VHDL simulation**

In this step, combinations of the inputs (stimulus) are fed into the design entity and the outputs are verified. Usually the stimulus and results are generated and saved into files before the VHDL simulation. The test bench will read in the stimulus, feed them into the design entity, obtain the outputs of the design entity and compare these outputs to the outputs that should be obtained.

To completely verify the functionality, all combinations of the inputs should be tested by the design entity. However, this is too complex when there are too many input combinations. Therefore, the functionality validation could be performed by simply comparing performance obtained from the VHDL simulation with that obtained from a high-level C program simulation. A properly designed high-level verification program should take into account of the mathematical limitation in a realistic hardware design including the finite resolution and limited dynamic range of the data representation.

4) **Synthesis**

Synthesis is a process of transforming a design specification into an implementation, i.e., converting an abstract design description into a hardware structure. This process is usually performed using the synthesis tools based on certain synthesis technology library provided by some ASIC and FPGA manufactures.

5) **Device mapping**

This process tries to find proper devices from a device library based on the synthesis results. In this phase, a timing model generation program provided by a device vendor or third party simulation model supplier could be used to generate the

accurate timing model of the design. The timing constraints defined by this timing model can then be applied in the VHDL simulation test bench.

6) **Timing simulation**

The timing model generated during the device mapping is combined into the test bench and the verification is performed again. When the design performs correctly with the timing model, it is ready to be manufactured. However, if the design fails with the specified timing model, the designer has to go back to the first step, modify the design and go through all these steps again until the design passes the timing simulation.

In this thesis, we implemented a turbo decoder and the corresponding test bench with VHDL. The functional verification is performed by comparing the decoding performance of the VHDL implementation with the C-program simulation with a corresponding dynamic range constraint. The turbo decoder module is implemented at Register-Transfer-Level (RTL) and the design description follows a proper coding style to make it synthesizable.

In section 4.1, we first investigate the implementation of a turbo encoder. Issues on the implementation of the Log-MAP based turbo decoder is discussed in section 4.2. The VHDL implementation of the Log-MAP based turbo decoder is presented in section 4.3. Further implementation optimization techniques are presented in section 4.4. Conclusions are given in section 4.5.

4.1 Turbo Encoder Design

In this section, we present the implementation of a turbo encoder. This turbo encoder consists of two 16-state identical constituent convolutional codes (CCs) separated by one interleaver.

The block diagram of the turbo encoder is shown in Figure 4-2. A block of K information bits are stored into a buffer (implemented by a RAM in our implementation) first. After all the bits are ready in the buffer, the first constituent code, CC_1, starts reading these bits out from the buffer and perform the encoding. During this process, the output of the counter is used to address the buffer, i.e., the bits are encoded sequentially

by CC_1. After CC_1 complete the encoding on the block, the counter is reset and the second constituent code, CC_2, starts reading the bits in an interleaved order.

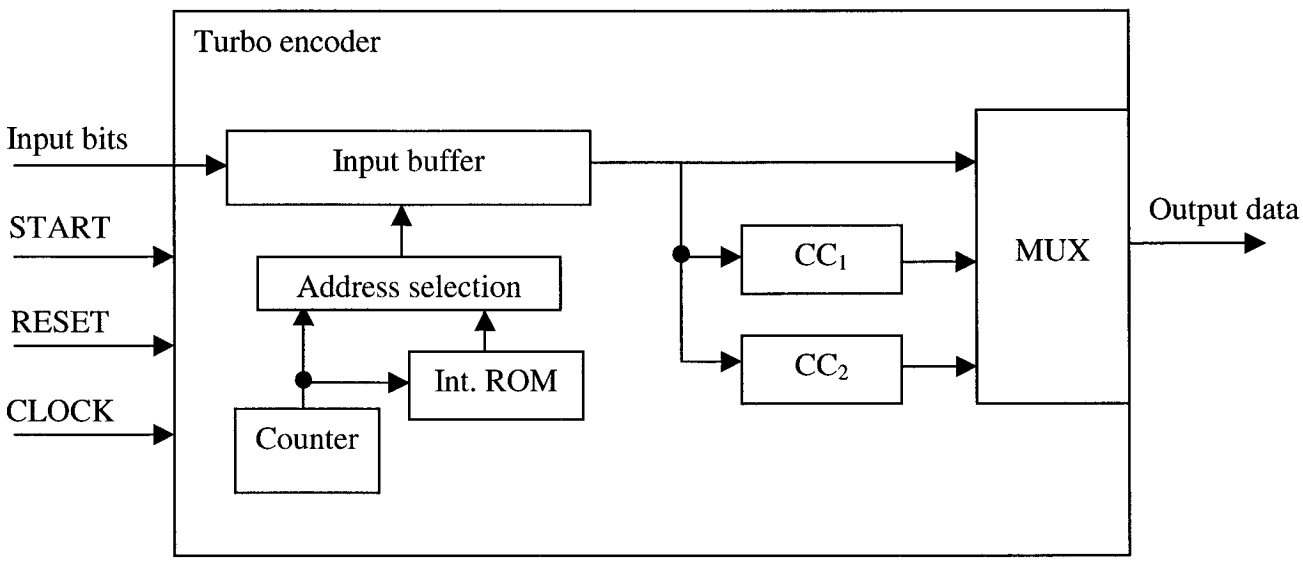


Figure 4-2 – Rate-1/3 turbo encoder block diagram.

The interleave pattern is stored in a Read-Only Memory (ROM), or an Erasable Programmable Read-Only Memory (EPROM). This time, the output of the counter is used to address the interleaving pattern ROM. The output of the ROM is the interleaved order and is used to address the input FIFO. Finally, the systematic bits from the buffer, the parity bits from CC_1 and the parity bits from CC_2 are multiplexed into a single coded bit sequence and are output into the channel.

4.1.1 Recursive Systematic Convolutional Code Implementation

The constituent encoder implementation is shown in Figure 4-3.

This 16-state convolutional encoder is implemented with a memory (M=4) shift register. The feed forward and feedback generator polynomials are implemented with eight AND gates. It is apparent that any generate polynomial could be obtained by adjusting the eight coefficients. Once being configured during the initialization process,

these parameters are fixed during the normal turbo encoding operations. The modulo-2 addition for the state updating and output calculation is implemented by an XOR gate.

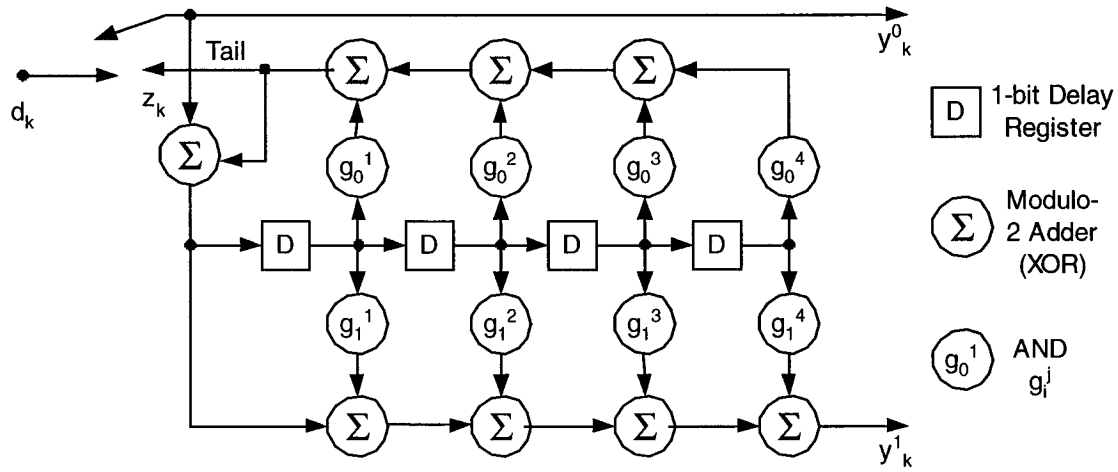


Figure 4-3 – Rate ½, 16-state RSC encoder.

In turbo coding, the input sequence is divided into blocks of K-bit. The encoder starts each block with a known state, usually state-0. It is also preferred that the encoder ends into a known state so that the backward recursion in the decoding has a known starting state. Setting the initial state to a known state is relatively simple and can be implemented by setting the encoder delay element to any value at the beginning of each block processing. When the encoding of the information bits is finished, the encoder is in a random state, depending on the actual information bit sequence. Setting the end state to some specific value requires some additional control. For example, in order to terminate the trellis into all-zero state, we have to make systematic bit d_k^s equal to the feedback term z_k. An apparent way is to append M=4 redundant systematic bits, each equals to the corresponding feedback term, to add a zero into the shift register. Only a few more NAND and XOR gates are required for this implementation. In a turbo encoder with two rate-½ CCs, when both encoders are terminated, there are totally $4 \cdot M$ redundant coded bits in the transmission vector. The bandwidth penalty (or transmission rate penalty) caused by these redundant bits is negligible when the block size is large enough. This encoder termination method by adding redundant bits is called bit-flushing method.

Another solution is to leave both CC encoders unterminated, where the state of the encoders are not known to the decoders at the end of each block. In this case, the SISO decoder usually assumes that the encoder has an equal probability to end in any state in starting the backward recursion. This, however, will reduce the reliability of the decisions on the bits at the end of the block. In [19], a turbo code with 132-bit interleaver was used to test the effect of terminating the CC encoders. The simulation results showed at a packet error rate (PER) of 10^{-3}, the performance of the turbo code with both CC unterminated is about 0.2 dB worse than that with one of the CC is terminated. Only about 0.05 dB further improvement can be obtained with both CC terminated. It was further shown in [20] that, for turbo codes with 1028-bit interleaver, although there are some performance difference in PER for turbo codes with non-terminated CCs and turbo codes with terminated CCs, there is very little difference between their BER performances, at BER above 10^{-7}.

Tail-biting is another method to force the CC encoder into a known state, where, it can avoid any transmission rate penalty. In tail-biting, the initial state of the CC encoder is decided by the first m information bits, where M is the length of the shift register in the encoder. The remaining $K–M$ information bits are then encoded with the initial state. At the end, M redundant bits are used to terminate the encoder state into the initial state. The implementation complexity of the termination is trivial compared to the turbo decoding process. Although M redundant bits are added, there are only $K–M$ information bits encoded and therefore no transmission rate penalty is introduced. In SISO decoding, the decoder will estimate the most likely initial state number to made decisions on the first M information bits. It was shown in [19] that using tail-biting to terminate both CC provides only marginal improvement over terminating one of the CC, at a PER of 10^{-3} or 10^{-4}. Furthermore, to accurately estimate the most likely state, the decoder needs to start the forward and backward metric recursion earlier in time (in a modulo sense since the starting state is the same as the ending state). The complexity involved is non-trivial anymore. For longer block length, this complexity becomes less significant. However, the advantage of the transmission rate saving compared to a double-termination by bit-flushing with redundant bits is also diminishing.

To obtain the optimum performance, we will terminate both encoders in our turbo encoder design. Since we are mainly looking at turbo codes with interleaver of about 1000 bits, the transmission rate penalty is negligible and therefore the double-termination by bit flushing is preferred because of its simplicity.

4.1.2 Interleaver Implementation in Turbo Encoder

The input data of the encoder is kept in a buffer that is implemented by a Random Access Memory (RAM). An interleaver accepts a sequence of data and output them in a rearranged order. The operation of an interleaver can be described as,

$$O(i) = I(\pi(i)) \tag{4.1}$$

where $\pi(i)$ is the input data index which is output at the i^{th} position. $\pi(i)$ is also called the interleaving pattern and is stored in the ROM. As shown in Figure 4-4, a counter is used to either address the RAM for the input data to the first CC, or address the ROM for the address of the input data to the second CC.

However, when only one RAM is used in the encoder, the subsequent block must wait until all the coded bits being sent out by the encoder before it can be processed. One possible solution to reduce this delay is to add an additional RAM. In this case, the first RAM is used for the encoding process, while the subsequent bits can be stored into the second RAM. Thereafter, two RAMs are used alternatively in the encoding process to store the next block.

4.1.3 Turbo Encoder Design Description in VHDL

The RTL model of this turbo encoder consists of two subsystems, the data subsystem and the control subsystem. The input signals to the data subsystem include the information data signals and the control signals generated by the control subsystem. It provides data path status signals to control subsystem and data output (code word). All of the interface signals and the encoding operations are synchronized by a unique clock signal, except the asynchronous RESET signal. The turbo encoder data path subsystem is shown in Figure 4-4 and the detailed description of the data path is presented in Appendix A.4. The control path of this turbo encoder is as shown in Figure 4-5.

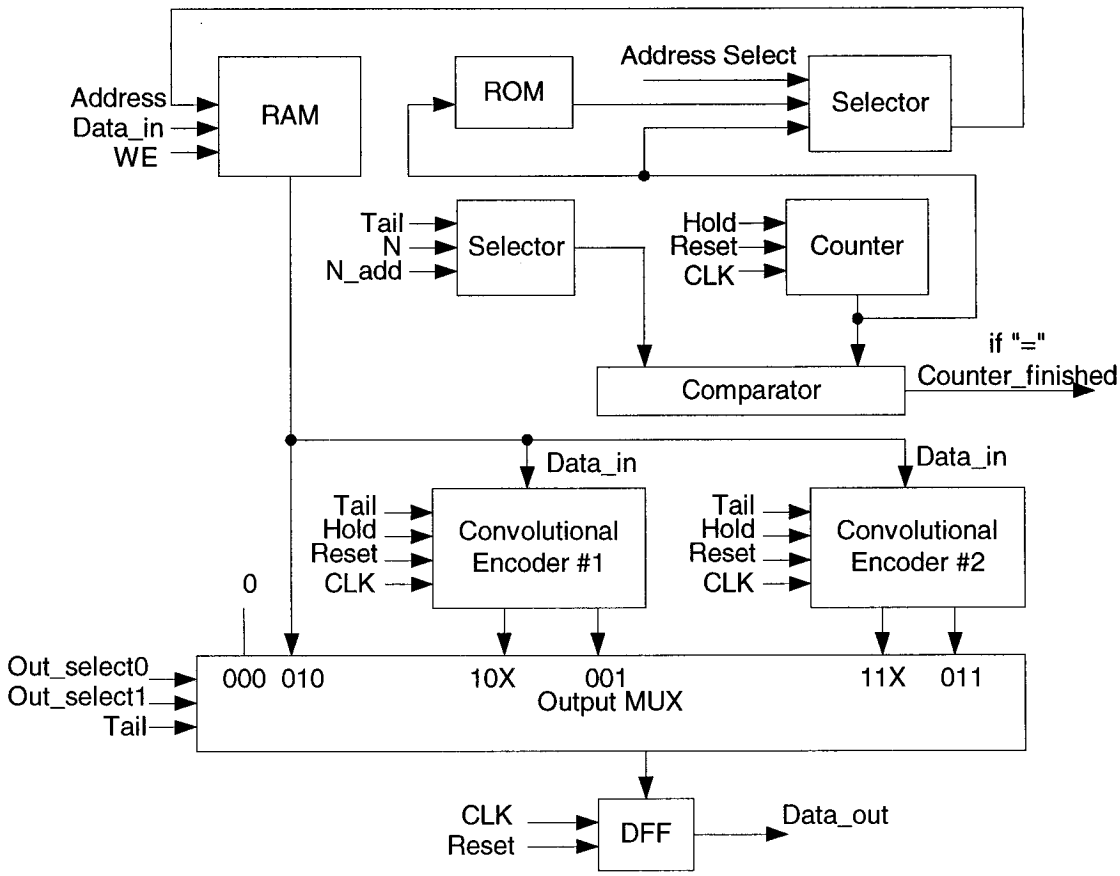


Figure 4-4 – Data path of turbo encoder

The data path and control path are designed to support a turbo encoder with the following operations procedures.

- In the first stage, which is called data_storing_stage, the encoder accepts K input data and stores them into an RAM. One input data is saved into the RAM in one clock period.
- The second stage, normal_encoding_stage, performs the encoding of the K input data. During this operation stage, the encoder outputs the systematic data, the parity data from the first encoder and the parity data from the second encoder sequentially. Outputting each bit takes one clock. Therefore, the total time duration required for this procedure is $3 \cdot K$ clock cycles.
- The third stage, termination_stage, performs the termination of both the convolutional encoders. Each encoder outputs not only the parity bits in this stage,

but also the tailing systematic bits. For the chosen convolutional encoders, assume that N_{add} tailing bits are required to terminate the trellis. For each tailing bit, the encoder will generate an additional parity bit. Therefore, N_{add} 4-bit groups are generated in the termination_stage.

- When the termination stage is over, the encoder will repeat this procedure for the next block of data.

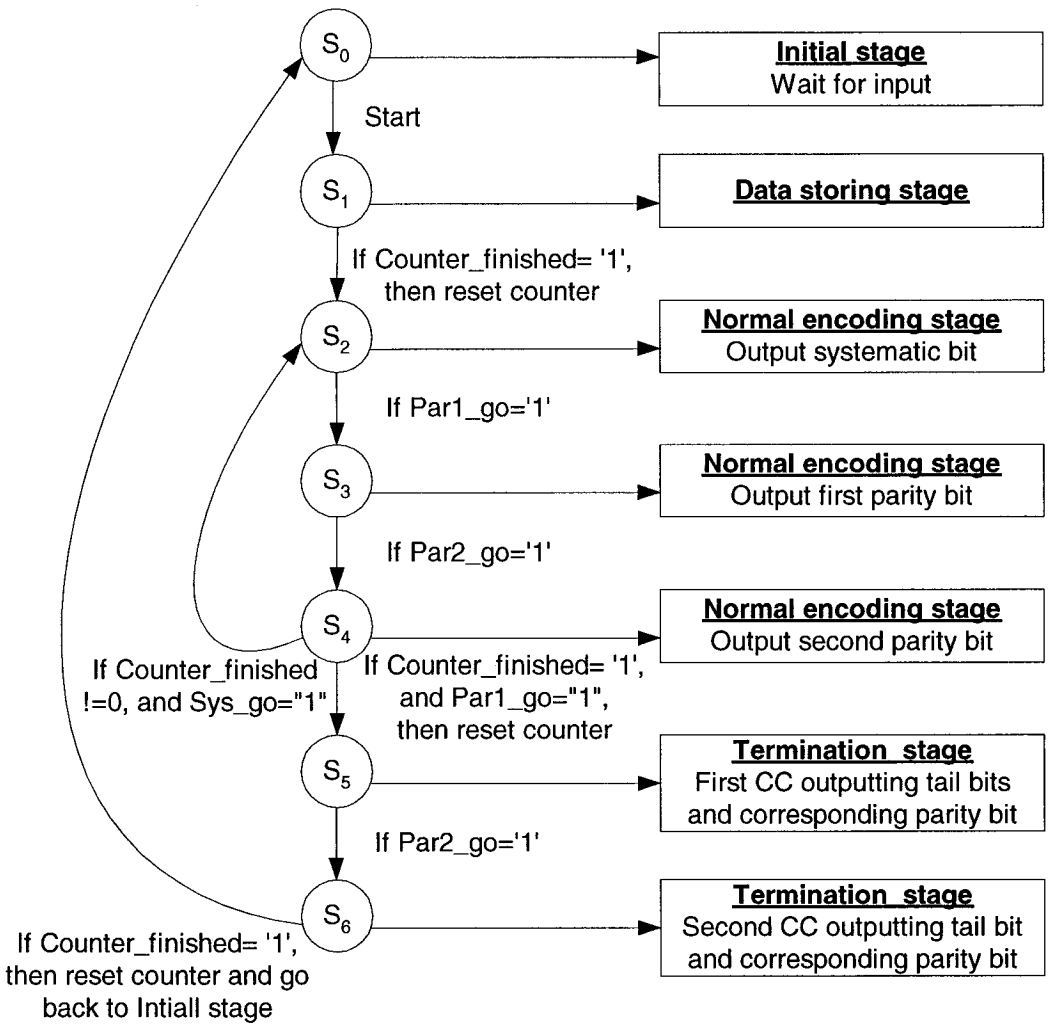


Figure 4-5 –Control path of turbo encoder.

4.1.4 Turbo Encoder Design Verification

A test bench for the turbo encoder is developed in VHDL and the simulation is performed with the SYNOPSIS software. A C-language simulation program is developed to generate random input sets and expected outcomes (test vectors). The RTL model of the turbo encoder is excited by the test bench with the pre-generated test vectors from C-language simulation programs. The outputs of the RTL model are then verified by the generated output expectation from the C simulations.

To perform the verification, the input data should be read from a file using the procedures and functions in VHDL "textio" package. However, for illustration purpose, a random sequence of length 16 is fixed in the test bench. The interleaving pattern is also fixed in the turbo encoder, defined as a memory type variable. The reverse-position interleaving is assumed, which means i^{th} input data will be at the K-i^{th} output data after the interleaving.

In addition to the output comparison, the state transitions of both encoders with the input sequence are also compared to the results obtained from the C-simulation step by step. The resulted state transition sequences from VHDL simulations and C simulations are exactly the same. This thus verifies that our VHDL implementation of the turbo encoder goes through correct path in the turbo code trellis and gives correct output bit sequence.

In Table 4-1, a block of 16 input information bits (d_k) and the output parity bits from the two CCs, x_k^{1p} and x_k^{2p}, are listed. To terminate each CC, at most 4 redundant systematic bits are added following the 16 information bits, which also generated 4 parity bits from this CC. Therefore, there are totally 16 tail bits, 4 systematic and 4 parity bits from each of the two CCs. These 16 redundant bits are shown in Table 4-2.

Table 4-1 – Turbo code encoder input and output sequences.

K	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
d_k	1	0	0	1	0	1	0	0	0	1	0	1	0	0	1	0
x_k^{1p}	1	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0
x_k^{2p}	0	1	1	0	1	0	0	1	0	1	1	0	0	0	0	0

Table 4-2 – Turbo code encoder tailing bits.

K	1	2	3	4
d_k^1	1	1	0	0
x_k^{1p}	1	1	0	0
d_k^2	0	1	1	0
x_k^{2p}	0	1	1	0

The VHDL simulation timing sequence is obtained and shown in Figure 4-6, where a clock signal of duration 100 *ns* is used. The timing sequence of the first 16 clocks are not shown in this figure since this period is only used to receive the 16 input bits and store them into the RAM. The signals in this figure can be explained as,

- "TUREN_T1/START": START signal for the turbo encoder
- "TUREN_T1/RESET": RESET signal for the turbo encoder
- "TUREN_T1/CLK": Clock signal for the turbo encoder
- "TUREN_T1/in_bit": Input bit sequence
- "TUREN_T1/U1/input_RAM": Input buffer
- "TUREN_T1/U1/out_co": Output coded bit sequence
- "TUREN_T1/U1/GIVE_OUTPUT/sys...": Turbo encoder output index
- "TUREN_T1/U1/GIVE_OUTPUT/REG_1...": The encoder state of CC_1
- "TUREN_T1/U1/GIVE_OUTPUT/REG_2...": The encoder state of CC_2
- "TUREN_T1/U1/GIVE_OUTPUT/add_bits...": The tail bit index

The encoder state transitions during the encoding processing of both CCs are shown and compared to those obtained from C simulation to verify the VHDL CC model executes in exactly the same it is supposed to. The output sequence, "out_co", is the concatenation of the multiplexed sequence $[d_1, x_1^{1p}, x_1^{2p}, d_2, x_2^{1p}, x_2^{2p}, \ldots, d_{16}, y_{16}^1, y_{16}^2]$ and the 16 tailing bits. It is very easy to see that the output sequence comply with the output bits shown in Table 4-1 and Table 4-2.

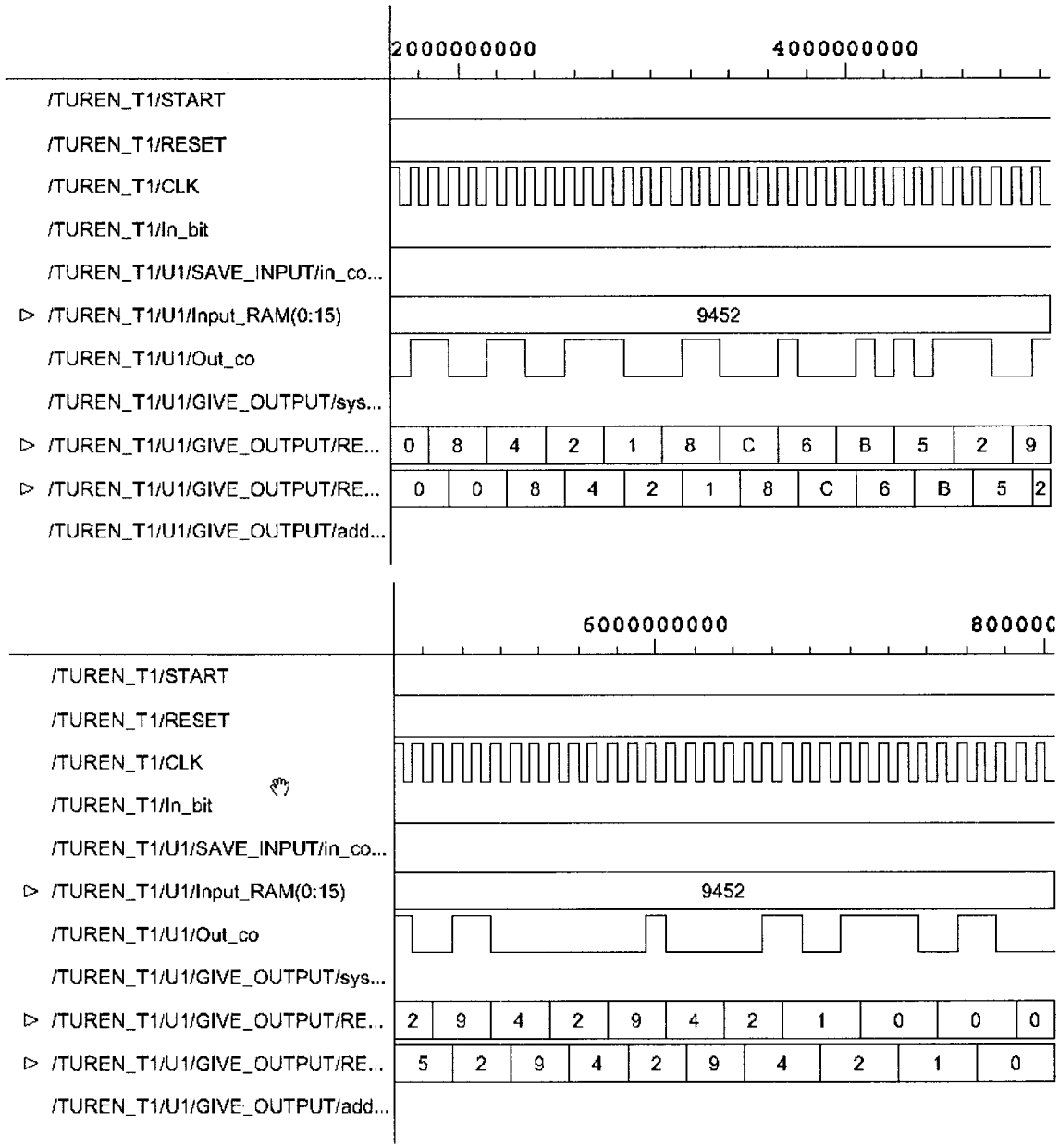


Figure 4-6 – VHDL turbo encoder simulation timing sequence.

4.2 MAP Based Turbo Decoder Design

This section describes the design considerations and detailed implementation of the turbo decoder.

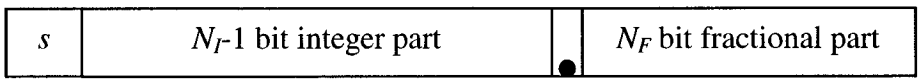
4.2.1 Implementation Considerations

In this section, we present some basic options on VLSI design of a turbo decoder and the rationales behind the decisions we made in our implementation.

4.2.1.1 Fixed-point vs. Floating-point Implementation

Turbo decoding algorithm involves large amount of mathematical computations on various data, including received symbols with noise, the a priori information, path metrics and so on. For digital hardware design, numbers are represented as either fixed-point or floating-point data types. For both these data types, word sizes are fixed at a set number of bits. The basic format of fixed-point data type and floating-point data type is shown in Figure 4-7.

(a) Floating-point representation



(b) Fixed-point representation

Figure 4-7 – General floating-point and fixed-point data representations

The floating-point representation consists of the exponent field, e (M_e bits), the sign bit, s, and the fractional field, f (M_f bits). The sign bit and the fractional field is usually considered as one field named mantissa (M_m=1+M_f bits). The exponent field, e, is a 2's complement number and the mantissa, m, represents a normalized 2's complement number (a normalized representation always has the most significant bit as the inverse of the sign bit). This (M_e+M_m)-bit floating-point representation is calculated as,

$$y_{float} = m \times 2^{e} \tag{4.2}$$

which has the following dynamic range and precisions,

$$\begin{aligned}
&\textit{Most positive}: && y = (2 - 2^{-M_f}) \times 2^{2^{M_e-1}-1} \\
&\textit{Least positive}: && y = 2^{-2^{M_e-1}+1} \\
&\textit{Least negative}: && y = (-1 - 2^{-M_f}) \times 2^{-2^{M_e-1}+1} \\
&\textit{Most negative}: && y = -2 \times 2^{2^{M_e-1}}
\end{aligned} \tag{4.3}$$

The fixed-point data format of length (N_B=N_I+N_F) is shown in Figure 4-7 (b). The binary vector follows the 2's complementation representation, where the first bit is the sign bit s and the rest bits decide the absolute value. N_I is the number of bits (including the sign bit) to the left of the point and it determines the dynamic range of the data; while N_F is the number of bits to the right of the point and decides the granularity of the data. This N_B-bit fixed-point format has the following dynamic range and precision:

$$\begin{aligned}
&\textit{Most positive}: && y = 2^{N_I-1} - \frac{1}{2^{N_F}} \\
&\textit{Least positive}: && y = \frac{1}{2^{N_F}} \\
&\textit{Least negative}: && y = -\frac{1}{2^{N_F}} \\
&\textit{Most negative}: && y = -2^{N_I-1}
\end{aligned} \tag{4.4}$$

It is apparent that, with equal word width, the dynamic range of fixed-point values is much smaller than floating-point values. However, using floating-point data types is accompanied by three major disadvantages:

- **Size and Power Consumption**

  The logic circuits of fixed-point hardware are much less complicated than those of floating-point hardware. This means that the fixed-point chip size is smaller with less power consumption when compared with floating-point hardware.

- **Memory Usage and Speed**

  In general, fixed-point calculations require less memory and less processor time to perform.

- **Cost**

  Fixed-point hardware is more cost effective where price/cost is an important consideration. When using digital hardware in a product, especially mass-produced products, fixed-point hardware costs much less than floating-point hardware and can result in significant savings.

Since a good ASIC design aims at a dedicated structure with a complexity as small as possible (small area, high throughput, low power consumption) while providing satisfactory decoding performance, apparently, fixed-point implementation is preferred when the same data bit-width is used. However, with the same bit-width, floating-point representation provides larger dynamic range and better resolution. This means that with floating-point representation, it is possible to use small bit-width to achieve the same dynamic range and resolution as compared to fixed-point representation. Smaller bit-width directly means smaller implementation area in an ASIC design because of the area reduction in wire routing. However, a pure combinational implementation of floating-point adder is much more complex than that of a fixed-point adder. We implemented a 8-bit floating-point adder and a 12-bit fixed point adder. It is observed that the 8-bit floating-point implementation is much more complex than its 12-bit fixed-point counterpart. Therefore, the fixed-point data type is chosen in our implementation.

Once we have decided to use the fixed-point data type in our design, the next step is to choose the proper data width N_B and a proper combination of N_I and N_F, since this is a crucial design issue in implementation using field programmable gate arrays (FPGA) and very large scale integration (VLSI) technologies. To properly select the values of N_B, N_I and N_F, we have to first take a look at their respective influences to the decoding performance.

4.2.1.2 Determination of The Fixed-point Data Type Width

In the implementation of a turbo decoder, several issues related to the finite-precision arithmetic are,

- the finite-precision representation of LLRs, which are originally real numbers,
- the finite-precision of the decoder arithmetic operations, and

- the integer representation of the *extrinsic* information involved in the iterative decoding process.

We will use FP(N_I, N_F) to denote the fixed-point format shown in Figure 4-7 (b). Following the notation used in [21], where N_I is called the dynamic of this fixed-point representation and N_F is called the precision.

It is shown in our simulation that the path metrics (α and β) keep increasing along the recursive calculation. In Figure 4-8, we plotted typical α values along the decoding trellis at the 10th iteration. It is shown that the maximum and minimum α values are at a range from –256 (which is pre-defined as the minimum value in the simulation program) to about 1600.

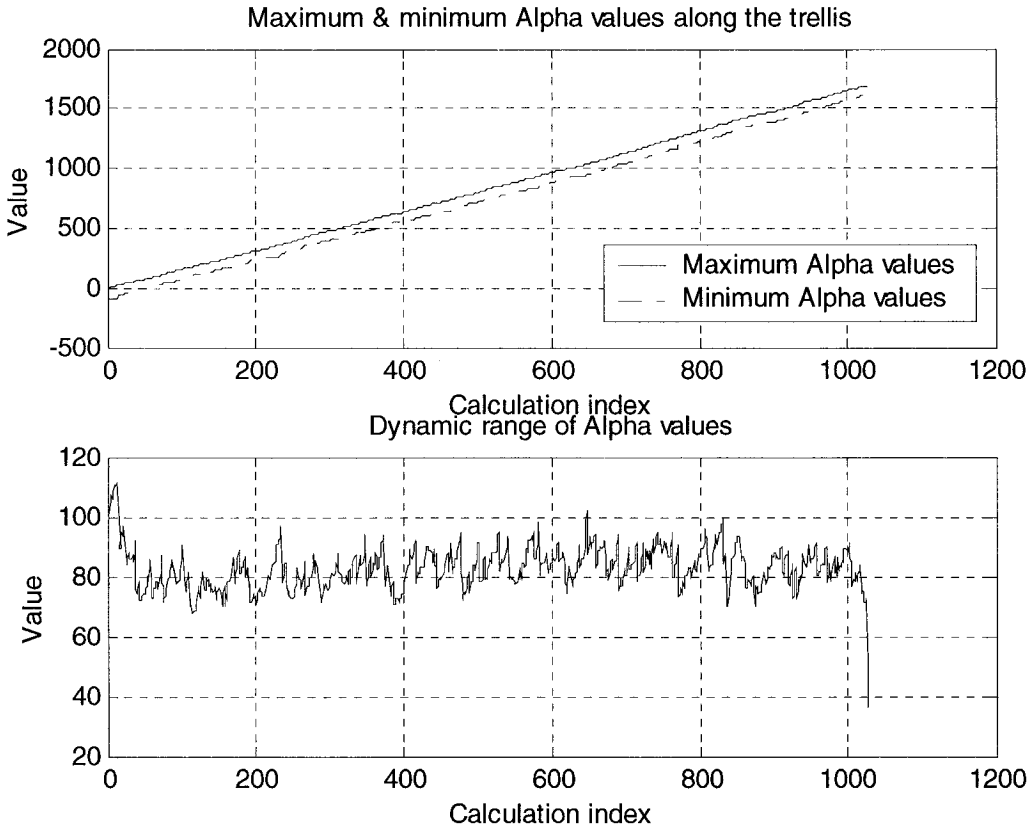


Figure 4-8 – Typical α values in a Log-MAP decoder in the 10th iteration.

Although the absolute value of α could be very large, the dynamic range is shown to be very well bound by 120. The dynamic range is defined as the difference between the

maximum value and the minimum value. For the backward recursive calculated path metric, β, same results were observed. Upper bounds of the dynamic range of path metrics were derived in [22] for Log-MAP decoding algorithm, where they are actually only valid for Max-Log-MAP algorithm. This was pointed out in [21] and the authors claimed that their simulations resulted in a similar dynamic range for Log-MAP algorithm. In the following, we will derive the upper bounds for the path metrics in our turbo decoder with the specified CC. We will further show that an upper bound of the dynamic range can be derived for Log-MAP algorithm by adding a modification factor to the path metric upper bound of the Max-Log-MAP algorithm.

In our simulations, a rate-1/2 16-state convolutional code is used as the CC. This guarantees that there is a valid path between any state at time k and any state at time k+4.

Without loss of generality, assuming at time k+4, the maximum α value is obtained at state-0, and the minimum α value at state-14, as shown in Figure 4-9.

The forward path metric, α, is calculated recursively as,

$$\alpha_k(s_k) = \max\left\{a_{k-1}(s_{k-1}^0) + \gamma(x, y, s_{k-1}^0, s_k),\ a_{k-1}(s_{k-1}^1) + \gamma(x, y, s_{k-1}^1, s_k)\right\} + f\left(|\Delta_a|\right). \quad (4.5)$$

with

$$\Delta_a = \left(a_{k-1}(s_{k-1}^0) + \gamma(x, y, s_{k-1}^0, s_k)\right) - \left(a_{k-1}(s_{k-1}^1) + \gamma(x, y, s_{k-1}^1, s_k)\right). \quad (4.6)$$

where, $s_{k-1}{}^0$ and $s_{k-1}{}^1$ are the two states that have transitions at time k–1 to state s_k at time k, and γ is the corresponding branch metric. We recall that the path metric of the survivor in VA is defined as only the first item in (4.5). The path metric in Log-MAP algorithm is therefore the path metric of the survivor path defined in VA plus a correction item. Following the definition of survivor path defined in VA, we define the path that is chosen by the *max* operation the survivor path and the other the concurrent path. Assume the survivor paths for $s_{k+4}{}^0$ and $s_{k+4}{}^{14}$ are shown in Figure 4-9 as the solid paths. Since there is at least one path connecting $s_k{}^0$ with $s_{k+4}{}^{14,}$ this path is a concurrent path and is discarded somewhere before time k+4. When this path is discarded at the last transition, as the path in dotted lines shown in , we have the following inequality,

$$\alpha_{k+3}(s^7) + \gamma_{k+3}(s^7) < \alpha_{k+3}(s^{15}) + \gamma_{k+3}(s^{15}). \quad (4.7)$$

with

$$\alpha_{k+4}(s^{14}) = \max\left\{\alpha_{k+3}(s^7) + \gamma_{k+3}(s^7), \alpha_{k+3}(s^{15}) + \gamma_{k+3}(s^{15})\right\} + f\left(|\Delta_a|\right). \qquad (4.8)$$

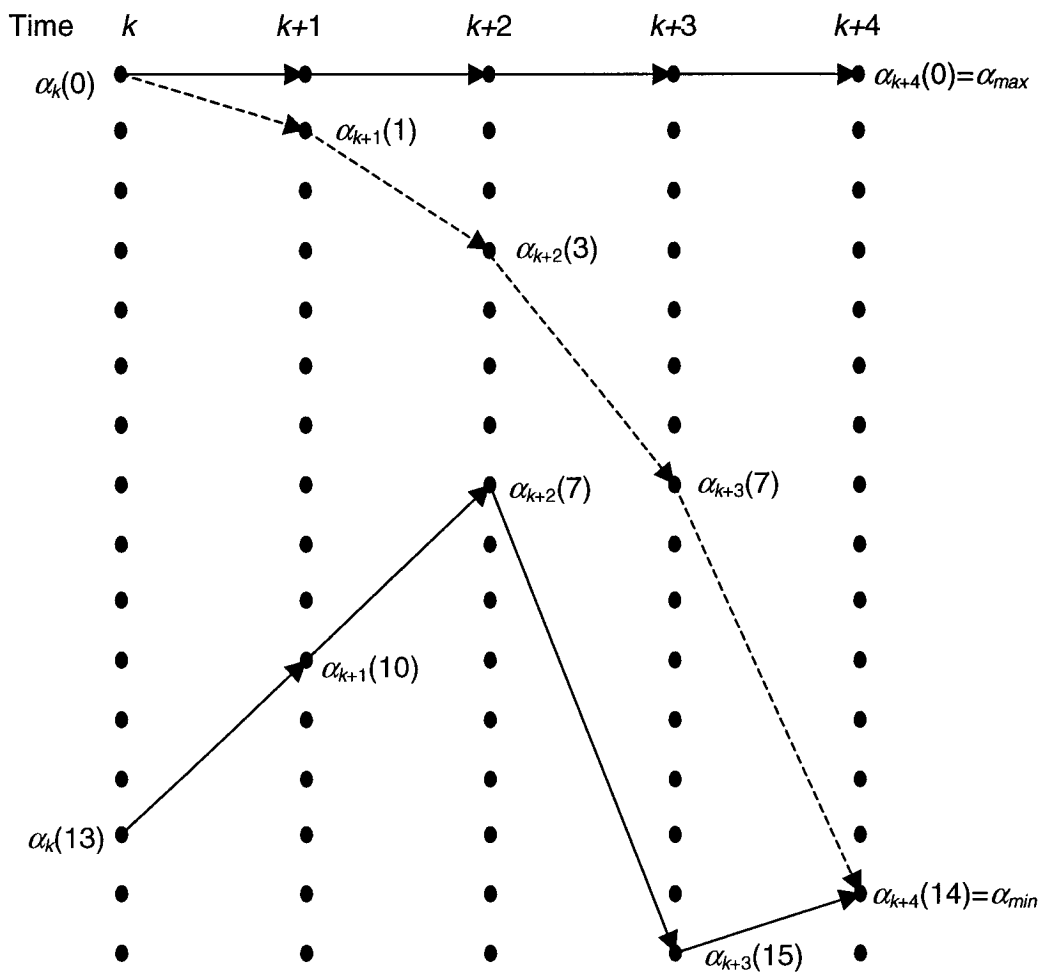


Figure 4-9 – Trellis of rate-1/2, 16-state CC in our turbo decoder implementation.

Define a virtual path metric, $\widetilde{\alpha}_{k+4}(s^{14})$, as,

$$\widetilde{\alpha}_{k+4}(s^{14}) = \alpha_{k+3}(s^7) + \gamma_{k+3}(s^7). \qquad (4.9)$$

The dynamic range of α at time k+4 is therefore bounded by,

$$\Delta_{\alpha}^{\max}(k+4) = \alpha_{k+4}(0) - \alpha_{k+4}(14) \le \alpha_{k+4}(0) - \widetilde{\alpha}_{k+4}(14). \qquad (4.10)$$

The same derivation applies when the concurrent path and the survivor for the state with the minimum α value merge before the k+4.

Now, the objective becomes finding the difference between the two paths originate from one state. Since these two paths start from the same state at time k, the difference between their path metrics at time k+4 is only caused by the four transitions happened during k to k+4. The maximum contribution of the 4 branch transitions to the survivor path [s^0, s^0, s^0, s^0, s^0] (denoted by the states it passes in the order of time) is,

$$\alpha_{k+4}(s^0) - \alpha_k(s^0) \le \sum_{i=0}^{3} \gamma_{k+i,survivor} + 4 \cdot \max_{\Delta_a} f\left(|\Delta_a|\right), \tag{4.11}$$

where $\gamma_{k+i,survivor}$ is the branch metric at the i^{th} branch after time k along the survivor path. The minimum contribution of the 4 branch transitions to the concurrent path [s^0, s^1, s^3, s^7, s^{14}] is,

$$\tilde{\alpha}_{k+4}(s^{14}) - \alpha_k(s^0) > \sum_{i=0}^{3} \tilde{\gamma}_{k+i} + 4 \cdot \min_{\Delta_a} f\left(|\Delta_a|\right) = \sum_{i=0}^{3} \gamma_{k+3}(s_{k+i}, s_{k+i+1}). \tag{4.12}$$

where $\tilde{\gamma}_{k+i}$ is the branch metric of the i^{th} branch starting from time k along the virtual concurrent path.

Therefore, the dynamic range of α with the assumed survivor path is bounded as,

$$\begin{aligned} \Delta_\alpha^{\max}(k+4 \mid s_k = 0) &< \sum_{i=0}^{3} \left(\gamma_{k+i,survivor} - \tilde{\gamma}_{k+i}\right) + 4 \cdot \max_{\Delta_a} f\left(|\Delta_a|\right) \\ &= \sum_{i=0}^{3} \left(\gamma_{k+i,survivor} - \tilde{\gamma}_{k+i}\right) + 2.78 \end{aligned}. \tag{4.13}$$

Each branch metric is calculated as,

$$\gamma_i((y_k^s, y_k^p), S_{k-1}, S_k) = \frac{y_k^s x_k^s(i)}{\sigma^2} + \frac{y_k^p x_k^p(i, S_k, S_{k-1})}{\sigma^2} + \ln \Pr\{S_k \mid S_{k-1}\} + K. \tag{4.14}$$

where,

$$\ln\left[\Pr\left(s_k \mid s_{k-1}\right) \mid d_k = 1\right] = \begin{cases} 0 & \text{when} \quad L_{apri} > 0 \\ L_{apri} & \text{when} \quad L_{apri} < 0 \end{cases}$$

and (4.15)

$$\ln\left[\Pr\left(s_k \mid s_{k-1}\right) \mid d_k = 0\right] = \begin{cases} 0 & \text{when} \quad L_{apri} < 0 \\ -L_{apri} & \text{when} \quad L_{apri} > 0 \end{cases}$$

The branch metric dynamic range is therefore upper-bounded by,

$$\Delta_\gamma \le \max\left(\left|\frac{2y_k^s}{\sigma^2}\right|\right) + \max\left(\left|\frac{2y_k^p}{\sigma^2}\right|\right) + \max\left(\left|L_{apri}\right|\right) = 2\cdot\lambda_c^{\max} + L_{\max}^{apri} = \Delta_\gamma^{\max}. \quad (4.16)$$

where $\lambda_c^{\max}$ is the maximum absolute value of the channel symbol LLRs, and L_{apri} is the clipping level of the *a priori* information.

It was shown in both [22] and [21] that clipping the *a priori* term in (4.16) at four times that of $\lambda_c^{\max}$ does not affect the decoding performance, we have,

$$\Delta_\gamma \le \Delta_\gamma^{\max} = 6\cdot\lambda_c^{\max} \quad (4.17)$$

The dynamic range of path metric can therefore be bounded by,

$$\Delta_\alpha^{\max} \le 4\cdot 6\cdot\lambda_c^{\max} + 2.78 \quad (4.18)$$

For a maximum SNR of 5 dB and rate 1/3 code, which is a pretty high value for our simulations, $\lambda_c^{\max}$ is calculated as 4.2. Therefore, the dynamic range of α is,

$$\Delta_\alpha^{\max} \le 4\cdot 6\cdot 4.2 + 2.78 = 103.58 \quad (4.19)$$

For Log-MAP, the *extrinsic* information (L_e), or the LLR (L), is calculated as, [22]

$$L = \ln\left[\sum_{\substack{s_{k-1},s_k \\ u_{k-1}=1}}\left(e^{\alpha(s_{k-1})}\cdot e^{\gamma(s_{k-1},s_k)}\cdot e^{\beta(s_k)}\right)\right] - \ln\left[\sum_{\substack{s_{k-1},s_k \\ u_{k-1}=0}}\left(e^{\alpha(s_{k-1})}\cdot e^{\gamma(s_{k-1},s_k)}\cdot e^{\beta(s_k)}\right)\right] \quad (4.20)$$

Assuming $\Delta_\alpha^{max} \le \Delta_\beta^{max}$, the absolute value is therefore bounded by,

$$\begin{aligned}
|L| &\le \ln\left[e^{\alpha_{\max}}\cdot e^{\gamma_{\max}}\cdot\sum_{\substack{s_{k-1},s_k \\ u_{k-1}=1}} e^{\beta(s_k)}\right] - \ln\left[e^{\alpha_{\min}}\cdot e^{\gamma_{\min}}\cdot\sum_{\substack{s_{k-1},s_k \\ u_{k-1}=1}} e^{\beta(s_k)}\right] \\
&= \Delta_\alpha^{\max} + \Delta_\gamma^{\max} + \left[\ln\left(\sum_{\substack{s_{k-1},s_k \\ u_{k-1}=1}} e^{\beta(s_k)}\right) - \ln\left(\sum_{\substack{s_{k-1},s_k \\ u_{k-1}=0}} e^{\beta(s_k)}\right)\right] \\
&= \Delta_\alpha^{\max} + \Delta_\gamma^{\max} + \left[\ln\left(\sum_{s_k} e^{\beta(s_k)}\right) - \ln\left(\sum_{s_k} e^{\beta(s_k)}\right)\right] \\
&= \Delta_\alpha^{\max} + \Delta_\gamma^{\max}
\end{aligned} \quad (4.21)$$

which can in turn be expressed as:

$$|L| \le \min\left[\Delta_\alpha^{\max}, \Delta_\beta^{\max}\right] + \Delta_\gamma^{\max}. \quad (4.22)$$

and the dynamic range is therefore calculated as:

$$\Delta_L^{\max} \le 2 \cdot \left(\min\left[\Delta_\alpha^{\max}, \Delta_\beta^{\max} \right] + \Delta_\gamma^{\max} \right). \tag{4.23}$$

For a SNR of 5 dB, the dynamic range is calculated as,

$$\Delta_L^{\max} \le \Delta_\alpha^{\max} + \Delta_\gamma^{\max} \le 257.6 \tag{4.24}$$

Equation was derived from two assumed survivors for the largest and smallest α values. The thereafter equations were derived assuming the maximum branch metric difference could be obtained for all intermediate branches of the two path, i.e., the Hamming distance between the survivor path and the concurrent path equals to the path length. Therefore, this Log-MAP decoding algorithm internal computation dynamic range is independent of the CC's specific coding structure. It only depends on the constraint length of the CC. Looking into the specific coding structure could provide tighter upper bound. In our example, the Hamming distance is smaller than their length. The maximum distance is obtained when these two paths have all the systematic bits to be complementary to each other, when large contribution could be obtained from the *a priori* information, which is usually much larger than channel LLRs. The final upper bound is obtained without any assumption of the survivor path, which is calculated as,

$$\Delta_\alpha^{\max}(k+4) < \max_{s_k, s_{k+4}} \left\{ \alpha_{k+4}(0) - \tilde{\alpha}_{k+4}(14) \right\} \tag{4.25}$$

However, it can be shown that the result will have very limited difference from that obtained using with the assumed CCs in our simulation.

Without the correction factor of 2.78, this is exactly the upper bound for α and β values in Max-Log-MAP algorithm. Since this correction term is so small compared to the other contributors, it can be concluded that the dynamic ranges for Log-MAP and Max-Log-MAP are almost the same and so are their data-width requirements.

With the dynamic range given in (4.24), at least 9 bits are required for the integer part of the internal fixed-point data. For best performance, we will choose 9 bits of dynamic range in our design.

This result is valid for the last turbo decoding iteration where the *a priori* information tends to be very large, resulting in very large absolute values of branch metrics. However, it is shown in our simulation that the dynamic range of path metrics changes significantly during different decoding iterations. The dynamic ranges were observed to be only 20 in

the 1st iteration, less than 90 in the 4^{th} iteration and up to 120 in the 10^{th} iteration, as shown in Figure 4-10. In [21], this observation is applied in a pipeline implementation of turbo decoder, where some dedicated SISO decoders are always used during some specific iteration. For those SISO decoders assigned to the early iterations, much shorter data width could be employed with negligible performance degradation, taking advantage of the much smaller dynamic range. The decoders in later decoding stage, however, have to use the data width decided above and they take larger area than the early-stage SISO decoders.

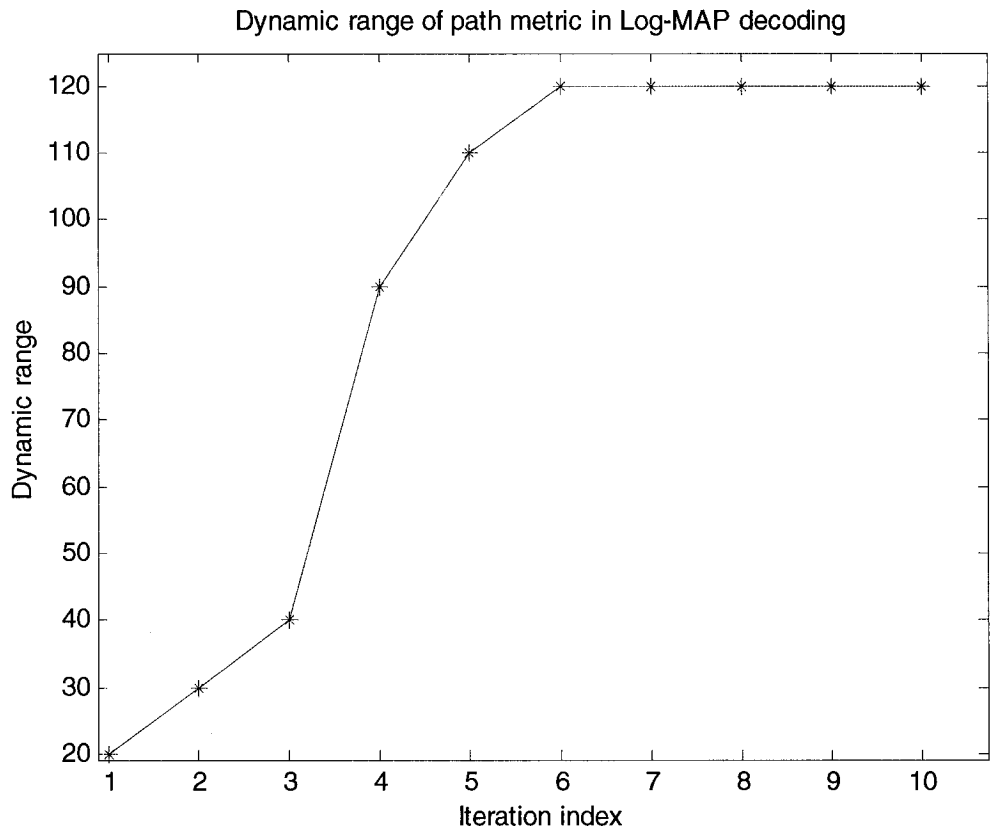


Figure 4-10 – Dynamic range of the path metrics in Log-MAP decoding.

The precision requirement of a fixed-point turbo decoder was investigated in [21] and [24] by simulations. In [24], almost perfect decoding could be achieved by using 3 bits of precision for a Log-MAP turbo decoder. It is further shown in [21] that 2 bits of precision is enough with negligible performance degradation. Up to now, there is no proof these results are code independent since they are all obtained from turbo codes with specific

CCs. Because of this, we will be using 3 bits of precision in our design for best results. Further optimization is required in the future to find out if less precision could still give tolerable performance.

The total number of bits in our design is 12 bits per fixed-point data, where 9 bits are for dynamic range and 3 bits for precision, i.e., FP(9,3).

4.2.1.3 Path Metric Normalization in Turbo Decoder

It has been shown that the path metric could be very high, where the dynamic range is very well bounded, as shown in Figure 4-8. To minimize the required number of bits for the data representation, the path metrics at any time should be normalized. In this section, we discuss several path metric normalization methods existing in the literature and the one we choose to implement in our turbo decoder.

4.2.1.3.1 Existing Normalization Methods

Several VLSI architectures for metric normalization in the Viterbi algorithm were discussed in [25], with respect to their implementation advantages and disadvantages. Since the path metrics calculation in Log-MAP algorithm follows the similar process as the Viterbi algorithm, these normalization methods can be applied to the Log-MAP algorithms without any modification.

1) **Trellis reset**

This technique periodically forces the encoder to a *ground* state, zero state for example, by adding redundant data. The decoding trellis at the receiver will therefore converge into the ground state periodically, where the decoder can reset the path metric to zero. When this technique is employed, the reset period has to be decided such that the accumulated path metrics would not exceed the maximum value that can be represented by the data type.

The major disadvantage of this method is the redundancy introduced to reset the trellis, which results in either a bandwidth loss or a transmission rate reduction. Furthermore, although this technique is relative simple to implement, the path metric updating procedure is interrupted periodically because the periodical termination of the trellis. This results in more control logic to be implemented.

2) **Variable shift**

In this method, the minimum path metric is subtracted from all of the path metrics. The path metric calculation is then performed as following:

- *All the path metrics are calculated.*
- *A comparator is used to determine the smallest path metric.*
- *The smallest path metric is subtracted from all the path metrics.*

In the implementation, a subtractor is required to perform the comparison during the path metric computation, while the smallest value is determined and stored into a register. After all the path metrics are obtained, a new set of path metrics are calculated by subtracting the smallest value from all the path metrics.

$$\alpha'(s_k) = \alpha(s_k) - \min_{s_k}[a(s_k)] \tag{4.26}$$

Employing this normalization technique requires the smallest bit-width requirement. However, although the logic of this method is straightforward, the VLSI implementation requires both intercommunications between different path metric updating modules (to calculate the minimum path metric) and interrupt in the path metric updating (to subtract the minimum path metric). Furthermore, this technique introduces large processing delay or large number of additional hardware since the normalization has to be performed after the smallest metric is available. The smallest metric is only available after all pre-normalization path metrics have been obtained.

3) **Fixed shift**

In this technique, all the path metrics are shifted by a fixed value when they are all positive (or negative). The sign bits of the path metrics are used as the indication of when a shifting is needed. This technique is relatively simpler to implement compared to the variable shift technique. However, since the sign bit is used as an indication, it requires one extra bit for the dynamic range.

4) **Modulo normalization**

The Modulo normalization method was first proposed in [26]. In this technique, the path metrics, $\alpha(s_k)$, are replaced by the normalized metrics, $\alpha_m(s_k)$,

$$\alpha_m(s_k) \equiv \left[\left(\alpha(s_k) + \frac{D}{2}\right) \bmod D\right] - \frac{D}{2} \tag{4.27}$$

where D is the dynamic range of the fixed-point representation. It was proved that the differences of the survivor metrics remain unchanged using modular arithmetic provided that $\Delta_{max} \leq D/2$, where Δ_{max} is dynamic range of the path metric.

For the purpose of illustration, these normalized values can be thought as evenly-distributed positions on a circle, with $\alpha_m(s_k)$ at angle $2\pi\alpha_m(s_k)/D$. The direction of increasing corresponds to counterclockwise motion around the circle. Since the dynamic range of the path metric is smaller than $D/2$, they all concentrate in half a circle, as shown in Figure 4-11. Since all the values are limited in half a circle and the data increasing direction is counterclockwise, it is very easy to determine the maximum and minimum path metrics.

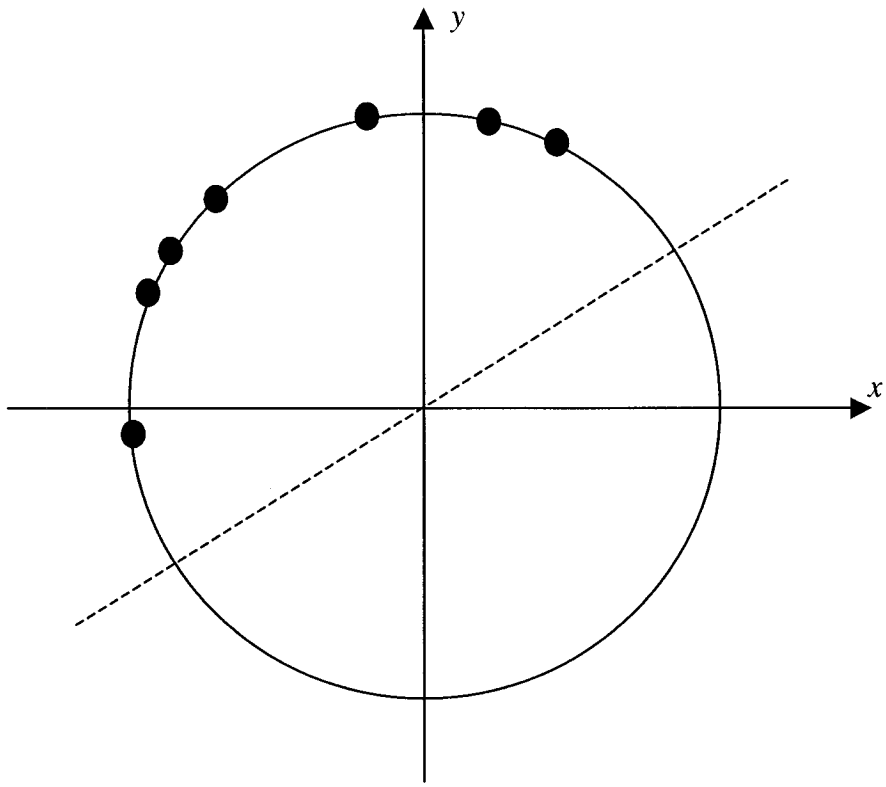


Figure 4-11 – Modulo representation of fixed-point path metrics.

The modular arithmetic can be implemented by 2's-complement adders and subtractors. This technique requires neither additional control logics nor global signal communications between different path metric updating modules and is therefore a preferred technique. However, same as the fixed-shift technique, it requires the fixed-

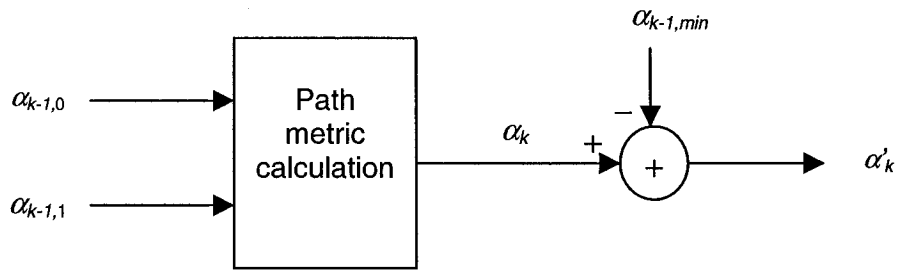
point representation with a dynamic range that is twice the dynamic range of the actual path metric.

4.2.1.3.2 Modified Variable Shift Normalization Method

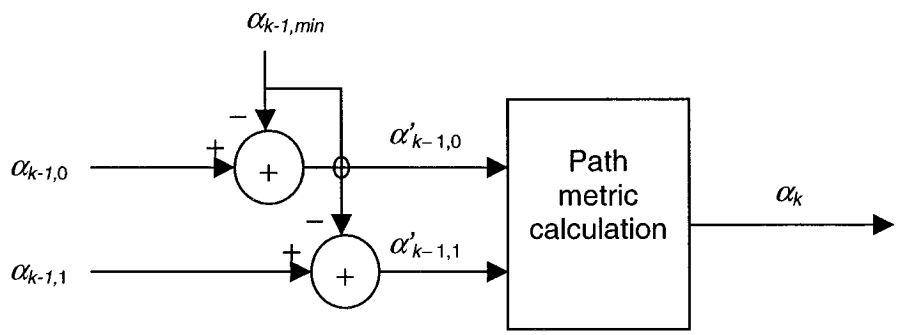
In Log-MAP decoding, for each set of channel input symbols (corresponding to the encoder output symbols resulted from each state transition), 2^M path metrics are updated. In our implementation, for area-efficiency, a single path metric updating module is used to compute all the path metrics sequentially, while the α path metrics are written into a RAM sequentially through a single data path (data port). Assuming path metric calculation is a single-clock operation, this requires at least 2^M clocks to complete the 2^M path metric calculations and memory writing.

To minimize the width of the fixed-point data, we choose the variable shift normalization method. However, employing this normalization method directly in our implementation will introduce both complexity and delay increase since the current minimum path metric can be obtained only after all the path metrics are available. Efficient implementation will be using one additional subtractor to perform the normalization sequentially. This introduces one additional subtractor with a delay of 2^M additional clocks.

A more efficient implementation is to calculate the minimum path metric value in the $(k-1)^{th}$ time interval and use it during the path metric updating in the k^{th} time interval. In this way, the path metric updating is always performed based on the normalized previous path metrics and the additional delay is avoided. For a rate-1/2 CC, each new path metric is calculated from two previous path metrics. Therefore, in this normalization method, two additional fixed-point subtractors are required to subtract the last-time minimum path metric from last-time path metrics when calculating the current metrics. The implementation of variable path metric normalization and modified variable path metric normalization methods are both shown in Figure 4-12.



(a) Variable shift path metric normalization for **one** path metric.



(b) Modified variable shift path metric normalization for **one** path metric.

Figure 4-12 – Variable and modified path metric normalization scheme.

4.2.1.4 Influence of Quantization on Turbo Decoding Performance

In a turbo decoder, it has been shown in Figure 2-3 the inputs to an SISO decoder are the analog systematic symbol sequence, parity symbol sequence and the digital *a priori* information sequence generated by the other SISO decoder. The analog systematic and parity symbols have to be first scaled by the signal-to-noise ratio (SNR) and then digitized by an analog-to-digital converter (ADC) before they are input into the SISO decoding. A finite precision ADC introduces quantization noise to the received signals, which can be modeled as an additional AWGN noise. This, of course, results in performance degradation in the turbo decoder. To obtain a satisfactory performance, an ADC with sufficient number of quantization levels has to be used. However, high-speed high-resolution ADC is both power-hungry and expensive. Therefore, it is important to find the minimum resolution requirement of the ADC in a turbo-coded system.

The influence of quantization was discussed in [21], [27], [28] and [29]. In [27], the analysis was based on turbo decoder with MAP algorithm. The internal computations in MAP decoder are performed with 32-bit floating point computation. The input symbols (after SNR scaling) to the MAP decoder are quantized. It was shown that with a 6-bit quantization, the turbo decoder gives nearly optimum performance close to that obtained using unquantized channel inputs. The 6-bit quantized inputs are used as a fixed-point data of format FP(3,3), i.e., a dynamic range of [-2, 2] and 3-bit precision.

In [28], negligible performance degradation was observed with an ADC of 5-bit with a dynamic range of [-2, 2], using one less bit for precision compared to that in [27].

In [21], a same conclusion was obtained that a 5-bit quantization with FP(3,2) is the best choice for the scaled channel inputs.

It was further shown in [29] that by applying an optimal scaling on the analog signal before they are input into the ADC, the quantization resolution requirement can be further decreased to 4-bit.

Therefore, we conclude here that a 5-bit quantization corresponding to a fixed-point representation FP(3,2) could be used at the input to the turbo decoder with negligible performance degradation due to quantization noise.

In our turbo decoder implementation, for implementation simplicity, the channel inputs will be represented by the same data format as the internal data, i.e. FP(9,3). The dynamic range is much higher that that is required and the 3-bit precision also guarantees no performance degradation from quantization noise.

4.2.1.5 Channel Estimation

This decoder design is under the assumption that the value of the noise power is known. Therefore, the exact noise variance σ^2 is used in the simulation and implementation of the turbo decoder. The APP information can be translated into metrics that can be implemented in some suitably efficient manner in decoder.

While the derivation of the APP requires detailed knowledge of the channel, the requisite knowledge may not be available for applications in reality, such as some mobile applications. Therefore, the proper channel estimation is necessary. The range of channel variation should be taken into account for the practical system.

In future work, the estimation of the channel parameters could be included within the decoding process [31], [32]. In an AWGN channel, by including the following estimation in (4.28) on the AWGN noise variance into the decoding process, any impact from an initial error in estimating the channel variance could be rapidly eliminated for AWGN channel and BPSK signaling.

$$\sigma^2 = \left[\frac{1}{k}\sum_{t=1}^{k} y_t^j\right]^2 + \left[\frac{1}{k}\sum_{t=1}^{k}(y_t^j - \frac{1}{k}\sum_{t=1}^{k} y_t^j)^2\right] - 1 \qquad (4.28)$$

The effect of SNR mismatch to turbo decoding is investigated in [33]. Simulation results shown the turbo decoding is very robust against SNR mismatch. The performance degradation of a turbo decoder caused by an SNR mismatch is very small when the SNR is under-estimated by no more than 3 dB and over-estimated by no more than 6 dB. When the mismatch is beyond this range, the performance degradation will be significant and the SNR has to be re-estimated.

4.2.2 Area Efficient vs. Delay Efficient Turbo Decoders

The logic diagram of a turbo decoder in an AWGN channel is shown in Figure 4-13.

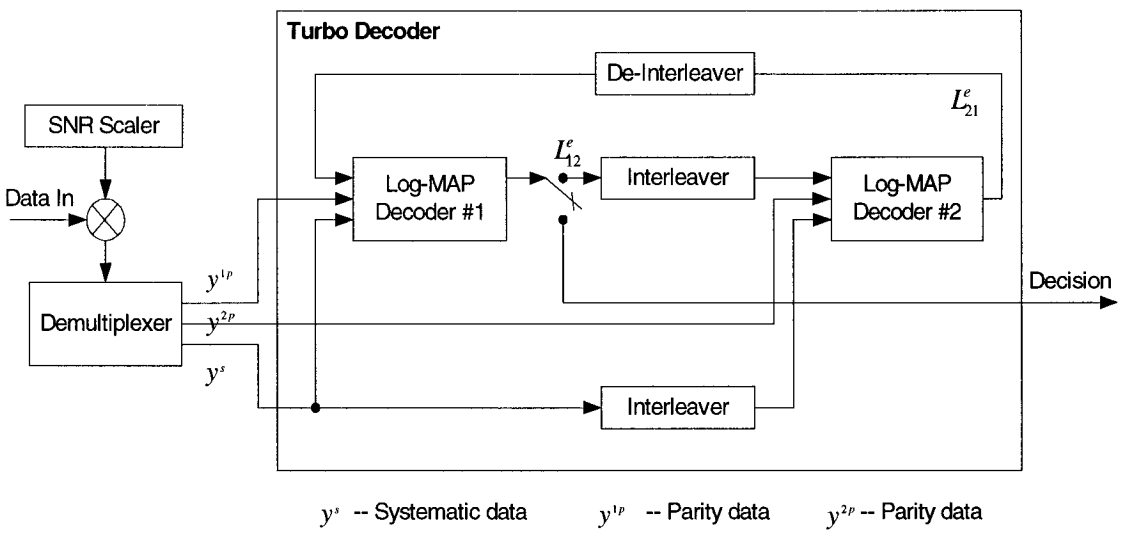


Figure 4-13 – Logic diagram of a turbo decoder.

The input channel data is first scaled by a factor proportional to the channel SNR. Since turbo decoding is not sensitive to the SNR estimation, the scaling is usually

performed on the analog inputs before the ADC. However, to avoid significant performance degradation caused by large variation in the noise (or received signal power), the SNR should be estimated by the turbo decoder, which will provides a control signal to the gain scaling circuit.

The digital signal output from the ADC is then demultiplexed into one systematic and two parity data sequences. These sequences are then written into three data buffers from which the turbo decoder can read and process them.

As shown in Figure 4-13, inside the turbo decoder, there are logically two different SISO decoding processes, corresponding to the two CCs in the turbo encoder. During the first iteration, the first SISO decoder accepts the input systematic data sequence, the first parity data sequence and the de-interleaved *extrinsic* information generated by the other SISO decoding during last iteration as *a priori* information, which is zero in the first iteration. The output is the *extrinsic* information sequence that will be interleaved and used by the second SISO as *a priori* inputs. The second SISO decoder takes the interleaved systematic sequence, the second parity data sequence and the interleaved *extrinsic* data sequence generated by the first SISO. The output will be de-interleaved and feedback to the first SISO decoder as *a priori* information.

Therefore, the major components contained in a turbo decoder are two SISO (Log-MAP in our case) decoders, 2 interleavers and 1 deinterleaver.

- **Delay Efficient Implementation**

  The iterative decoding structure can be implemented in either a delay efficient way or an area efficient way. To achieve short delay, two SISO decoders are used in a manner of pipelining. In this case, the first SISO decoder, SISO-1, starts with the first received turbo block.

  After SISO decoder finishes, the second SISO decoder SISO-2 will starts the second SISO decoding process of the current decoding iteration, using the output *extrinsic* information generated by the SISO-1. At the same time, SISO-1 starts the decoding of the next turbo block.

  After this decoding process, the turbo decoder finishes the first iteration decoding (consisting of two SISO decoding) on the first turbo block and half of the first iteration of the second turbo block. At this time, the SISO-1 starts the second turbo

decoding iteration of the first turbo block and the SISO-2 starts the second half of the first iteration on the second turbo block. In this way, the SISO-1 always performs the first SISO decoding in each iteration for every turbo block and the SISO-2 always performs the second SISO decoding in each iteration.

The decoding process continues in this pipelining mode until it finishes all the iterations of the first turbo block and starts the decoding of third turbo block.

This decoding halves the decoding delay since two SISO decoders operate completely in parallel. However, besides implementing two SISO decoders, a total of 4 RAMs are needed to store the intermediate *extrinsic* outputs from the two SISO decoders in order to achieve correct RAM reading and writing operations.

This method can be extended to using more than two SISO decoders to further decrease the decoding delay.

- **Area Efficient Implementation**

  In an area efficient implementation, the two logical SISO decoders can be implemented by one physical SISO decoder that alternately operate over the different set of input sequences corresponding to the two CCs in the turbo encoder. In this case, only one RAM is needed to store the intermediate *extrinsic* information, because only one SISO decoding process is active at any time.

  However, in this case, the decoding delay is doubled since all SISO decoding processes are performed serially.

4.2.3 Specifications of our Turbo Decoder

During the turbo decoder design, we have to make a compromise between the implementation complexity (hardware area) and the transmission delay. For low transmission rate and high-density applications, the turbo decoder should be designed to minimize the ASIC chipset area, by reusing the hardware components as much as possible. Since usually a hardware component can perform the function for one specific module only, different modules sharing this component has to use it sequentially. This results in long system delay. However, providing a separate copy of the same hardware component to each module that needs it can significantly reduce the delay since these modules can operate in parallel. Because that the design contains multiple copies of same

function components, the implementation complexity (reflected directly by the chipset area in ASIC design) is increased.

In this thesis, we designed an area-efficient turbo decoder, where the hardware components are shared between different design modules as much as possible. The fundamental design specifications of our turbo decoder are:

- The turbo decoder operates in a block-wise mode.
  Each time, the turbo decoder will process one block of received data and the next block will not be processed until the decoding of current block is complete. This can minimize both the number of processing (computation) units and the size of required memory.
- The turbo decoder uses only one Log-MAP module.
  During the turbo decoding process of each block, the same Log-MAP decoder is reused by all the decoding processs during all the iterations.
- Interleaving/deinterleaving is implemented by a reading-from/writing-into a single RAM. A Read-Only Memory (ROM) contains the interleaving pattern. No additional deinterleaving pattern is required.
- Inside the Log-MAP decoder, the branch metric calculation module is required by both the forward and the backward path metric calculations. Since no overlapped operation is performed to reduce the system delay, which means that the backward recursion begins only at the end of the forward recursion. Only one branch metric calculation module is implemented.
- A further area reduction could be obtained by implementing a generic module for the forward and backward path metric calculation, when the forward path metric calculation and the backward path metric calculation share a same hardware module.
- Some other implementation decisions are made to ease the development effort. These will be described in the following sections.

4.3 An Area Efficient Turbo Decoder Implementation

In this section, we present and discuss our area-efficient turbo decoder implementation in details.

4.3.1 Block Diagram of Turbo Decoder

In our turbo decoder implementation, we assume that the received analog signals are scaled by the SNR factor and then digitized. As discussed in section 4.2.1.2, the 12-bit fixed-point binary sequence is used for data representations in the turbo decoder internal computation. The turbo decoder implementation block diagram is shown in Figure 4-14.

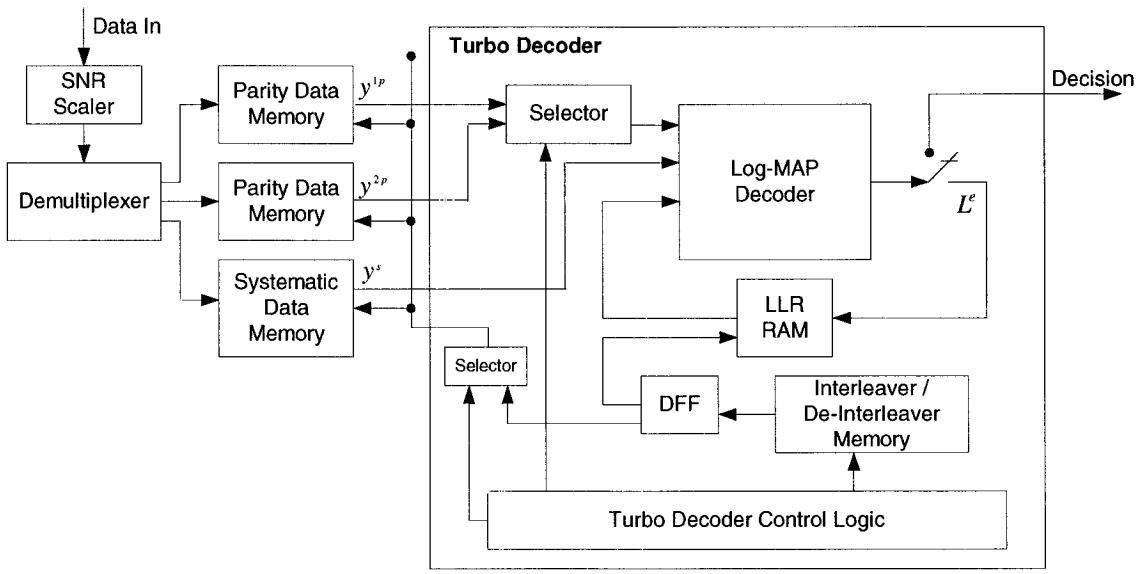


Figure 4-14 – High-level block diagram of turbo decoder.

As shown in Figure 4-14, the digitized sequence is then demultiplexed into three sequences, one systematic data sequence and two parity data sequences. There are three RAMs used as the three buffers, from which the turbo decoder reads its inputs. The demultiplexer, the three RAMs are implemented outside of the turbo decoder.

The turbo decoder consists of a Log-MAP decoder, an *extrinsic* data RAM, a ROM storing the interleaving pattern and some operational control logic circuits. The Log-MAP decoder alternately reads the two parity sequences to perform the first and second SISO decoding in each turbo decoding iteration.

The interleaving and deinterleaving processes are performed implicitly by reading from or writing into the *extrinsic* RAM according to the interleaving pattern stored in the interleaving ROM.

During the first SISO decoding of each iteration, the Log-MAP decoder reads the systematic data RAM, the first parity data RAM and the *extrinsic* data RAM (*a priori* information) sequentially. A synchronized counter, named block counter, is used to generate the address for the RAM reading operation. The generated *extrinsic* information is also written into the *extrinsic* RAM using the same counter output for addressing.

During the second SISO decoding of each iteration, the Log-MAP decoder reads the systematic data RAM and the *extrinsic* data RAM in an interleaved pattern, while it reads the second parity RAM in normal order. A same counter is used to generate the address for the parity RAM reading during this process. To read the systematic and *a priori* data out in the interleaved order, the output of this counter is used to drive the address bus of the interleave ROM. The output of the ROM is then used as the address to read the systematic data RAM and *extrinsic* data RAM. The output *extrinsic* data is written into the *extrinsic* RAM following the interleaved order. The deinterleaving is therefore simply achieved when the next SISO decoder reads the *extrinsic* RAM sequentially.

The control logic of the turbo decoder is shown in Figure 4-15. The operation sequence is explained as follows,

- Initial State – The turbo decoder is initialized before it starts decoding a received block. During this step, all its components with memory, including the Log-MAP decoder, the block counter and registers are initialized. The interleave ROM is initialized with the desired pattern, and the *extrinsic* RAM is filled with all zeros. The turbo decoder is waiting for the "START" signal to start the decoding.
- MAP Decoder #1 Start – When a logic-1 is detected on the "START" signal, the block counter is enabled and the Log-MAP decoder starts reading data from systematic RAM, first parity RAM and the *extrinsic* RAM. The addresses of these three RAMs are the output of the block counter. A logic-1 is put on the "Log-MAP START" signal to start the Log-MAP decoding process.
- MAP Decoder #1 Calculation – During this state, the turbo decoder logic circuit provides "Log-MAP START" signal to the Log-MAP decoder periodically. When a logic-1 is detected on the "Log-MAP START" signal, the Log-MAP module reads the current inputs from the three data RAMs and performs the corresponding decoding process. When there is a logic-1 on the "Log-MAP

Output Valid" signal, this *extrinsic* output will be written into the *extrinsic* RAM sequentially, i.e., non-interleaved order. After decoding over the entire block is finished, when all the *extrinsic* data have been written into the *extrinsic* RAM, a "Log-MAP Decoding Finished" signal will be provided by the Log-MAP module. When the current decoding iteration is the last iteration, the decoder goes to the Decoding-Finished state and the LLR is calculated instead of the *extrinsic* information. The LLR values are not written into the *extrinsic* RAM, while the sign of the LLR is output via the output "Decision" port of the turbo decoder.

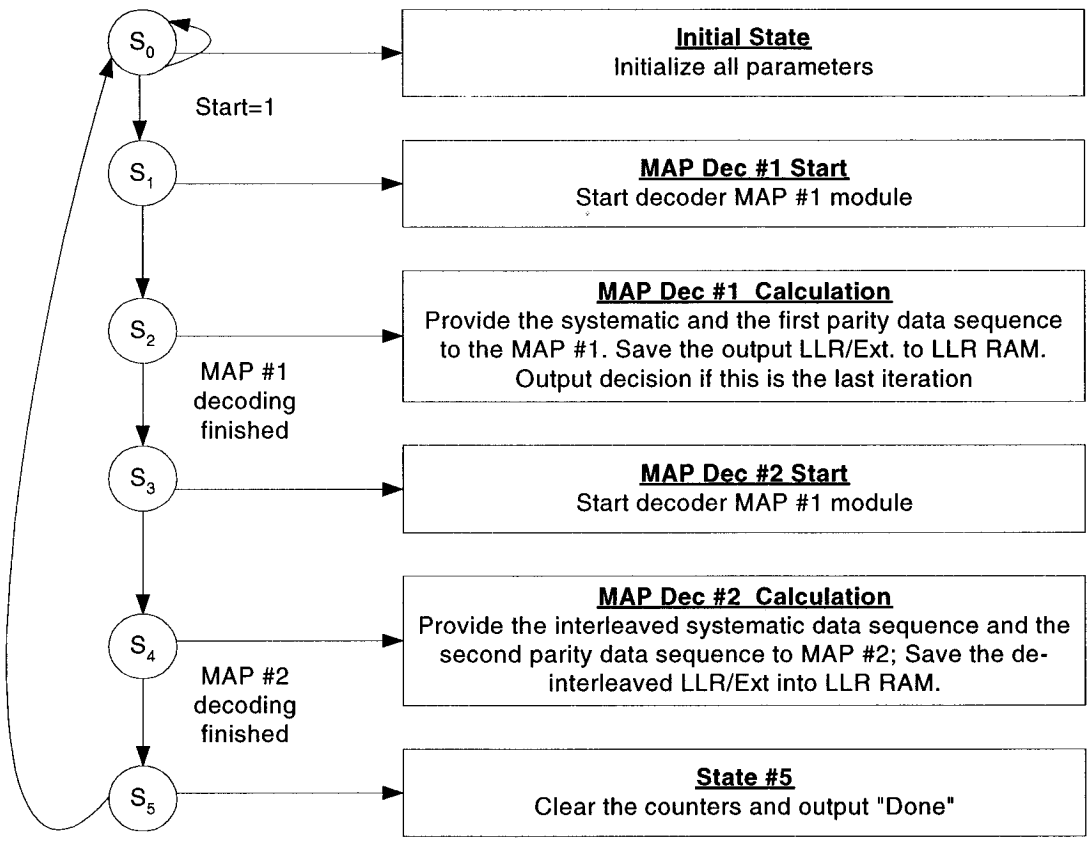


Figure 4-15 – State diagram of turbo decoder.

- MAP Decoder #2 Start – Upon detection the "Log-MAP Decoding Finished" signal, the block counter is reset and re-enabled. This time, the parity data input of the Log-MAP decoder is configured as the data from the second parity RAM. The address bus of the parity RAM is still driven by the output of the counter. The

address bus of the systematic RAM and *extrinsic* RAM are driven by the output of the interleave ROM, whose address is address-driven by the block counter. With all these configurations done, a logic-1 is put on the "Log-MAP START" signal to start the second Log-MAP decoding process during this turbo decoding iteration.

- MAP Decoder #2 Calculation – This process is the same as the previous decoding process, except for the RAM reading and writing logic. When the decoding process is finished, the turbo decoder is switched into the MAP Decoder #1 Start state.
- Decoding-Finished – The turbo decoder reset the counters and the other components, put a logic-1 onto the "Turbo Decoding Done" signal and switch into the "Initial state".
- At any time, the turbo decoder can be returned into "Initial-State" by asserting the asynchronous "RESET" signal.

4.3.2 Log-MAP decoder

The most computation and control intensive component in a turbo decoder is the SISO decoding module. The Log-MAP algorithm is chosen as the preferred SISO algorithm in our turbo decoder implementation. In this section, we present the implementation of the Log-MAP decoder.

4.3.2.1 Implementation considerations

The functional block diagram of a Log-MAP decoder is shown in Figure 4-16, based on the Log-MAP decoding algorithm discussed in section 2.2.1.3. The major components include,

- a branch metric calculation module – γ-calculation,
- a forward path metric calculation module – α-calculation,
- a backward path metric calculation module – β-calculation,
- a LLR calculation module,
- a table lookup module – LUT,
- a RAM to store the intermediate α values – α-RAM,

- control logic to decide the decoding status and generate the control signals,
- a synchronous counter to indicate the state index – state counter, and
- a synchronous counter to indicate the input data index in one turbo block – block counter.

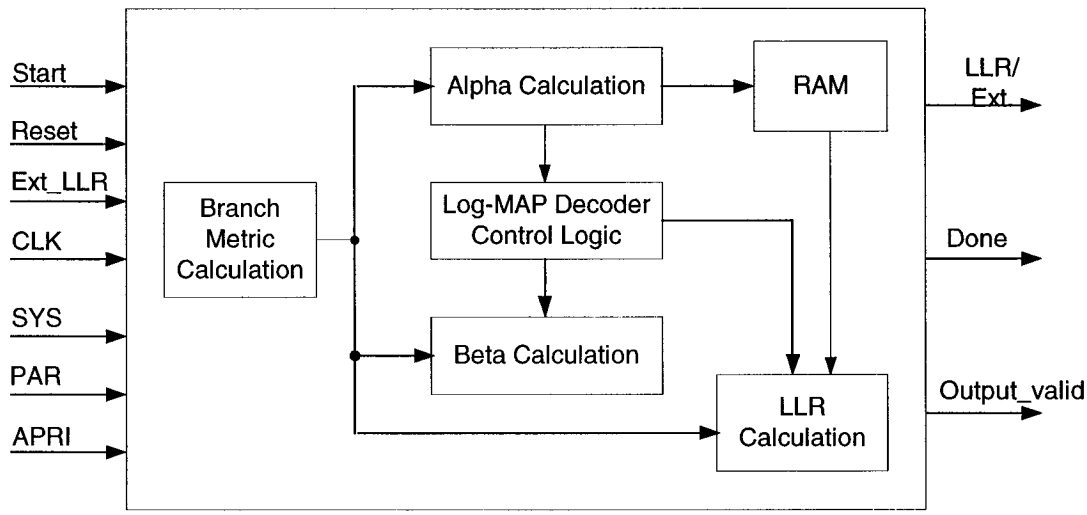


Figure 4-16 – Block diagram of Log-MAP decoder.

The data path of the Log-MAP decoder is shown in Figure 4-17. It can be observed that,

- A combinational γ-calculation module receives the input systematic, parity symbols (after SNR scaling) and a priori data and outputs the branch metrics for the current trellis state transition. There are two sets of 4 branch metrics output from this module. One set of γ values is calculated with the systematic, parity symbols and a priori data for the calculation of α, β and LLR. The other set of values are calculated with only the parity, which is used by the LLR calculation module to compute the *extrinsic* information. The input signal "Ext_LLR" is used to choose which set of γ values are used by the LLR-calculation module.
- The α-calculation module receives the outputs from the γ-calculation module. For every set of input symbols, 2^M α values are generated and written into the α-RAM sequentially.

- The β-calculation module receives the outputs from the γ-calculation module. For every set of input symbols, 2^M β values are generated and written into 2^M 12-bit registers, which will be read by the LLR-calculation to calculate the next *extrinsic* or LLR value.

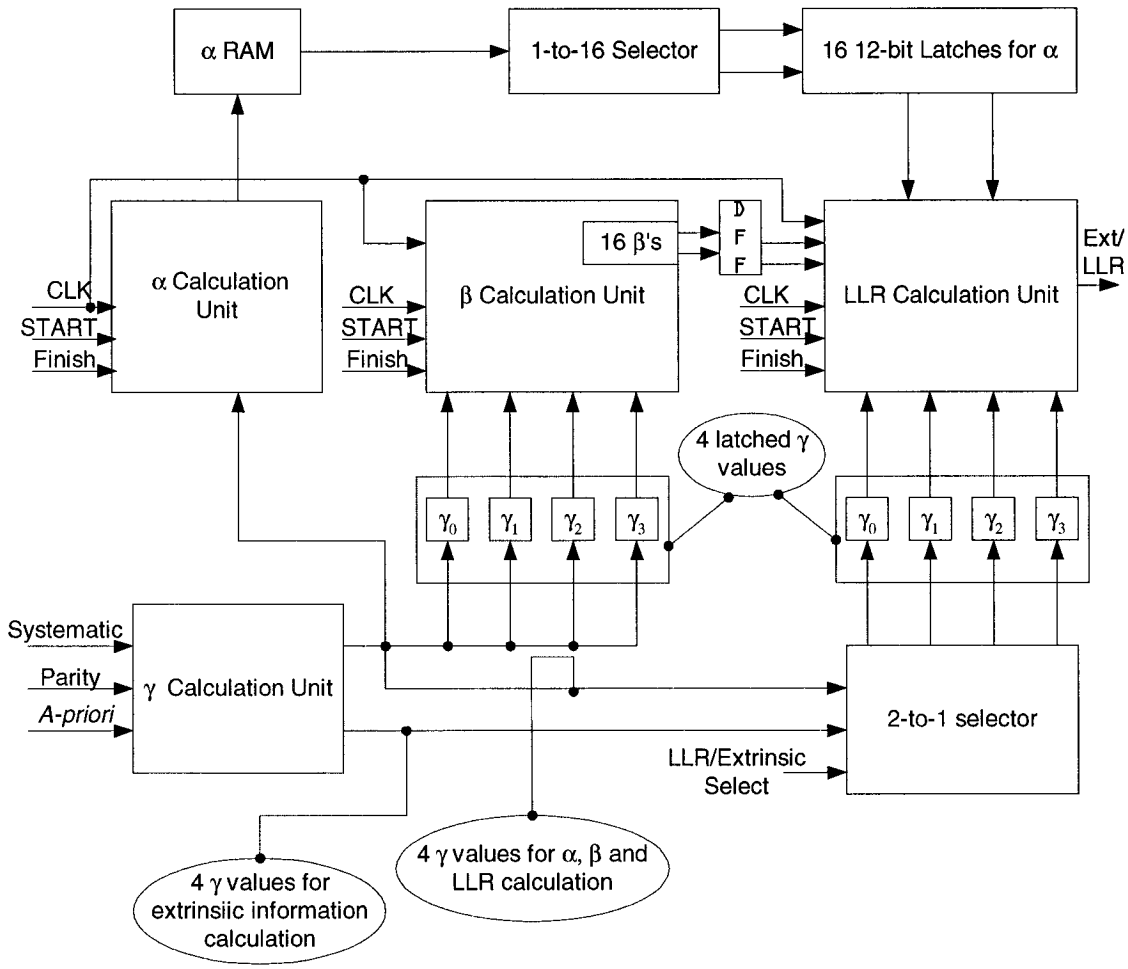


Figure 4-17 – Implementation of Log-MAP decoder.

- LLR-calculation takes 2^M α values, 2^M β values and a set of 4 γ values to calculate the *extrinsic* information or the LLR. During the current LLR/*extrinsic* calculation, the Log-MAP control logic is responsible for reading the α values from the α-RAM into 2^M 12-bit registers to be used in next LLR/*extrinsic* calculation.

- At the last Log-MAP decoding in the last iteration, LLR is calculated instead of the *extrinsic* information and the sign of the LLR is output as the decision.

The state diagram of the Log-MAP decoder is shown in Figure 4-18. The decoding operation can be described as:

- "Log-MAP-Idle" – Before the Log-MAP decoder starts the decoding, it goes in "Log-MAP-Idle" state. All the component modules are in reset state. The Log-MAP decoder switches into "Log-MAP-Initial" state after it stays in this state for one clock. This is to guarantee that all components are reset.

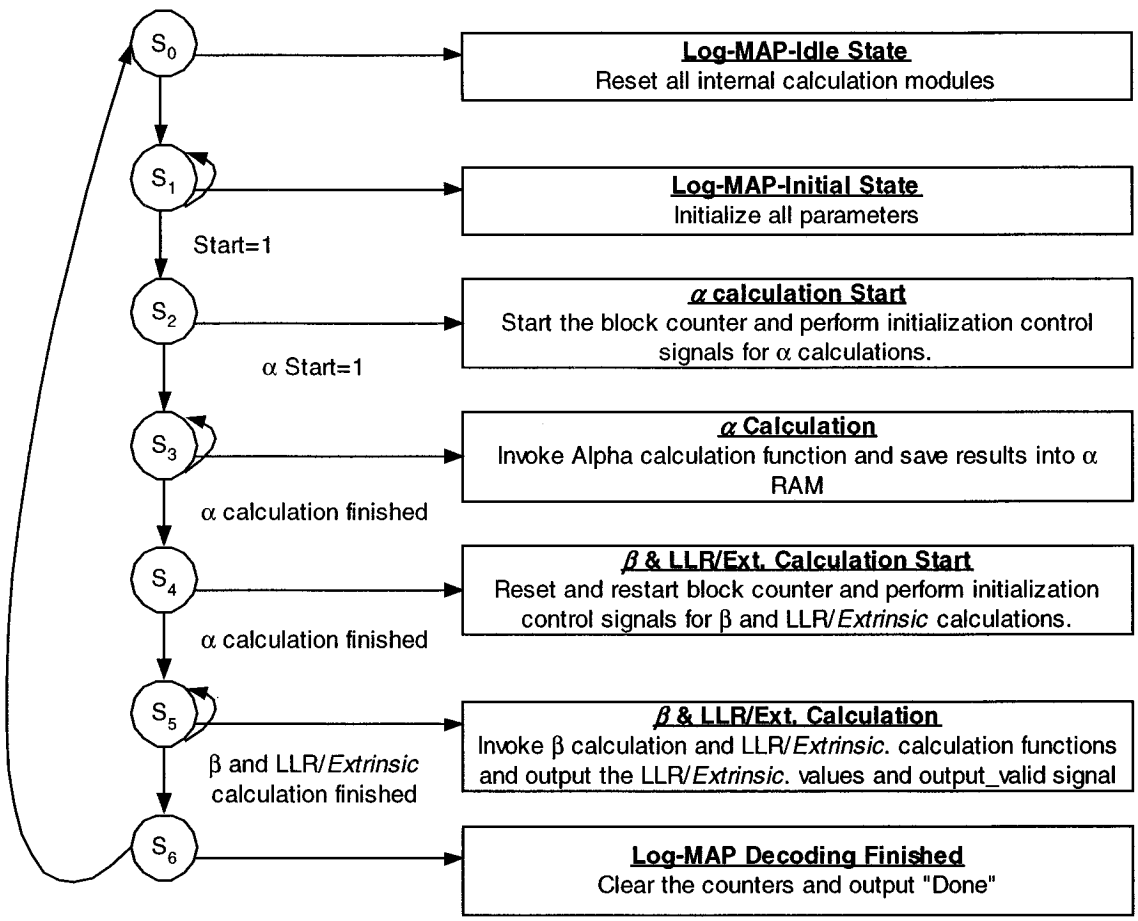


Figure 4-18 – State diagram of the Log-MAP decoder.

- "Log-MAP-Initial" – During this state, all the components are initialized to their initial state for the next decoding process. The α-calculation modules and *β*-

calculation modules are initialized according to (2.6) and (2.7), since our turbo decoder assumes that both CCs are terminated by bit flushing. The initial β values in the LLR calculation module are also initialized by (2.7). The first element of the α-RAM is initialized by one and all the other cells are initialized by zeros. The Log-MAP decoder will stay in this state until the "Log-MAP-Start" signal is detected.

- "α-Calculation-Start" – Upon the detection of the "Log-MAP-Start" signal is detected, the Log-MAP decoder starts the forward path metric calculation by asserting the "α-Start" signal and enters into "α-Calculation" state after one clock.
- "α-Calculation" – In this state, the Log-MAP decoder drives the α-calculation module to generated the α values. It first puts the correct input data addresses (derived from the block-counter) on their address bus of the external systematic, parity and *a priori* data RAMs. The branch metrics are then calculated and appears at the input ports of the α-calculation module. We assume that these operations are finished instantly since the branch-metric calculation module is a pure combinational logic circuit and introduces very small delay. At the same time, the Log-MAP control logic asserts "α-Calculation-Start" signal to start the α-calculation module. It then waits for the α-calculation module to output all the 2^M α values. Every time a α value is generated, the Log-MAP control logic writes it into the α-RAM, using the output of the block-counter and the state-counter as the address. When all 2^M α values are written into the RAM and a "α-Calculation Finished" signal is detected, it will reset the state index counter, increase the block counter by one, and starts another α-calculation process. When the value of the block-counter reaches the block length, the Log-MAP decoder stops providing "α-Calculation-Start" signal to α-calculation module, resets the block-counter and enters "β & LLR/Ext.-Calculation-Start" state.
- "β & LLR/Ext.-Calculation-Start" – Now that the forward path metrics are available, the Log-MAP decoder starts the backward path metric calculation and the LLR/*extrinsic* information calculation processes. The Log-MAP decoder starts

these two processes by asserting the "β-Calculation-Start" and "LLR-Calculation-Start" signals and enters into "α-Calculation" state after one clock.

- "β & LLR/Ext.-Calculation" – In this state, the Log-MAP decoder drives the β-calculation module to generated the β values and the LLR/ext. calculation module to calculate the *extrinsic* or LLR values. It first puts the correct input data addresses (derived from the block-counter) on their address bus of the external systematic, parity and *a priori* data RAMs. The branch metrics are then calculated and appears at the input ports of the β-calculation module. At the same time, the Log-MAP control logic asserts "β-Calculation-Start" signal to start the β-calculation module and LLR calculation module. The 2^M β values are stored into 16 12-bit registers to be used in next LLR calculation. The output *extrinsic* information is written into the *extrinsic* RAM. After the current calculations are finished, the β-calculation module and LLR-calculation module will assert "β-Calculation-Finished" and "LLR-Calculation-Finished" signals once the 2^M β values and one LLR/*extrinsic* value are stored (output) properly. Upon detection of this two signals, Log-MAP control logic will reset the state index counter, increase the block counter by one, and repeats the same operation to start the next β-calculation and LLR-calculation processes. When the value of the block-counter reaches the block length, the Log-MAP decoder stops the two calculation modules, resets the block-counter and enters "Log-MAP-Finished" state.
- "Log-MAP-Finished" – In this state, all the modules are reset and the output "Log-MAP-Done" signal is asserted to tell the turbo decoder that this Log-MAP decoding over the block is finished. Upon the next clock, the Log-MAP decoder enters "Log-MAP-Idle" state.
- At any time, the Log-MAP decoder can be returned into "Log-MAP-Initial State" by asserting the asynchronous "RESET" signal.

4.3.3 γ-Calculation Module

The γ-calculation calculates the path metric using the input systematic and parity symbols and the a priori information. The calculation follows (4.14). The input

systematic and parity symbols are already the scaled values by the SNR. Therefore, there mathematic operations only include fixed-point adding and inversion. This module is implemented as a pure combination logic circuit.

4.3.4 α/β/LLR Calculation Modules

Calculation of the forward path metrics, α, is performed according to (2.16). For Log-MAP algorithm, calculation of $\alpha(s_k)$ at time k is basically a recursive Add-Compare-Select-Offset (ACSO) operation over all possible transitions from the previous states to s_k. The function of an ACSO circuit can be expressed as,

$$r = \max\{a+b, c+d\} + f\left(\left|(a+b)-(c+d)\right|\right) \tag{4.29}$$

where $\{a, b, c, d\}$ are the four inputs, r is the single output, and $f(|x|)$ is the modification factor defined in section 2.2.1.3, which is implemented with a lookup-table (LUT). The calculation of α depends heavily on the trellis structure of the constituent code (CC). For a CC with generator polynomial (31, 27), the trellis structure is plotted in Figure 4-19.

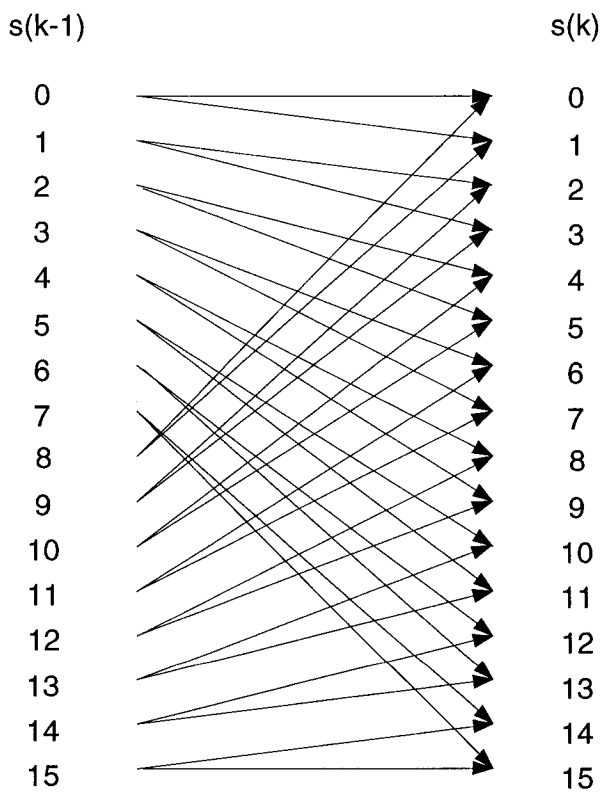


Figure 4-19 – Trellis diagram of the constituent code (CC).

With this trellis structure, the $\alpha(s_k)$ at time k is calculated as,

$$\begin{aligned}\alpha(s_k) = \max\left\{\alpha(s_{k-1}^0) + \gamma(s_{k-1}, s_k, d_k = 0), \alpha(s_{k-1}^1) + \gamma(s_{k-1}, s_k, d_k = 1)\right\} \\ + f\left(\left|\left(\alpha(s_{k-1}^0) + \gamma(s_{k-1}, s_k, d_k = 0)\right) - \left(\alpha(s_{k-1}^1) + \gamma(s_{k-1}, s_k, d_k = 1)\right)\right|\right)\end{aligned} \tag{4.30}$$

where s^i_{k-1} is the trellis state at time (k-1) that has a transition to state s_k at time k, caused by an input of "i", $i \in \{0, 1\}$. Therefore, $\alpha(s_k)$ is calculated by an ACSO operation over two previous α values and two branch transition metrics. The design objective becomes to efficiently find the four inputs of the ACSO in calculating each α. This can be implemented using two lookup tables (LUT), one for the transition caused by an input of "0" and the other for an input of "1". When the trellis structure is changed, the calculation operation can be changed correspondingly by updating the content of the lookup tables. The data path of the α-calculation module is plotted in Figure 4-20.

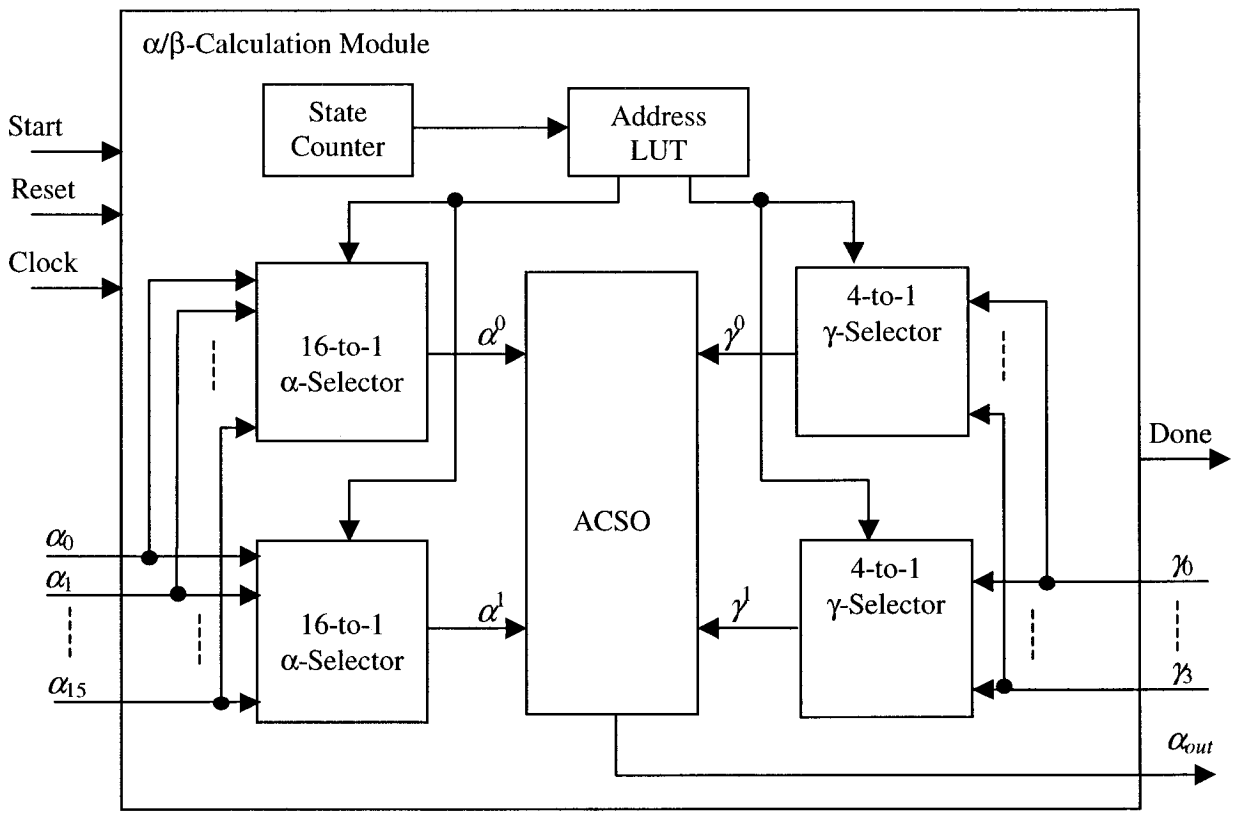


Figure 4-20 – α/β calculation module implementation.

The operation of the α-calculation module is as follows,

- The assertion of the "Start" signal tells the α-calculation module that the input α values and γ values are ready.
- When the "Start" is asserted, the state counter is enabled. The state counter output, i, $i \in \{0, 1, \ldots, 15\}$, is used to indicate which current-state α is being calculated. This value is used to lookup the selector addresses of the two last-state α values and the two γ values used in calculating the i^{th} current-state α value. The corresponding α and γ values then appear at the input to the ACSO, which calculates the new α value. These are all implemented as a combinational logic circuit.
- After all the α values are calculated and output, the α-calculation module output a "Done" signal to the Log-MAP decoder and wait for the next "Start" signal.

The β-calculation module has exactly the same data structure as the α-calculation module. Although we implemented separate α-calculation module and β-calculation module, they can be one physical device since their operations have very small overlap in time and they share very similar structure. A few modifications are needed to integrate both of their interfaces into one single device, such as providing the all the required LUTs and all the necessary interface signals.

The LLR/Ext. calculation module is slightly different from the path metric calculation modules. According to (2.20), the LLR (*extrinsic* information) is calculated by iteratively performing ACSO operations. Calculation of a LLR/Ext. value requires 2^M (16 in our example) α values, 2^M β values and all possible (4 in our case) γ values. Knowledge of the trellis structure is necessary to perform the computation, which is still implemented as table-lookup operations. Combining equation (2.18) and the modification factor proposed for Log-MAP algorithm, the LLR/Ext. is calculated as

$$\begin{aligned} L(d_k) \approx & \max_{(S_k, S_{k-1})}{}^{*}(\overline{\gamma}_1(y_k, S_{k-1}, S_k) + \overline{\alpha}_{k-1}(S_{k-1}) + \beta_k(\overline{S}_k)) \\ & - \max_{(S_k, S_{k-1})}{}^{*}(\overline{\gamma}_0(y_k, S_{k-1}, S_k) + \overline{\alpha}_{k-1}(S_{k-1}) + \beta_k(\overline{S}_k)). \end{aligned} \tag{4.31}$$

where max* stands for ACSO operation.

The computation of (4.31) is implemented as two parallel recursive ACSO operations, as shown in Figure 4-21. The upper branch is used to calculate the first term in (4.31) and the lower branch for the second term. Although the max operation is performed over all combinations of (s_k, s_{k-1}), we know that there are only two s_k values, s^0_k and s^1_k, that have transitions from any s_{k-1}. One of the two transitions between this last state to the two current states is caused by an input "0" and the other by an input "1". This means that each of them is used in either the upper branch or the lower branch shown in Figure 4-21.

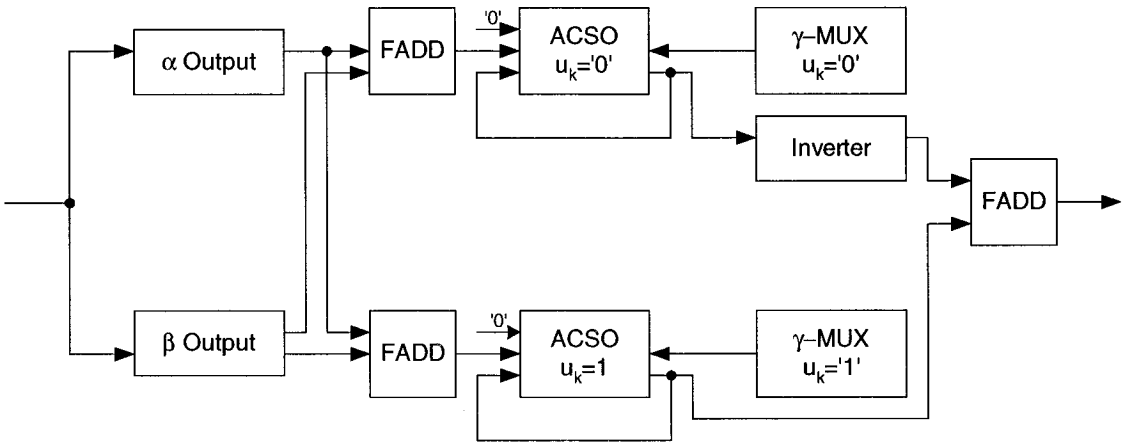


Figure 4-21 – LLR calculation module block diagram.

Therefore, the LLR/Ext. calculation is performed as follows,

- Initially, the feedback inputs of both ACSOs are all initialized as zeros.
- A "Start" signal is asserted by the Log-MAP control logic to indicate that all the input signals, including the α values, β values and γ values are ready at the input ports. A state counter is enabled.
- The output of the state counter is taken as the last trellis state, s_{k-1}. This value is used to select the α values in current calculation recursion. The two current states that have transitions to s_{k-1}, are looked up and used as the address to select the two β values to be used in the upper and lower branches. At the same time the

needed γ values are also looked up. With all these data, in each branch, an ACSO is used to generate a temporary result.

- This temporary result is feedback during the next recursion in the ACSO operation. After this operation is repeated over all possible last state values, the output at the two ACSOs are the results of the two max* operations.
- The LLR/*Extrinsic* is then calculated with these two results by a simple fixed-point subtraction.

4.3.5 Add-Compare-Select-Offset Unit

It has been shown that the ACSO is the basic building component of Log-MAP decoder, whose function is given in (4.29). The building block diagram of an ACSO is shown in Figure 4-22. The implementation consists of fixed-point adders, subtractors (implemented by an adder and an inverter), selector and a LUT. The subtractor is used to select the bigger operand. The absolute value of the subtraction result, Δ, is used as the address to the LUT, which outputs the modification factor, $f(|\Delta|)$.

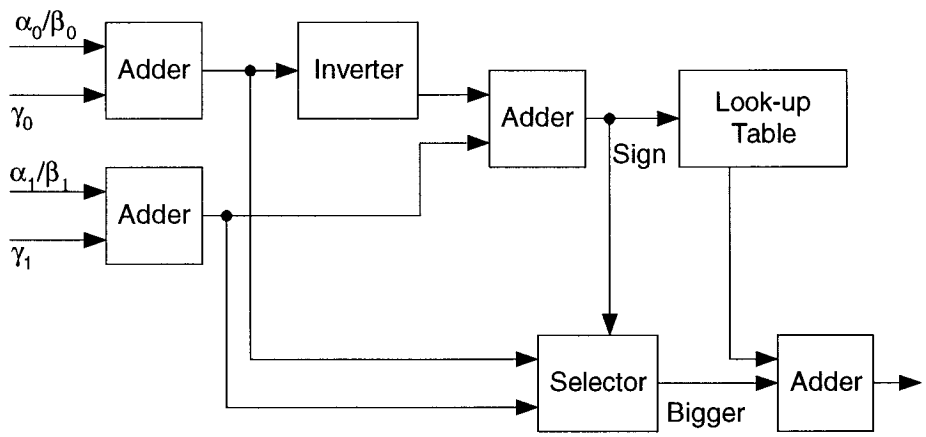


Figure 4-22 – Block diagram of Add-Compare-Select-Offset unit.

The correction term $f(|\Delta|) = \log(1+e^{-|\Delta|})$ is implemented by a pre-calculated look-up table. It was shown in [9] and also confirmed by our simulation results that performance degradation is negligible when using only 8 stored values with $|\Delta|$ ranging from 0~5.

Further simplifications can also be used so that the computation can be implemented without LUT with the tradeoff of minor performance degradation [9].

In this turbo decoder design, the LUT is implemented with a small RAM. The absolute offset, $|\Delta|$, is used as the address of their corresponding $f(|\Delta|)$ values. Since it was proved that the input can be limited to a maximum value of 5, whose FP(9,3) representation is [000000101.000], only the six least-significant bits (LSBs) are needed. Therefore, the LUT RAM needs an address bus of six bits. The output is limited from 0 to 0.69 and only the last three bits is required in FP(9,3) representation. Therefore, a RAM with 64 entries, each of 3-bit wide is required to implement the LUT. The LUT entries are shown in Table 4-3, where the "Address" is the six LSBs of $|\Delta|$ and "Data" will be the three LSBs of $f(\Delta)$. It is shown in this table that only 22 entries out of the 64 entries have non-zero values and there are only five different values.

Table 4-3 – LUT entries.

Address	Data	Address	Data	Address	Data	Address	Data
000000	110	010000	001	100000	000	110000	000
000001	101	010001	001	100001	000	110001	000
000010	101	010010	001	100010	000	110010	000
000011	100	010011	001	100011	000	110011	000
000100	100	010100	001	100100	000	110100	000
000101	011	010101	001	100101	000	110101	000
000110	011	010110	000	100110	000	110110	000
000111	011	010111	000	100111	000	110111	000
001000	011	011000	000	101000	000	111000	000
001001	010	011001	000	101001	000	111001	000
001010	010	011010	000	101010	000	111010	000
001011	010	011011	000	101011	000	111011	000
001100	010	011100	000	101100	000	111100	000
001101	001	011101	000	101101	000	111101	000
001110	001	011110	000	101110	000	111110	000
001111	001	011111	000	101111	000	111111	000

4.3.6 Timing Sequence of Log-MAP Decoding

The VHDL simulation of Log-MAP decoder timing sequence was performed and the input/output signal timing sequence is shown in Figure 4-23, where a clock signal of duration 40 *ns* is used.

In this simulation, a block length of 8 (including 4 tail bits) is used. The decoder, (including all the computation modules, α-calculation, β-calculation and γ-calculation) is initialized with a RESET signal during time 0 to 40*ns*.

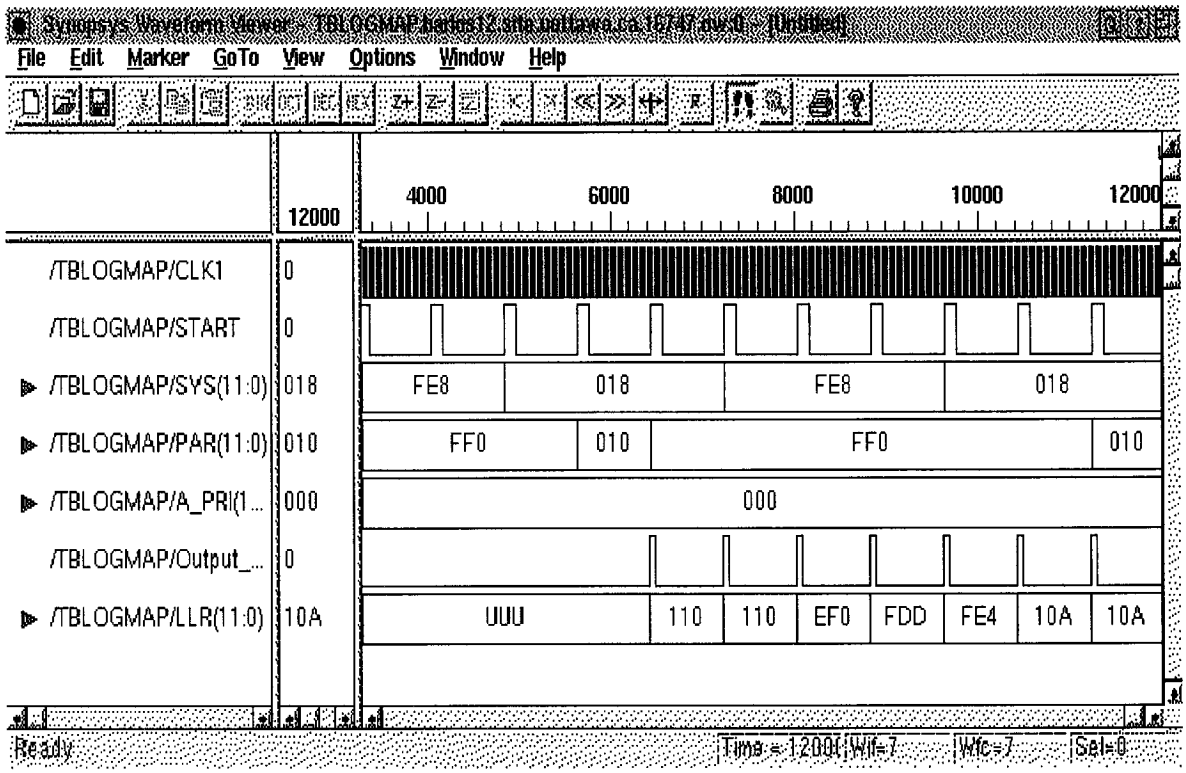


Figure 4-23 – VHDL Log-MAP decoder simulation timing sequence.

The entire decoding is divided into two periods. During the first period, the three input sequences are fed into the Log-MAP decoder and the forward path metrics are calculated and stored into α-RAM. With the last step α values calculated, the Log-MAP enters the second period, β and *extrinsic* information (LLR) calculation period. During this period, the three input sequences are again input in the reverse order (so as to avoid storing the γ values by re-calculation). The *extrinsic* information (LLR) is output during this period.

The START signal is periodically fed into the Log-MAP decoder starting from 40*ns*. The period is chosen to be 800*ns* (20 clock cycles), which is long enough for the internal

α-calculation or β-calculation to finish calculating all 2^M path metrics. Each assertion of the START signal tells the Log-MAP decoder that the systematic symbol, parity symbol and *a priori* symbol needed for calculating the next path metrics (and the *extrinsic* information) are valid on the three input ports.

During the first period (α-calculation period), after the α calculation is finished, the Log-MAP decoder will be waiting for next START signal to perform next α calculation. It doesn't give an "operation finish" signal. However, during the second period, whenever there is a valid *extrinsic* information (or LLR) is calculated and appears on the output port, an "Output-Valid" signal is asserted for one clock period to tell the Turbo decoder that the data at the output port is ready to be read. The total decoding process takes 20*15 clock cycles and it could be further reduced by carefully adjusting the control signals.

Note that the *extrinsic* information (or LLR) is output in inverse order. It is the responsibility of the turbo decoder to write these symbols into the right RAM locations.

4.3.7 Turbo Decoder Design Verification

The verification of the turbo decoder implementation is performed by comparing the decoding performance of the VHDL implementation with that of the C-simulation. The verification strategy is shown in Figure 4-24.

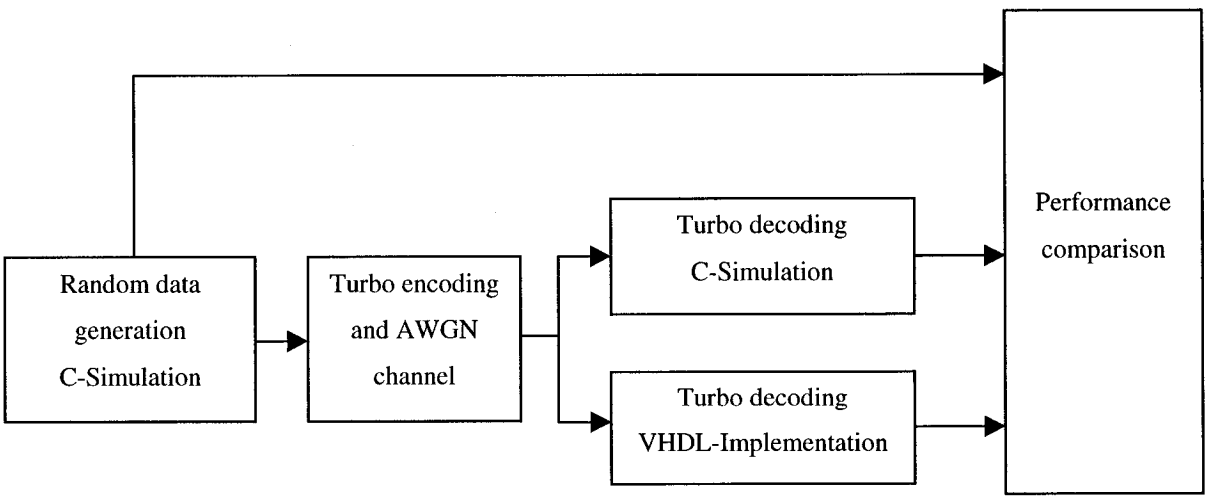


Figure 4-24 – Turbo decoder verification strategy.

To perform the design verification, C-simulation is first used to generate random data sequence to be encoded and transmitted. After a block of random data is generated, the turbo encoding is applied and the additive noise is added to the code word symbols. These are the received symbol sequence through an AWGN channel. The symbol sequence is first passed through a floating-point turbo decoder (C-simulation) and the results are saved in a disk file. To obtain a fair performance comparison, this floating-point turbo decoder takes into account of the dynamic range limit of the fixed-point turbo decoder implementation and employs the same path metric normalization method. The random data sequence and the channel-received data sequence scaled by SNR are stored into disk files to be used by the VHDL turbo decoding simulation.

A test bench of the VHDL turbo decoder implementation was developed in VHDL to test the decoding performance. The test bench reads a block of received data (already scaled by the SNR factor) and loads them into the systematic and parity symbol buffers. After all the buffers are ready, it sends a "START" signal to the turbo decoder and waits for the decoding outputs. When the turbo decoder finishes the decoding on this block, the test bench saves these results into an array and compares them to the original random data sequence (correct results) to obtain the error performance.

In the last step, the error performance of the two turbo decoders (floating-point C-implementation and fixed-point VHDL-implementation) on the same set of simulation data (random data and random noise) are compared.

The error performances of the floating-point turbo decoder and fixed-point implemented in AWGN channels were obtained through simulation and are shown in Figure 4-25. It is shown that the numbers of errors are very close and therefore the VHDL implementation is verified.

It should be noted that the floating-point C-simulation of turbo decoding only takes into account of the dynamic range of the fixed-point implementation. The quantization (precision) effect of the finite word length is not considered, which may bring some performance degradation to the fixed-point implementation. This can explain the small deviation of the two performance curves in Figure 4-25.

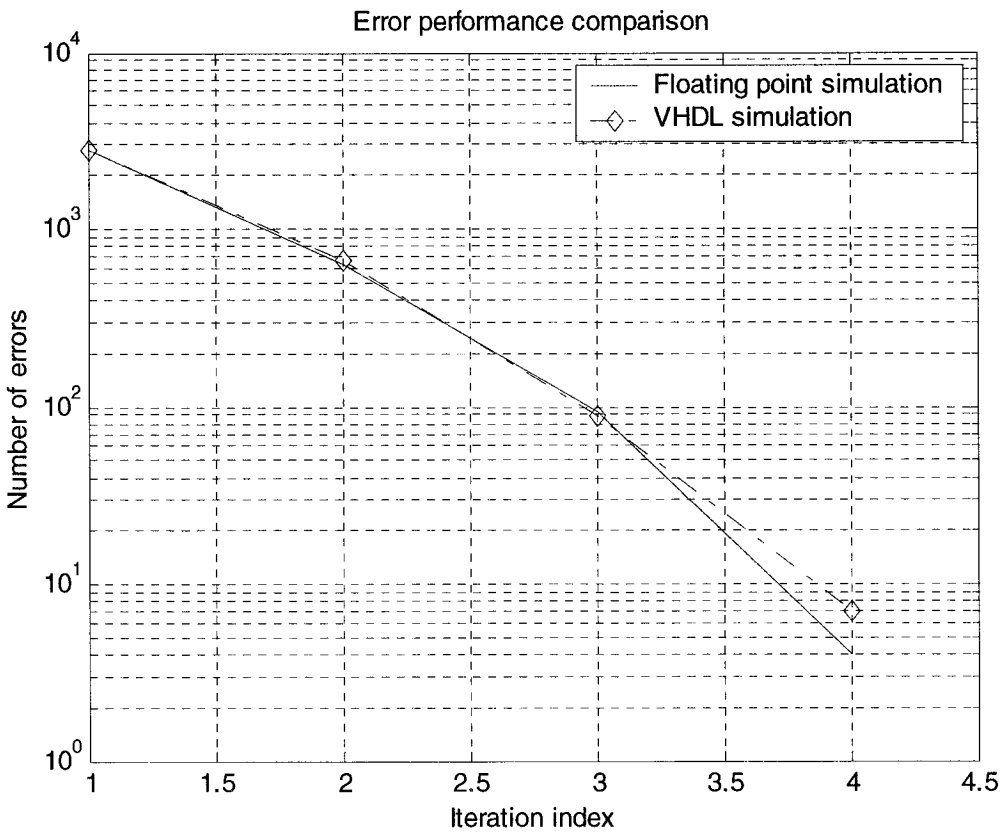


Figure 4-25 – Performance comparison of simulation and VHDL implementation.

4.4 Further Area Optimizations

The turbo decoder implementation in this thesis is an area-efficient design (in the sense of maximum hardware sharing) and performs block-wise operations. In this section, we present some implementation techniques to further reduce the area by reducing the memory requirement in the turbo decoder implementation. In addition, we discuss some other optimization options in turbo decoder implementation.

First of all, in our implementation, it is noted that the α-calculation module and β-calculation module have very similar structure and their operations have no overlap in time. Therefore, a common path metric calculation could be design include all their interface requirements. This will reduce one metric calculation module at almost no cost.

During our design, we mainly considered hardware sharing and didn't consider the memory requirement optimization for area reduction. However, it was shown in the

literature that memory could contribute significantly to the implementation area of the digital processing chipset. In the following, we discuss various memory optimization techniques that can be further applied to our design to achieve better area efficiency.

Architectural Experiments have shown that up to 50-80% of the area cost in (application-specific) architectures for real-time signal processing is due to memory units [34]. In our designed turbo decoder, two RAMs are used to store the forward recursion path metrics (α-RAM of size $2^M \cdot (K+4) \cdot N_{FP}$-bit fixed-point data) and the output *extrinsic* data sequence (L_{ex}-RAM of size K N_{FP}-bit fixed-point data). The total memory requirement is therefore,

$$C_M = K \cdot (2^M + 1) \cdot N_{FP} + 4 \cdot 2^M \cdot N_{FP} \tag{4.32}$$

To reduce the memory requirement, it is first noted that the *extrinsic* information is not necessarily presented using the same data width as the path metric data. It has been shown in [22] that a 6-bit representation is enough for *extrinsic* information without any noticeable decoding performance degradation. Therefore, in our implementation, the LLR-RAM could have 6-bit data width instead of 12-bit data width.

However, for the example turbo code with 16-state CCs, the major memory requirement comes from the α-RAM, which is 16 times larger than the LLR-RAM. In the following, we present several techniques to significantly reduce the path metric storage requirement.

4.4.1 Preprocessing over a Sliding Window

This technique is proposed in [35] by Viterbi. The operation of this technique is shown in Figure 4-26. The entire turbo block is divided into p blocks, each with L_p steps (trellis state transitions). One forward path metric recursion unit (F_1) and n backward path metric recursion units B_1–B_n are used.

This technique is based on the trellis convergence property. It has been shown in [34], [35] and [36] that a backward recursion starting with unknown initial state would generate very accurate results after certain convergence period. The structure shown in Figure 4-26 is assuming that the backward recursion B_1, staring with unknown state at nL_p, will generate accurate backward path metrics from L_p to 0, where the backward recursion during nL_p to L_p is the convergence period. The backward path metrics

generated by B_1 during L_p to 0 can therefore be used to calculate the *extrinsic* information (or LLR). The operation of this technique is as follows,

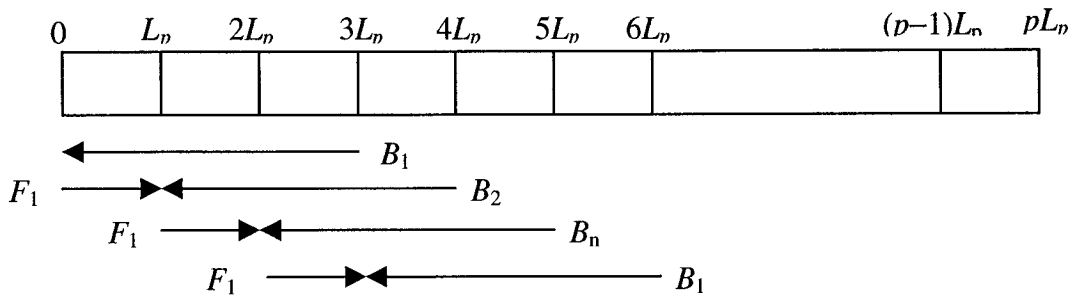


Figure 4-26 – Preprocessing over sliding window with n=3

- B_i (i=1 to n) starts backward recursion from trellis index $(n+i-1)\cdot L_p$.
- When B_1 finished the length-3L recursion, F_1 starts the forward recursion from 0. The LLR calculation is performed along with forward path metric calculation with the backward path metrics calculated by B_1 during its recursion from L_p to 0. At the same time, B_1 starts another backward recursion from $2n$ L_p.
- When F_1 and LLR calculation module finishes the calculation over the first block (0 – L_p), B_2 has finished the backward path metric calculation over the second block. Therefore, $F1$ and LLR calculation modules could continuously work on the second block. B_2 would start another backward recursion from $(2n+1)\cdot L_p$.
- This operation is repeated until the end of the entire turbo block.

The totally amount of memory required with this technique is now,

$$C_M = (L_p \cdot 2^M + K) \cdot N_{FP} \tag{4.33}$$

The length of the convergence period is determined by the convolutional encoder. Usually, a length of 5 or 6 times of the constraint length is enough to provide accurate convergence, which is usually much shorter than the total block length of the turbo code. Therefore, the memory requirement could be significantly reduced.

However, the memory reduction is achieved at the price of using more backward path metric calculation module. Assuming a convergence length of L_{con}, in the case of using n backward calculation modules, the sub-block size L_p is calculated as,

$$L_p = \frac{L_{con}}{n-1} \tag{4.34}$$

It is observed that more memory reduction involves larger number of backward calculation units, which requires more implementation area. Therefore, there is an optimum compromise to achieve the smallest implementation area.

The implementation area in this method could be reduced further if all the backward calculation recursions share a single physical backward calculation module. This, however, introduces large decoding delay.

4.4.2 Halfway Solution

This technique is also proposed in [35] by Viterbi. The operation of this technique is shown in Figure 4-27.

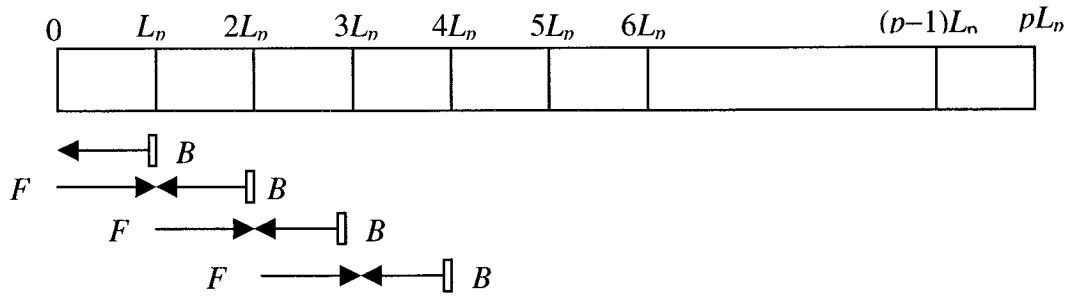


Figure 4-27 – Halfway solution

In this solution, the overall block is still divided into sub-locks of length L_p. There are only one forward path metric calculation module (F) and one backward path metric calculation module (B). This technique is based on the fact that the backward recursion can be carried out with initial values generated during the last decoding iteration. It has been shown that this introduces negligible performance degradation when applied to

Max-Log-MAP decoding algorithm. It can be applied to Log-MAP algorithm in a very straightforward way.

The operation is as follows:

- During the first turbo decoding iteration, the sub-block back recursions are initialized in a uniform and arbitrary way.
- B starts calculation from trellis index L_p, with the initial values generated from last decoding iteration.
- When B finishes the backward recursion over this subblock, F starts the forward recursion, while the *extrinsic* information (or LLR) is calculated along with the forward recursion. At the same time, B starts the backward recursion from 2 L_p, with the initial values generated from last decoding iteration.
- When B finishes the backward recursion from $(i+1)\cdot L_p$ to $i\cdot L_p$, the resulted backward path metrics at time $i\cdot L_p$ are stored in a RAM to be used during the next decoding iteration.

The memory requirement with this technique is,

$$C_M = \left[\left(p + L_p - 1\right)\cdot 2^M + K\right]\cdot N_{FP} \tag{4.35}$$

4.4.3 Other techniques

Extension of the memory reduction techniques introduced above were proposed in [37] and another efficient techniques was proposed in [23] for parallel window processing MAP decoding algorithm. Depends on the application, these are all options to perform area optimization. Interest readers can read these references for the detailed descriptions of these techniques.

4.5 Conclusions

In this chapter, we address various issues encountered in the turbo decoder implementations and present the VHDL design of both turbo encoder and turbo decoder.

We first give a flexible turbo encoder implementation with adjustable generator polynomials, i.e., the CCs of the turbo encoder can be programmed to have various polynomials with up to 16 states, and self-termination mechanism. The data path and

state diagram of the encoder are presented and explained. The design is then verified with the input-output timing sequence using a 16-bit input sequence and a reverse-order interleaver as an example.

Following the design of the turbo encoder, a turbo decoder based on Log-MAP decoding was designed and implemented. Different from turbo encoder design, which only involves binary XOR operations and simple logic operations, a turbo decoder is very complicated and requires large amount of mathematical computations.

Before the implementation of the turbo decoder, we first discussed some critical issues about VLSI implementation of turbo decoder.

- First of all, fixed-point data type is shown to be more efficient in the sense of implementation area, power consumption, operation speed and cost.
- Theoretical analysis of the data width requirement of the fixed-point data representation in the turbo decoder was performed based on the dynamic range and precision requirements of the internal data. It was shown that the dynamic range is mainly decided by the forward and backward path metrics accumulation. Although the path metrics are unbounded, the dynamic range of the path metrics is very well bounded. It is further shown by simulation that the dynamic range of the path metrics increases with the number of iterations. Later iteration tends to have larger dynamic range. However, after six iterations, the dynamic tends to converge to an upper bound. The conclusion of the analysis is that for out turbo code with 16-state CC, 12-bit fixed-point representation is more than enough to provide a performance very close to floating point simulation results, provided a proper normalization operation is performed during the path metric accumulations.
- Several path metric normalization methods are discussed. A modified normalization method is proposed to reduce the additional hardware and decoding latency required by the normalization operation.
- Effect of turbo decoder front end ADC quantization level and the channel SNR estimation to the turbo decoding performance are also briefly discussed.

Taking into account all the above discussions, an area-efficient turbo decoder based on Log-MAP SISO algorithm is designed and implemented with VHDL language.

- The VHDL implementation of the turbo decoder is presented following a top-down design procedure.
- The designed turbo decoder is shown to contain one Log-MAP decoder module, which is used alternatively as two SISO decoders with respect to the two CCs in the encoder. In this way, hardware is minimized at the price of decoding delay. The operation of the turbo decoder is explained with its "data path diagram" and "control path state diagram".
- The Log-MAP implementation is then presented with corresponding data path diagram and control logic state diagram. It mainly consists of the branch metric calculation module, the forward path metric calculation module and the backward path metric calculation module.
- The implementation logic descriptions of α-calculation module, β-calculation module and γ-calculation module are then presented. The core unit of all these modules is the ACSO unit, which was presented subsequently.
- An operation timing-diagram is obtained by simulating the Log-MAP decoder with an 8-bit block (including tailing bits). The diagram clearly shows the Log-MAP decoder input/output signal timing sequence, which confirms that the VHDL implementation performs the desired operation.
- Finally, the VHDL turbo decoder implementation is verified with simulation over noisy received channel symbols, with an interleaver size of 1020 bits. A corresponding simulation program is developed to generate the required data files (correct fixed-point data, noisy systematic and parity data etc.). Furthermore, floating-point simulation program is modified with the dynamic range limitation given by the 12-bit fixed-point data width and the simulation results are used to verify the performance of the VHDL implementation. Simulation results show very little performance difference between the fixed-point VHDL implementation and the floating-point simulation with the dynamic range constraint.

In designing the turbo decoder, we tried to minimize the hardware area by sharing components as much as possible. However, memory is another major component that might induce large implementation area. Therefore, at the end of this chapter, we presented several memory optimization techniques that can be combined with our turbo

decoder design to further minimize the overall implementation area. Generally speaking, reducing memory requirement means using some more hardware components. Therefore, whether reducing the memory requirement results in smaller area depends on if the increased hardware will balance the area saving from the memory reduction. There is usually a compromise to be determined.

Chapter 5

Conclusions

5.1 Summary

In this thesis, we first introduced the fundamental turbo coding concepts and the major components of a turbo coded systems: the turbo encoder, interleaver and the iterative turbo decoding structure in Chapter 2. The four popular SISO decoding algorithms were introduced briefly. Furthermore, the implementation complexities of these four SISO decoders were analyzed and compared in this chapter. It was observed that Log-MAP is no more than twice as complex as SOVA, where complexity of Max-Log-MAP is in between them. The actual implementation of Log-MAP will not be really twice as complexity as that of SOVA since it is more easy to perform parallel processing in Log-MAP.

In Chapter 3, turbo decoding with various SISO algorithms were simulated and analyzed to provide a comparative evaluation of their BER performance versus their computational complexity. The simulation results were used to evaluate the turbo codes performance using MAP, Max-Log-MAP, Log-MAP and SOVA decoding algorithms. It was observed that turbo coding using MAP and Log-MAP provide up to 8.6 dB coding gain for a BPSK transmission in AWGN channel at a BER of 10^{-5} compared with uncoded system. Turbo coding using Max-Log-MAP and SOVA provides about 8.0 dB coding gain at a BER of 10^{-5}.

The complexities of the four SISO decoding algorithms were compared using the execution time on the same simulation platform as the measure in this section. The results clearly indicated that the MAP algorithm is much more complex than the other three. SOVA has the minimum implementation complexity.

Based on the performance comparison and complexity comparison, the Log-MAP algorithm was chosen to be used in our turbo decoder implementation due to its good performance at low signal-to-noise ratio compared to Max-Log-MAP and SOVA algorithms, and its very reasonable implementation complexity.

In Chapter 4, we first present a VHDL implementation of turbo encoder, which involves mainly binary operations. It is shown that very simple circuit is needed for turbo code encoder. The dominant complexity of turbo codec is in the turbo code decoder. The turbo encoder is verified with a 16-bit input block by comparing the operation timing sequence, output sequence and the intermediate state transitions of the VHDL implementation with theoretical values.

Next, we addressed and analyzed the major design issues encountered in the hardware implementation of a turbo decoder, including data type, data width and path metric normalization method. It was concluded that fixed-point implementation is more practical than floating-point implementation. The data width is decided mainly by the dynamic range and precision requirement of the algorithm to obtain negligible performance degradation while keeping the minimum number of bits per data to minimize the implementation routing area. For the 16-state convolutional code, it was observed that the 9 bits are enough to cover the dynamic range of the path metrics and final LLR values. Research work in the literature showed that 3-bit precision is enough to provide negligible performance degradation compared to a floating-point turbo decoding. Therefore, with a fixed-point implementation, a 12-bit fixed-point format FP(9, 3) is more than enough to provide good performance. This format was used in the later turbo decoder implementation. A new normalization method was proposed based on variable shift method to reduce either the addition hardware or the additional delay required by the path metric normalization operations.

With the conclusions obtained from these discussions, we implemented an area-efficient turbo decoder that employs the Log-MAP decoding algorithm. Implementation

of the turbo decoder and its major components were described in detail using data-path diagrams and control path flow charts. The VHDL implementation of the turbo decoder operation was first verified using its timing diagram. Then the turbo decoder function was verified with a developed C-VHDL test bench program. Function of the turbo decoder fixed-point VHDL implementation was verified by comparing VHDL simulation performance to that obtained with floating-point C-program implementation. The C-simulation for VHDL implementation verification was performed with the same data dynamic range limitation and the same path metric normalization method. Finally, some memory reduction techniques proposed in the literature were introduced at the end of this chapter that can be added to our design to further perform the area-optimization.

5.2 Thesis Contribution

This thesis presents a practical hardware implementation of turbo codec with Log-MAP decoding algorithm. By simulating and analyzing the MAP, Max-Log-MAP, Log-MAP and SOVA decoding algorithms, the Log-MAP algorithm is selected to be the best compromise for a practical turbo codec implementation. The design is performed and simulated using VHDL, and is verified by comparing the VHDL simulation results to C-program simulation results.

- A brief description was provided on various SISO decoding algorithms. Implementation complexity analysis was performed on the three low-complexity SISO decoding algorithms, Max-Log-MAP, Log-MAP and SOVA.
- The C language simulations of MAP, Max-Log-MAP, Log-MAP and SOVA decoding algorithms in a BPSK digital communication system are performed to validate them with published literature research results. The simulation results were used to compare the performance of turbo decoding using these SISO decoding algorithms.
- Complexities of the SISO decoding algorithms were compared using the execution time on the same simulation platform as the measure. The results showed a good indication of the relative complexities of the four SISO decoders, although no implementation customization could be considered.

- A turbo encoder was designed and implemented in VHDL. Data path and control path state diagram were presented. The design was verified with the simulation timing diagram.
- Some of the well-know critical design issues in turbo decoder implementation were investigated, including the data word length, the quantization level requirement, the path metric normalization methods, etc.. The data width requirement was decided from both mathematical analysis and literature review. A modified path metric normalization method was proposed to minimize the additional hardware complexity and additional delay in the Log-MAP decoder.
- An area-efficient turbo decoder was implemented using VHDL as the design entry and design simulation tool. The design emphasizes on the portability and flexibility of the turbo codec. One advantage of our design is that the turbo codec can be adapted to meet different requirements easily.
- A combined C-VHDL test bench is developed to verify the performance of the VHDL turbo decoder implementation. The C-section of the test bench is used to generate random data block, perform turbo encoding, pass the turbo code word through AWGN channel, perform SNR scaling and do 32-bit floating-point to 12-bit fixed point data conversion. The original data sequence and the noisy turbo code word are stored in disk files to be read by the VHDL test bench. The VHDL test bench reads the noisy data generated by C-code and uses the data as stimulus to the turbo decoder implementation module. The outputs from the VHDL turbo decoder module are then compared to the original data sequence save by the C-program to decide the error performance. Very little performance difference was found between the 12-bit fixed-point VHDL implementation and floating-point C implementation.
- Several memory optimization methods were briefly discussed that can be applied to our turbo decoding implementation to further reduce the implementation area.

5.3 Suggestions for Future Research Work

This section presents the work that could be done in the future to improve the turbo code implementation and performance.

- In current design, the internal data width is represented with 12-bit data which seems more sufficient than necessary. We could find out the best tradeoff between the complexity (data width) and the performance degradation.
- Combine the hardware sharing and memory optimization techniques to find the most area efficient design.
- Our implementation is not targeting at the power efficiency and delay minimization, power and delay optimization techniques could be applied to minimize the turbo codec power consumption and reduce the overall delay of the turbo decoder.
- More simulations could be performed for validating the hardware design technologies that could improve the turbo decode performance or be used on complexity reduction.

Appendix A

In this section, we describe the technique used in the simulations to generate the random sequence input for the turbo encoder and analyze the reliability of the simulation results.

A.1 SISO Decoding Algorithms Complexity Analysis

In this section, we present the detailed complexity analysis procedure, from which the results in Table 2-1, Table 2-2 and Table 2-3 are derived.

Following the algorithm description in section 2.2.1.2, for Max-Log-MAP algorithm, we have,

- γ-calculation
  - $\gamma = y'_s \cdot x_s + y'_p \cdot x_p + p\{d_k\}$
  - There are totally 4 possible γ values for 4 combinations of $\{x_s, x_p\}$
  - Two inversion operations are needed
  - Each γ calculation takes 2 *adds*
  - **Totally: 8 *adds and 2 inversions***
- α-calculation
  - $\alpha(s_k) = \max\{\alpha_1(s_k) + \gamma_1, \alpha_2(s_k) + \gamma_2\}$, s_k=0 – 15
  - Each α takes 2 *adds* and 1 *max*
  - **Totally: $2 \cdot 2^M$ adds and 2^M *maxs***
- β-calculation : exactly same as α-calculation
  - **Totally: $2 \cdot 2^M$ adds and 2^M *maxs***
- LLR-calculation

- $L(d_k) = \max_{d=1}\{\alpha_1(s_k) + \beta_1(s_{k+1}) + \gamma_1, \ldots, \alpha_{16}(s_k) + \beta_{16}(s_{k+1}) + \gamma_{16}\} - \max_{d=0}\{\alpha_1'(s_k) + \beta_1'(s_{k+1}) + \gamma_1', \ldots, \alpha_{16}'(s_k) + \beta_{16}'(s_{k+1}) + \gamma_{16}'\}$
- Each input in the 2^M-input *max* takes 2 *adds*.
- Each 2^M-input *max* is implemented by (2^M-1) 2-input *max*
- Subtraction takes 1 *add* and 1 *inversion*
- **Totally: $2 \cdot 2^M \cdot 2 + 1$ *adds*, $2\cdot(2^M-1)$ *maxs* and 1 *inversion***

- The total computation complexity of the Max-Log-MAP is therefore,
  - **$8 \cdot 2^M + 9$ *adds***
  - **$4\cdot 2^M - 2$ *maxs***
  - **3 inversions**

The only difference between the Log-MAP from the Max-Log-MAP is adding the modification factor shown in section 2.2.1.3. The complexity is therefore calculated as,

- γ-calculation
  - $\gamma = y'_s \cdot x_s + y'_p \cdot x_p + p\{d_k\}$
  - There are totally 4 possible γ values for 4 combinations of $\{x_s, x_p\}$
  - 2 *inversion* operations are needed
  - Each γ calculation takes 2 *adds*
  - **Totally: 8 *adds and 2 inversions***
- α-calculation
  - $\alpha(s_k) = \max\{\alpha_1(s_k) + \gamma_1, \alpha_2(s_k) + \gamma_2\} + f(|\Delta|)$, s_k=0 - 15
  - Each α takes 3 *adds*, 1 *max* and 1 *table lookup*
  - **Totally: $3\cdot 2^M$ adds, 2^M *maxs and* 2^M *table lookups***
- β-calculation : exactly same as α-calculation
  - **Totally: $3\cdot 2^M$ adds, 2^M *maxs and* 2^M *table lookups***
- LLR-calculation
  - $LLR(d_k) = \max^*_{d=1}\{\alpha_1(s_k) + \beta_1(s_{k+1}) + \gamma_1, \ldots, \alpha_{16}(s_k) + \beta_{16}(s_{k+1}) + \gamma_{16}\} - \max^*_{d=0}\{\alpha_1'(s_k) + \beta_1'(s_{k+1}) + \gamma_1', \ldots, \alpha_{16}'(s_k) + \beta_{16}'(s_{k+1}) + \gamma_{16}'\}$
  - $\max^* = \max + f(|\Delta|)$
  - Each 2^M-input *max** is implemented by (2^M-1) 2-input *max**

- The first max* is implemented as max*{$\alpha_1(s_k) + \beta_1(s_{k+1}) + \gamma_1$, $\alpha_2(s_k) + \beta_2(s_{k+1}) + \gamma_2$}, which takes: 5 *adds*, 1 *max* and 1 *table lookup*
- The following (2^M–2) 2-input max* are implemented as max*{x, $\alpha_2(s_k) + \beta_2(s_{k+1}) + \gamma_2$}, which takes: 3 *adds*, 1 *max* and 1 *table lookup*
- Subtraction takes 1 *add* and 1 *inversion*
- **Totally: $2\cdot(2\cdot 2^M + 2^M - 1) + 1$ *adds*, $2\cdot(2^M - 1)$ *maxs*, $2\cdot(2^M - 1)$ *table lookups*, 1 *inversion***

- The total computation complexity of the Log-MAP is therefore
  - **$12\cdot 2^M + 7$ *adds***
  - **$4\cdot 2^M - 2$ *maxs***
  - **$4\cdot 2^M - 2$ *table lookups***
  - **3 *inversions***

The computation complexity of SOVA can be analyzed following the algorithm introduced in [11] as,

- Branch Metric calculation
  - Same as γ-calculation in Max-Log-MAP algorithm
  - **Totally: 8 *adds and* 2 *inversions***
- Selecting the ML path – the first Viterbi algorithm
  - $PM(s_k) = \max\{PM_0 + \gamma_0, PM_1 + \gamma_1\}$
  - For each PM, 2 *adds* and 1 *max*
  - The *max* operation also results in $\Delta = |PM_0 - PM_1|$
  - **Total: $2\cdot 2^M$ adds and 2^M *maxs***
- Finding the survivors and concurrent path for each state at each time instance –the second Viterbi algorithm
  - **Total: $2\cdot 2^M$ adds and 2^M *maxs***
- Reliability updating by back-tracing the two paths from the state on the ML path
  - Assuming double trace-back for $6\cdot(M + 1)$ branches, so $6\cdot(M + 1)$ *bit comparisons* are necessary.

- For each different bit decision on these two paths, *LLR* updating is performed by $LLR_{new}=\min\{LLR_{old}, \Delta/C\}$, where C is a constant and Δ/C is simply implemented by binary shifting.
- In the worst case, all the decisions are different along the two paths, so we have to performed LLR updating on $6\cdot(M+1)$ bits, which means: $6\cdot(M+1)$ *mins*
- Assuming one *bit comparison* is the same as one *add* operation.
- Assuming one *min* is the same as one *max*.
- **Totally: $(6\cdot M + 6)$ *adds* and $(6\cdot M + 6)$ *maxs***

- The total computation complexity of the SOVA is therefore
  - **$4\cdot 2^M + 6\cdot M + 14$ *adds***
  - **$2\cdot 2^M + 6\cdot M + 6$ *maxs***
  - **2 *inversions***

The analysis presented in this section are mainly based on the mathematical algorithms described in section 2.2. Different implementations of these algorithms can easily result in different computational complexity results. However, since these complexities are represented in the number of basic mathematical operations and the algorithms are based on the most up-to-date implementations, the complexity comparison based on these numbers is very accurate.

A.2 Generate Random Sequence with Normal (Gaussian) Distribution

We use the Box-Muller method to generate random deviates with a normal (Gaussian) distribution [17],

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{6.1}$$

Consider the transformation between two uniform deviates on (0,1), x_1 and x_2, and two quantities y_1 and y_2,

$$\begin{aligned} y_1 &= \sqrt{-2\ln x_1}\cos 2\pi x_2 \\ y_2 &= \sqrt{-2\ln x_1}\sin 2\pi x_2 \end{aligned} \tag{6.2}$$

Equivalently we can write

$$x_1 = \exp[-\frac{1}{2}(y_1^2 + y_2^2)]$$
$$x_2 = \frac{1}{2\pi}\arctan\frac{y_2}{y_1} \tag{6.3}$$

Now the Jacobian determinant can readily be calculated

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2\}} = -[\frac{1}{\sqrt{2\pi}}e^{-y_1^2/2}][\frac{1}{\sqrt{2\pi}}e^{-y_2^2/2}] \tag{6.4}$$

Since this is the product of a function of y_2 and a function of y_1 respectively, y_1 and y_2 are independently distributed according to the property of normal distribution.

Suppose that, instead of picking uniform deviates x_1 and x_2 in the Unit Square, we pick up v_1 and v_2 as the ordinate and abscissa of a random point inside the unit circle around the origin. The sum of their squares, $R \equiv v_1^2 + v_2^2$ is then a uniform deviate, which can be used for x_1, while the angle that (v_1, v_2) defines with respect to the v_1 axis can serve as the random angle $2\pi x_2$. The cosine and sine in (6.2) can now be written as $v_1/\sqrt{R}$ *and* $v_2/\sqrt{R}$, obviating the trigonometric function calls.

A.3 Estimating the Error Probability of the system simulated

In a Monte Carlo simulation, when a total of N bits are tested, out of which n bits are observed to be in error, an unbiased estimator of the BER, p', is calculated as [17],

$$p' = \frac{n}{N} \tag{6.5}$$

In the limit $N \to \infty$, the estimate p' will converge to the true value, p. For the common case of finite N, the reliability of the estimation is quantified in terms of confidence intervals. The confidence interval of an estimation is often expressed by three values, h_1, h_2 and a confidence level $1-\alpha$, which meet the following relationship,

$$p[h_2 \le p \le h_1 \,|\, n, N] = 1-\alpha \tag{6.6}$$

Since Monte Carlo simulation is only a label for the implementation of a sequence of Bernoulli trials, for a certain value of N, the number of error events (so as p') is binomially distributed. It is shown in [17] that the binomial distribution converges to the

normal distribution as $N \to \infty$ if p *is a constant*, with a mean of p and variance of $\sigma_p^2 = p\cdot(1-p) / N$. With this normal approximation, the confidence interval is calculated as,

$$p\left\{\frac{N}{N+d_a^2}\left[p'+\frac{d_a^2}{2N}-d_a\sqrt{\sigma_p^2+\left(\frac{d_a}{2N}\right)^2}\right]\le p\le\frac{N}{N+d_a^2}\left[p'+\frac{d_a^2}{2N}+d_a\sqrt{\sigma_p^2+\left(\frac{d_a}{2N}\right)^2}\right]\right\}\le 1 \quad (6.7)$$

where the value of d_a is chosen as,

$$\frac{1}{\sqrt{2\pi}}\int_{-d_a}^{d_a} e^{-t^2/2}dt = 1-\alpha \quad (6.8)$$

For most cases of interest, the approximations $N/(N+d_a^2) \approx 1$ and $p'\cdot(1-p') \approx p'$ results in negligible difference. The equation above is therefore simplified as,

$$p\left[p'+\frac{d_a^2}{2N}-d_a\sqrt{\sigma_p^2+\left(\frac{d_a}{2N}\right)^2}\le p\le p'+\frac{d_a^2}{2N}+d_a\sqrt{\sigma_p^2+\left(\frac{d_a}{2N}\right)^2}\right]\le 1-\alpha \quad (6.9)$$

In the case of $p'=10^{-k}$ and $N=\eta\cdot 10^k$, with k not necessarily an integer, the confidence interval can be further expressed as,

$$p(h_2 \le p \le h_1) = 1-\alpha$$

with (6.10)

$$h_{\pm} = 10^{-k}\cdot\left[1+\frac{d_a^2}{2\eta}\left(1\pm\sqrt{\frac{4\eta}{d_a^2}+1}\right)\right]$$

In the case of turbo decoding, an error event is defined as a block error which contains multiple bit errors. The multiple bit errors contained in one error event are correlated and bit errors from different error event are independent of each other. The BER estimation shown in (6.5) is still an unbiased estimation and asymptotically normally distributed, with a larger variance in this case. This means that for correlated bit errors, more bit errors need to be observed to achieve the same confidence interval compared to the case of independent bit errors. The estimation of the variance depends on the specific characteristics of the correlations between the bit errors. In the simplest case,

where an error event contains a fixed number of (m+1) bit errors, the variance is calculated as,

$$\sigma_p^2 \approx \frac{1}{N} p(1-p)(1+2m) \tag{6.11}$$

Since the confidence interval is proportional to σ_p, it is then stretched by a factor of $\sqrt{1+2m}$.

A.4 Detailed Description of Turbo Encoder Implementation

In this section, we present the detailed implementation of the turbo encoder data path. The turbo encoder datapath block diagram is shown in Figure 4-4 in Chapter 4. The detailed information of each component and signal used in the turbo encoder data path are described below.

1) An RAM is used to store the input data
2) The *Write_enable* (*WE*) signal is enabled during the *data_storing_stage*, and disabled after all the data have been saved. The address signal is driven by the output of the ADDRESS_MUX. The data_out signal drives the two CONVOLUTIONAL_ENCODERs and the overall output of the turbo encoder.
3) A ROM is used the store the interleaving pattern
4) The ROM stored the pre-defined interleaving pattern. The i^{th} element of the ROM indicates the after-interleaved position of the i^{th} input data.
5) A COUNTER is used to provide the address to both the RAM and ROM
6) The COUNTER counts from *0* to *N-1*. The COUNTER provides the address to both the RAM for writing data into RAM in the *data_storing_stage* and for outputting systematic data from RAM and providing data to the first convolutional encoder to produce the first parity bit in the *normal_encoding_stage*. It also provides the address signal to the ROM, which in turn addresses the SRAM, for the data input to the second convolutional encoder in the *normal_encoding_stage*.
7) There are three control signals for the COUNTER, CLK, *counter_reset* and *counter_hold*.

8) The *counter_reset* reset the COUNTER to zero if it is '1'.

9) The *counter_hold* holds the value of the COUNTER. This is used in the *normal_encoding_stage* and the *termination_stage*. In *normal_encoding_stage*, each encoding cycle consists of 3 clocks. The counter remains the same for these 3 clocks. Therefore, the COUNTER adds one in the first clock and keeps value in the following two. In the termination stage, the COUNTER remains the same for four clocks each time. In this way, the COMPARATOR can provide valid conditional signals to the control path.

10) An ADDRESS_MUX is used to address the RAM for writing and outputting operation

11) The ADDRESS_MUX drives the address line of the RAM. The control is the *addr_sel.*

- *addr_sel is '0' during data_storing_stage*
- addr_sel *is '0' during the output period of the systematic bits and the first parity bits in the* normal_encoding_stage
- addr_sel *is '1' during the output period of the second parity bits in the* normal_encoding_stage
- *Don't care in the* tremination_stage

12) A COMPARATOR is used to provide the condition signals to the control path to indicate different stage of the encoding operation for state transition

13) In the simulation, three state-transition signals are provided by this COMPARATOR, to indicate the completion of the *data_storing_stage* (*State_change_0*), the *normal_encoding_stage* (*State_change*), and the *termination_stage* (*State_change_1*).

14) Because those three signals are generated in non-overlapping period, they can be simplified into only a single signal, *counter_finish.* With the control signal *counter_reset*, the three state transition signals can be replaced.

15) Two CONVOLUTIONAL_ENCODERs performs the convolutional encoding

16) The convolutional encoders are implemented according to . For each encoder, a shift register and some AND gates and XOR gates are required.

- sh_reset *signals sets the shift register to zero state*

- sh_hold *holds the convolutional encoder even if there is a clock signal and input*
- tail *signal indicates the working mode of the convolutional encoder*
- Data_in *is the input data for the convolutional encoder*
- *The systematic bit and parity bit are output from different output lines. The systematic output is mainly used in the* termination_stage.

17) An OUTPUT_MUX is used to provide the output of the turbo encoder

The OUTPUT_MUX guarantees the current output is the signal from the right component. In *normal_encoding_stage*,

- *out_sel0* = '1', *out_sel*1 = '0', *tail* = '0' *:outputting systematic bit*
- *out_sel*0 = '0', *out_sel*1 = '1', *tail* = '0' *: outputting the first parity bit*
- *out_sel*0 = '1', *out_sel*1 = '1', *tail* = '0' *:outputing the second parity bit*

During the terminating_stage

- *out_sel*0 = '0', *out_sel*1 = '0', *tail* = '1' *:outputting the first tailing bit*
- *out_sel*0 = '1', *out_sel*1 = '0', *tail* = '1' *:outputting the first parity bit*
- *out_sel*0 = '0', *out_sel*1 = '1', *tail* = '1' *:outputting the second tailing bit*
- *out_sel*0 = '1', *out_sel*1 = '1', *tail* = '1' *:outputting the second parity bit*

Reference

[1] C. E. Shannon, "The Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, pp. 379-423, July 1948, and pp. 623-656, Oct., 1948.

[2] S. G. Wilson, "Digital Modulation and Coding," Prentice-Hall, Inc, 1996.

[3] C. Berrou and A. Glavieux, "Near Optimum Error Correcting Coding And Decoding: Turbo-Codes," *IEEE Trans. Commun.*, vol. 44, No. 10, pp. 1261-1271, Oct. 1996 .

[4] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes," *Proc. of Int. Conf. Commun.*, pp. 1064-1070, 1993.

[5] S. Benedetto, G. Montorsi, "Unveiling Turbo Codes: Some Results on Parallel Concatenated Coding Schemes," *IEEE Trans. Inform. Theory*, vol. 42, No. 2, pp.409-429, March 1996.

[6] S. Benedetto, R. Garello and G. Montorsi, "A Search for Good Convolutional Codes to be Used in the Construction of Turbo Codes," *IEEE Trans. Commun.*, Vol. 46, No. 9, pp. 1101-1105, Sept. 1998.

[7] C. Heegard, S. B. Wicker, "Turbo Coding," Kluwer Academic Publishers, Boston/Dordrecht /London, Jan. 1999.

[8] R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate," *IEEE Trans. Info. Theory*, pp. 284-287, Mar. 1974.

[9] P. Robertson, E. Villebrun and P. Hoeher, ``A Comparison of Optimal and Sub-optimal MAP Decoding Algorithms Operating in the Log Domain," *Proc. of Int. Conf. Commun.*, pp. 1009-1013, June 1995.

[10] J. Hagenauer, P. Hoeher, "A Viterbi Algorithm with Soft-Decision Outputs and its Applications," *Proc. of IEEE Globecom Conf.*, pp. 1680-1686, Nov. 1989.

[11] C. Berrou, P. Adde, E. Angui, and S. Faudeil, "Low Complexity Soft-Output Viterbi Decoder Architecture," *Proc. of Int. Conf. Commun.*, pp. 737-740, May 1993.

[12] M. P. C. Fossorier, F. Burkert, S. Lin, and J. Hagenauer, "On the Equivalence Between SOVA and Max-Log-MAP Decodings," *IEEE Commun. Letters*, vol. 2, No. 5, pp. 137-139, May 1998.

[13] W. E. Ryan, "A Turbo Code Tutorial", New Mexico State University, Box 30001 Dept. 3-O, Las Cruces, NM 88003.

[14] Small World Communications, "Iterative Decoding of Parallel Concatenated Convolutional Codes," Application note, version 1.4, 13 Jan. 1999.

[15] L. Lin and R. S. Cheng, "Improvements in SOVA-Based Decoding for Turbo Codes," *Proc. of Int. Conf. Commun*, pp. 1473-1478, June 1997.

[16] S. Hong, W. E. Stark, "Design and Implementation of a Low Complexity VLSI Turbo-Code Decoder Architecture for Low Energy Mobile Wireless Communications," *Journal of VLSI Signal Processing Systems*, pp. 2350-2354, 2000.

[17] M, C. Jeruchim, P, Balaban, and K. S, Shanmugan, "Simulation of Communication Systems," Plenum Press, New York and London, 1992.

[18] D. Pellerin, D. Taylor, "VHDL Made Easy!," Prentice Hall, NJ 07458, 1997.

[19] S. Crozier, P. Guinand, J. Lodge, and A. Hunt, "Construction and Performance of New Tail-Biting Turbo Codes," *Proc. of 6th International Workshop on Digital Signal Processing Techniques for Space Applications*, paper 1.3, Spet. 1998.

[20] S. Crozier, J. Lodge, P. Guinand, and A. Hunt, "Performance of Turbo-Codes with Relative Prime and Golden Interleaving Strategies," *Proc. of 6th International Mobile Satellite Conference (IMSC'99)*, pp. 268-275, June 1999.

[21] G. Montorsi and S. Benedetto, "Design of Fixed-Point Iterative Decoders for Concatenated Codes with Interleavers," *IEEE Journal on Selected Areas in Communications*, vol. 19, No. 5, pp. 571-582, May 2001.

[22] Y. Wu, B. D. Woerner, and T. K. Blankenship, "Data Width Requirements in SISO Decoding With Modulo Normalization," IEEE Trans. On Commun., vol. 49, No. 11, Nov. 2001.

[23] A. Worm, H. Lamm and N. Wehn, "VLSI Architectures for High-Speed MAP Decoders", *Proc. of 14th International Conference on VLSI Design*, pp. 446-453, Jan. 2001.

[24] H. Michel, A. Worm and N. Wehn, "Influence of Quantization on the Bit-Error Performance of Turbo-Decoders," *Proc. of Vehicular Technology Conf.*, pp.581-585, 2002.

[25] C. B. Shung, P. H. Siegel, G. Ungerboeck, H. K. Thapar, "VLSI Architectures for Metric Normalization in the Viterbi Algorithm", *Proc. of Int. Conf. Commun.*, vol. 4, pp. 1723-1728, April, 1990.

[26] A. P. Hekstra, "An Alternative to Metric Rescaling in Viterbi Decoders," *IEEE Trans. Commun.*, vol 37, No. 11, pp. 1220-1222, Nov. 1989.

[27] G. Jeong and D. Hsia, "Optimal Quantization for Soft-Decision Turbo Decoder," *Proc. of Vehicular Technology Conf.*, vol. 3, pp. 1620-1624, 1999.

[28] A. Wiesler, O. Muller and F. Jondral, "MAP-Algorithm with Fixed-Point Representation for Software Radios," *Proc. of Vehicular Technology Conf.*, Vol. 6, pp. 2962-2968, 2000.

[29] Y. Wu, B. D. Woerner, "The Influence of Quantization and Fixed Point Arithmetic Upon The BER Performance of Turbo Codes," *Proc. of Vehicular Technology Conf.*, pp. 1683-1687, 1999.

[30] Z. Blazek and V. K. Bhargava, "A DSP-Based Implementation of A Turbo-Decoder," *Proc. of IEEE Globecom Conf.*, vol. 5, pp. 2751-2755, 1998.

[31] S. Kim. "Probabilistic Reasoning, Parameter Estimation, and Issues in Turbo Decoding," Ph.D. dissertation, Cornell University, 1998.

[32] S. Kim and S. B. Wicker, "On Mismatched and Self-Matching Turbo Decoding," Submitted to IEEE Trans. Inform Theory, 1998.

[33] T. A. Summers and S. G. Wilson, "SNR Mismatch and Online Estimation in Turbo Decoding," *IEEE Trans. Commun.*, vol. 46, No. 4, pp.421-423, April. 1998.

[34] A. Worm, H. Lamm and N. Wehn, "A High-Speed MAP Architecture with Optimized Memory Size and Power Consumption," *Proc. of IEEE Workshop on Signal Processing Systems* (SiPS), pp. 265-274, 2000.

[35] A. J. Viterbi, "An Intuitive Justification and Simplified Implementation of MAP Decoder for Convolutional Codes," *IEEE Journal on Selected Areas in Communications*, vol. 16, pp. 260-264, Feb. 1998.

[36] F. Raouafi, A. Dingninou and C. Berrou, "Saving Memory in Turbo-Decoders Using the Max-Log-MAP Algorithm," *IEE Colloquium on Turbo codes in digital broadcasting – could it double capacity ?*, pp. 14/1 – 14/4, 1999.

[37] E. Boutillon, W. J. Gross and G. Gulak, "VLSI Architectures for the Forward-Backward Algorithm," on-line reference, http://lester.univ-ubs.fr:8080/~boutillon/sanchezt/main.htm.

[38] H. David, G. Gehnen and H. Meyr, "MAP Channel Decoding: Algorithm and VLSI Architecture," *VLSI Signal Processing*, vol. VI, pp. 141-149, 1993.

[39] W.J. Gross, P.G.Gulak, "Simplified MAP algorithm suitable for implementation of turbo codes," *Electronics Letters*, vol. 34, pp. 1577-1578, 1998.

[40] G Masera., G. Piccinini, M. R. Roch and M. Zamboni, "VLSI Architectures for Turbo Codes," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 7, No. 3, pp. 369-379, Sept 1999.

[41] P. A. Beerel and K. M. Chugg., "A Low Latency SISO with Application to Broadband Turbo Decoding," *IEEE Journal on Selected Areas in Communications*, vol. 19, No. 5, pp. 860-870, May 2001.

[42] J. A. Heller and I. M. Jacobs, "Viterbi Decoding for Satellite and Space Communication," *IEEE Trans. Commun. Tech.*, vol. COM-19, No. 5, pp. 835-848, Oct. 1971.