

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

An FPGA Implementation of RM-BTC Codec using Log-MAP Algorithm

Qing Li

A Thesis
in
the Department
of
Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

July 2002

©Qing Li, 2002



**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72910-9

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

An FPGA Implementation of RM-BTC Codec using Log-MAP Algorithm

Qing Li

A Thesis

in

the Department

of

Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

July 2002

©Qing Li, 2002



**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72910-9

Canada

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Mr. Qing Li (4352475)

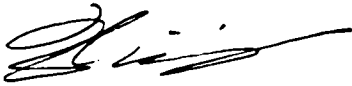
Entitled: "A FPGA Implementation of RM-BTC Codec Using Log-MAP Algorithm"

and submitted in partial fulfillment of the requirements for the degree of


Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

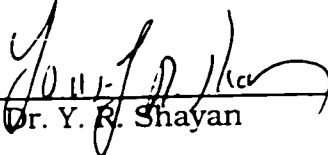
Signed by the final examining committee:



Dr. W-P. Zhu Chair



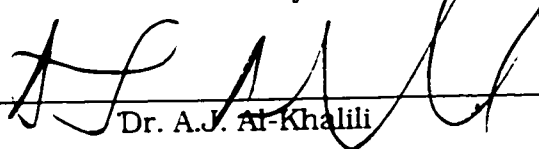
Dr. C. Rozon Examiner, External to the Program




Dr. Y. R. Shayan Examiner



Dr. M.R. Solymani Supervisor




Dr. A.J. Al-Khalili Supervisor

Approved by: 

Dr. M.O. Ahmad
Chair, Department of Electrical
and Computer Engineering

SEP 18 2002
20



Dr. Nabil Esmail
Dean, Faculty of Engineering
and Computer Science

ABSTRACT

An FPGA Implementation of RM-BTC Codec using Log-MAP Algorithm

Qing Li

Due to their powerful error correcting capability and superior coding gain, Turbo Codes are used in 3rd generation wireless and satellite communication systems. For these applications, efficient implementation of Turbo Codes, i. e., development of codec providing high throughput with small chip area and low power consumption is of growing importance.

In this thesis, Turbo Code using Reed-Muller code as its constitute code is implemented in VHDL and logic synthesis is executed. The Max-Log-MAP algorithm is used due to its significantly reduced complexity and negligible performance degradation from MAP algorithm. The implementation of codec mainly focuses on achieving the smaller chip area and lower power dissipation, and target to device Virtex-E FPGA. For this purpose, the system and module level optimization of codec architecture is carefully considered through the parallelism and pipeline, interleaving technique, function unit sharing and memory access. The quantization and finite accuracy are also discussed. The simulation in RTL level on a wide variety of test vectors is done, and results show that the encoder/decoder execute properly and correct functionality is realized. The synthesis reports show that chip utilization is reasonable and more resource remains for future improvement.

Acknowledgments

I am so fortunate to have the opportunity to work on this interesting topic under the supervision of Dr. Asim A. Al-Khalili and Dr. M. Reza Soleymani. I want to express my sincere gratitude to both supervisors and truly appreciate their constructive guidance, valuable advice and helpful discussion throughout my thesis research. Their kind relationship with student make me could not ask more from them. Their financial support also provides me a more freedom to pursue my academic goal.

I am grateful to Usa Vilaipornsawai for her sharing with me knowledge of RM code and to whom I owe much of understanding of this topic. I want to thank all peoples who help and encourage me throughout my study in concordia university for their valuable solicitude and help.

Finally, I would like to express my heartfelt gratitude to my wife for her unconditional love and support. Without her understanding, it is impossible for me to continue my study and finish this research.

Table of Contents

List of Figures	viii
List of Tables	ix
List of Acronyms	x
1. Introduction	1
1.1 Introduction	1
1.2 Review of Error Correction Codes	4
1.3 Objective of thesis	8
1.4 Contribution	8
1.5 Overview	9
2. Block turbo codes	11
2.1 Basics of block turbo codes	11
2.1.1 Linear block code	11
2.1.2 Block turbo codes	13
2.2 Reed-Muller turbo codes	15
2.2.1 RM turbo encoder	16
2.2.2 RM turbo decoder	19
2.2.3 Interleaver	23
2.3 Minimal trellis construction of RM code	24
2.4 Trellis-based MAP algorithm	27
2.5 Summary	31
3. Algorithms for implementation	32
3.1 Introduction	32
3.2 Max-Log-MAP algorithm	33
3.3 Max-Log-MAP algorithm with correction	35

3.4 Arithmetic units	38
3.4.1 Pre-processing model	38
3.4.2 Gamma calculation model	39
3.4.3 Alpha/Beta calculation model	40
3.4.4 Extrinsic information calculation model	41
3.4.5 Log-likelihood soft-output calculation model	42
3.5 Summary	42
4. Design and implementation of RM-BTC	43
4.1 Review of turbo code implementation	44
4.2 Hardware choices and FPGA	45
4.2.1 Virtex-E FPGA architecture	47
4.2.2 Arithmetic function cores	50
4.3 Design issues	51
4.3.1 Quantization and finite accuracy	51
4.3.2 Data- and hardware-sharing	53
4.3.3 Parallelism and pipeline	55
4.3.4 Iteration stop criterion	56
4.3.5 (De)-interleaving technique	57
4.4 Turbo encoder implementation	59
4.4.1 Interface input/output	59
4.4.2 Generator matrix and trellis	60
4.4.3 Finite state machine	62
4.4.4 Encoder data path	62
4.5 Turbo decoder implementation	64
4.5.1 Interface input/output	64
4.5.2 Finite state machine	68
4.5.3 Pre-processing component	70
4.5.4 Gamma unit	71
4.5.5 Alpha/Beta unit	71
4.5.6 Extrinsic information unit	73

4.5.7 LLR unit	75
4.6 Memory architecture	75
4.7 Summary	79
5. Simulation and synthesis	80
5.1 Test model	80
5.2 Simulation	81
5.3 Synthesis and comparison	84
5.4 Summary	87
6. Conclusion	88
6.1 Summary	88
6.2 Practical limitation and solutions	89
 Bibliography	
Appendix A Synthesis reports	98
Appendix B Turbo encoder VHDL source codes	102
Appendix C Turbo decoder VHDL source codes	113

List of Figures

Figure 1.1	Typical digital communication block diagram	2
Figure 2.1	Block diagram of parallel turbo code	14
Figure 2.2	Re-ordered two dimensional RM turbo code	18
Figure 2.3	Soft-in-Soft-out decoder	21
Figure 2.4	Iterative decoding procedure with initial $L^h(x) = 0$	21
Figure 2.5	One section of trellis structure of block code	28
Figure 3.1	Log-MAP decoding flow chart diagram	37
Figure 3.2	Comparison of different algorithm on performance	38
Figure 4.1	Virtex-E architecture overview	48
Figure 4.2	A 2-slice Virtex-E CLB	49
Figure 4.3	A detailed view of Virtex-E slice	49
Figure 4.4	Different bit quantization effects on performance	52
Figure 4.5	4-bit quantization on different SNR values	53
Figure 4.6	I/O interface of turbo encoder	59
Figure 4.7	The illustrative trellis diagram of RM(32,26) code	61
Figure 4.8	Finite state machine of turbo encoder	62
Figure 4.9	Data path of turbo encoder	63
Figure 4.10	Data path architecture of Log-MAP turbo decoder	66
Figure 4.11	I/O interface of turbo decoder	67
Figure 4.12	Finite state machine of turbo decoder	69
Figure 4.13	Data path of branch metric calculation	71
Figure 4.14	Data path of forward/backward recursion calculation	73
Figure 4.15	Data path of extrinsic information calculation	74
Figure 4.16	Data path of log-likelihood ratio calculation	75
Figure 5.1	Block diagram of test model	81

List of Tables

Table 4.1 Input/Output interface definition of turbo encoder	60
Table 4.2 Generator memory address and cell content	61
Table 4.3 Input/Output interface definition of turbo decoder	67
Table 4.4 L_c values for E_b/N_0 at rate $R=0.684$	70
Table 4.5 Look-up table for correction term	72
Table 4.6 Memory description of turbo encoder and decoder	77
Table 5.1 BER/Block versus SNR values	83
Table 5.2 Design summary of turbo encoder	85
Table 5.3 Design summary of turbo decoder	85
Table 5.4 Comparison between implementations	86

List of Acronyms

ARQ	automatic repeat request
ASIC	application specific integrated circuit
AWGN	additional white gaussian channel
BCH	Bose-Chaudhuri-Hocquenghem code
BER	bit error rate
BPSK	binary phase shift key
BTC	block turbo codes
CLB	configurable logic block
CTC	convolutional turbo codes
DSP	digital signal processing
ECC	error correction codes
FEC	forward error correction
FIFO	first in first out
FPGA	field programmable gate array
FSM	finite state machine
GRM	programmable routing matrix
HCCC	hybrid concatenated convolutional codes
IOB	input/output block
LC	logic cell
LUT	look-up table
MAP	maximum a posteriori probability
MSB	most significant bit
PCCC	parallel concatenated convolutional codes
RAM	random access memory
RM-BTC	Reed-Muller block turbo codes
ROM	read only memory
SCCC	serial concatenated convolutional codes
SISO	soft in soft output

SNR	signal to noise ratio
SOVA	soft-output Viterbi algorithm
SRAM	static random access memory

Chapter 1

Introduction

Increasing demand for information exchange is a characteristic of modern civilization, and the distance can't prevent people from communicating across the world. The past ten years have seen rapid growth in mobile communication and this is expected to continue. The information transmission should be done in such a way that the received information should be as close as possible to the transmitted information. Therefore, providing a high speed data transfer, reliable and cost-effective data communication system has drawn much attention in the past decades.

1.1 Introduction

Since digital techniques offer many benefits in technical implementation and economic benefit, digital communication systems have become the main stream technology in communication systems. A typical digital communication system is shown in Figure 1.1[3]. The original signal source feeds into source encoder which efficiently compresses data with little or no redundancy; the binary output data stream from the source encoder, called the information sequence, is passed to the channel encoder, which introduces some controlled redundancy in binary information sequence. The added redundancy can be used at the receiver to overcome the effects of noise and interference encountered in the trans-

mission of the signal through the noisy channel. So the coded output from the channel encoder provides the error detecting and correcting capability to the channel decoder at the receiver end. The digital modulator maps the coded word into a set of waveforms that are suitable for transmission over the channel, where the data is usually corrupted by noise. At the receiving end, the demodulator processes the channel-corrupted waveforms and following detector will make a hard or soft decision based on decoding algorithm used. With the knowledge of the code used by the channel encoder, the channel decoder processes the quantized/unquantized output from the detector and outputs the estimated coded word whose errors have been detected and corrected. As a final step, source decoder attempts to reconstruct the original information sequence generated by the source.

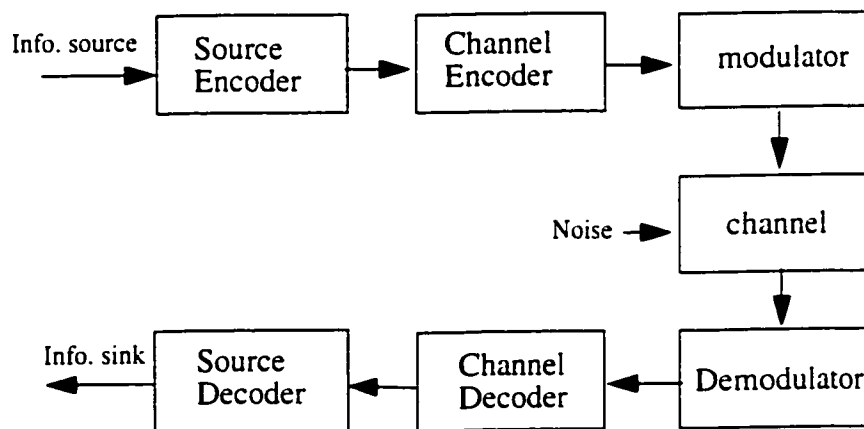


Figure 1.1 Typical digital communication block diagram

The frequency of errors occurring in the decoded sequence is a measure of reliable information transfer. It is called bit error rate (BER). In general, it is related to the code characteristics, modulation type, transmitter power and the method of demodulation and decoding used, etc.. In this thesis, we focus on an algorithm and its implementation for channel coding, which minimizes the error bits in the decoded sequence.

The channel decoder can control errors in many ways[4]. For example, a bilateral exchange, called automatic repeat request (ARQ), can minimize the amount of wasted effort/bandwidth needed to control errors when both forward- and reverse-channel communications are reliable. However, in many cases, the reliable reverse channel is unavailable, and the system cannot tolerate the delay imposed by an ARQ system. So the forward error correction (FEC) is used mainly in protecting data against disturbance introduced by the physical channel.

Forward error correction adds redundancy to the data stream at the transmitter end so that the receiver can both detect and correct errors unilaterally. However, this requires expanded bandwidth to transfer the coded sequence including the original information and the error-correcting redundancy. Although increasing transmitted power and expanding bandwidth can reduce the error rate sufficiently, many applications are strictly power-limited and bandwidth-limited. Therefore, what engineers consider is to find a powerful forward error correcting coding scheme that balances the transmit power, bandwidth and data reliability.

While channel coding provides protection to transmitted information, it is also required to reduce the transmit power which is normally represented in terms of coding gain. The coded system requires less signal-to-noise ratio (SNR) than uncoded system to achieve the same bit error probability. This reduction expressed in decibel (dB) is called coding gain. A large coding gain means less transmit power. Therefore, a powerful coding scheme is to achieve as much coding gain as possible[3,5]. Nevertheless, the construction and selection of the coding type depend on some constraints and requirements such as channel environment, implementation complexity, as well as performance, speed and

bandwidth, etc.. That's why the development of error-correcting code (ECC) has been continuously improving for half century since the concept was proposed by Claude E Shannon.

1.2 Review of Error Correcting Codes

Claude E Shannon laid the foundation for modern digital communication with his ground breaking paper [6] in 1948. The two classical theorems in information theory are the source coding theorem and the channel coding theorem. The source coding theorem shows the smallest number of bits required to represent a given source without any loss; the channel coding theorem shows the maximum theoretical data rate with 'good' data code for reliable communications. The theorems show that if the data rate is less than the channel capacity, reliable information transmission can be achieved if one chooses proper encoding and decoding techniques.

There are two broad categories of codes used in communication system for error control coding, block codes and convolutional codes. In block codes, a block of n digits generated by the encoder in a particular time unit depends only on the block of k input message digits within that time unit. The larger the block length n , the smaller the probability of decoding error. In convolutional codes, a block of n digits generated by the encoder in a time unit depends not only on the block of k message digits within that time unit, but also on the preceding $(N-1)$ blocks of message digits. Due to the difference between them, block codes are better suited for error detection and convolutional codes are mainly used for error correction. Recent studies have shown that convolutional codes perform as well or better than block codes in many error control applications[7-9].

Bose-Chaudhuri-Hocquenghem (BCH) codes, discovered by Bose, Chaudhuri[69] and Hocquenghem[70] in 1959, are among the most extensively used powerful error-correcting cyclic codes known. The important sub-class of BCH codes, Reed Solomn codes were proposed by Reed and Solomn in 1960[71]. It has a better performance at higher SNR region and copes well with burst error, whereas convolutional codes outperforms the Reed Solomn code at lower SNR region.

Burst errors often occur in storage media such as tapes and compact discs because of defects. Aforementioned codes concentrate on codes capable of correcting random errors. They are in general not efficient at correcting burst errors. In order to take advantages of each code, some new type of codes are introduced in coding schemes such as Interleaved, Product and Concatenated Codes. Because the large block lengths offer the small probability of decoding error, these codes are long block length codes that are derived from short block codes.

An interleaved code can correct a single burst of length $A\tau$ or less if the original code can correct a single burst of length τ or less, where A is degree of interleaving and τ is burst of length. However, interleaving a code introduces delay (delayed interleaving) because transmission cannot begin until all A code words have arrived.

A product code is an (N_1N_2, K_1K_2) code in which each code word forms an $N_1 \times N_2$ rectangular array such that each row is a code word from an (N_1, K_1) code C_1 and each column is a code word from an (N_2, K_2) code C_2 . Product code is capable of correcting burst errors of length τ' or less, where $\tau' = \max(N_1\tau_2, N_2\tau_1)$ if C_1 and C_2 are capable of cor-

recting bursts of length τ_1 and τ_2 . Product codes were proposed by Elias and they can be extended to higher dimension.

Concatenated coding was introduced by Forney[2] as a practical technique for implementing a code with a very long block length and a large error-correcting capability. The principle is the application of two levels of coding, an inner and an outer code linked by an interleaver. The inner encoder, channel and the inner decoder are viewed as a super channel. The information is first encoded by outer encoder, for example Reed Solomn encoder, and then the output from outer encoder is fed into the inner encoder, for example convolutional encoder. The decoding process is in the reverse order: the received sequence with errors is decoded by inner decoder with hard or soft decision algorithm and then decoded by outer decoder to correct the residual errors.

Turbo codes, introduced by Berrou *et al.*[10], built from a particular concatenation of two recursive systematic convolutional codes linked by nonuniform interleaving, is the biggest breakthrough in recent error correction code [ECC] history[10,11]. For the AWGN channel, turbo codes perform to within 0.7 dB of Shannon's limit as opposed to 2 dB or more for other state-of-the-art techniques of similar complexity. Since 1993, turbo codes have become the hottest topic in ECC field. Soon after convolutional turbo codes (CTC) was proposed by Berrou *et al.* in 1993[10], the concept of turbo codes was extended to block turbo codes (BTC) in particular by Pyndiah *et. al*[12,13]. The difference between CTC and BTC is the component code used. Convolutional codes are used as component code in CTC whereas the block codes are used in BTC. Because of its excellent forward error correction capability and superior coding gain, the utilization of turbo codes has increased dramatically on AWGN channels, such as satellite, wireless and space commu-

nications which require high performance at very low SNR. In Chapter 2, the details of BTC will be discussed.

There are many decoding algorithms for different code constructions such as the efficient iterative algorithm proposed by Berlekamp for BCH codes[8]; the Viterbi algorithm for minimizing the probability of word error for convolutional codes[3]; the MAP algorithm proposed by Bahl *et. al* for minimizing the symbol (or bit) error probability by estimating the *a posteriori* probabilities of the states and transition of a Markov source through a discrete memoryless channel[14]. In this thesis, we are interested in the trellis-based MAP algorithm because the state transition and decoding process can be represented by a trellis diagram, and it is more efficient than other non-trellis algorithms.

Most algorithms can accept hard- or/and soft-input and perform hard- or/and soft-decision decoding. In the hard decision decoding, the input of the decoder is quantized into two levels; whereas in the soft decision, the input is quantized into more than two levels. Multi-level quantization or the real channel information input makes soft-input decoding to have more advantages over hard-input decoding due to more reliable information being available. The additional information provided by the soft decision in most instances can provide about 2 dB of additional coding gain and can significantly increase the usefulness of a particular code[1].

Based on the MAP algorithm in logarithmic domain[15,16], J. Hagenauer *et al.* gave a detailed analysis and mathematical formula about the iterative decoding of both block and convolutional turbo codes with MAP and its modified versions[17]. An alternative coding scheme with Reed-Muller code as component code was presented in [5]. The two papers [5, 17] are the basis of decoding algorithm used in this thesis.

1.3 Objective of thesis

Although turbo codes have outstanding performance, they are computational complex. Also there exists many technical difficulties in the implementation of a practical turbo-coded system. The main objective of this thesis is to design the hardware architecture and arithmetic function units of Reed-Muller block turbo codes (RM-BTC) Codec in RTL model, and aim at the implementation using Field Programmable Gate Array (FPGA), and produce a prototype. The second goal is to reduce the chip area as much as possible. The actual decoding speed and chip area obtained through synthesizing and mapping to Xilinx FPGA are taken as a reference for future improvement.

1.4 Contribution

This thesis is an engineering-intensive design work based on an existing decoding algorithm. It presents a practical implementation of a turbo-code codec on a Xilinx FPGA chip. The main contributions contains:

- Investigation and modification of the decoding algorithms suitable for hardware implementation.
- Behavioral simulation of the modified algorithm and quantization of variables.
- Hierarchical architecture design of codec and functional units and finite state machine (FSM) for codec. Especially employing a single decoder for both horizontal and vertical decoding for the sake of saving chip area without performance degradation.
- Simulation and synthesis of proposed hardware implementation.
- Correct gate-level simulation of turbo encoder.

1.5 Overview of thesis

The focus of this thesis is on the implementation of turbo-coded codec. It is organized as follows:

Chapter 2 provides an introduction to block turbo codes related to this thesis. Besides the basic concepts of block turbo codes, the construction of Reed-Muller code and its minimal trellis diagram using Massey algorithm are given. After discussion of the soft-in-soft-out (SISO) decoder and parallel iterative decoding, trellis-based MAP algorithm for linear BTC is presented.

Chapter 3 studies the modified algorithm to be implemented based on trellis-based decoding. The Log-MAP and Max-Log-MAP as well as Lookup-table-based Log-MAP algorithms are discussed in detail. Decomposition of each arithmetic functional unit for forward recursion, backward recursion, branch probability as well as extrinsic information calculation is provided. Finally, the iterative decoding procedures are given as a guideline of FSM design.

Chapter 4 contains all aspects of design and implementation of RM-BTC codec. First, the review of turbo code implementation in the past is given; some issues such as quantization, fixed-point representation are also reviewed. Second, the aimed target implementation Xilinx Virtex-E FPGA architecture is provided. Finally, according to the modified algorithm and target device, the detailed designs of RM-BTC encoder and decoder are given, which include the overall architecture, finite state machine and data path components.

Chapter 5 covers the results of simulation and synthesis of the RM-BTC encoder and decoder. A variety of test information blocks are sent to the encoder and the decoder

output is compared with original information block under different SNRs. The hardware mapping results are given for encoder and decoder, respectively.

Chapter 6 summarizes the work presented in this thesis, and discusses several ideas for improving the current design in future work.

Chapter 2

Block turbo codes

In this chapter, we first discuss some basic concepts related to block turbo codes [3,18], which are relevant to this thesis. Then Reed-Muller code used in this thesis is discussed. Minimal trellis construction and trellis-based decoding of RM-BTC are discussed in detail.

2.1 Basics of block turbo codes

2.1.1 Linear block code

A block code $C(n,k)$ consists of $M = q^k$ code words of length n ($k < n$), whose elements are selected from an alphabet of q elements of $GF(q)$. the ratio k/n is defined to be the code rate which determines the amount of redundancy. Here we just consider the binary block code, i.e. $q = 2$. The 2^k distinct code words are a subset of 2^n binary sequence of length n , which are selected to transmit k -bit blocks of information.

An (n,k) block code is linear if component-wise modulo- q sum operation of any two code words is another code word. In our case, modulo-2 sum is used in obtaining another code word. Viewing from linear algebra, an (n,k) linear block code is a k -dimen-

sional subspace of the n -dimensional space of all the binary n -tuples. We can construct the linear block code with vector and matrix format.

The k -bit information vector $\mathbf{M} = (m_0, m_1, m_2, \dots, m_{k-1})$ would be encoded into block codeword \mathbf{C} of length n , $\mathbf{C} = (c_0, c_1, c_2, \dots, c_{n-1})$, the linear operation performed in a linear block encoder can be represented as

$$\mathbf{C} = \mathbf{M} \times \mathbf{G} \quad (2.1)$$

where \mathbf{G} is called the generator matrix, with a set of k linearly independent binary n -tuple as its row, arranged as a $k \times n$ matrix.

Further, a linear systematic block code can be constructed when the generator matrix \mathbf{G} can be reduced to the following special “systematic” form by a sequence of elementary row operations and column permutations, $\mathbf{G} = [\mathbf{I}_k: \mathbf{P}]$, where \mathbf{I}_k is a $k \times k$ identity matrix and \mathbf{P} is a $k \times (n-k)$ matrix determining the parity check bits.

In the linear systematic block code, the first k bits of the codeword are identical to the information bits to be transmitted. This enables direct extraction of information bits at decoder from the codeword without attempting to recover information bits. As seen in the following formula, code word \mathbf{C} is made up of two part: the information bits m_i and the parity check bits p_{ji} .

$$c_i = \begin{cases} m_i & 0 \leq i \leq k-1 \\ \sum_{j=0}^{k-1} p_{ji} \cdot m_j & k \leq i \leq n-1 \end{cases} \quad (2.2)$$

This form simplifies the decoding operation later. However, the position of information bits is not necessarily restricted to the first k bits of code words if the position of

information bits can be obtained through systematic-like G matrix. We still treat this form of code word as systematic-like code.

2.1.2 Block turbo codes

Today, concatenated coding schemes have proven to be attractive schemes for obtaining high coding gain with moderate decoding complexity. They aim to achieve the same performance as that of a single long and powerful error correcting code but with a lower complexity when associating two or more less powerful error correcting codes. Concatenated coding schemes can be divided into three categories: parallel, serial and hybrid concatenated coding[28]. In practice, the first two schemes are widely used because of the reasonable complexity of the decoders given a certain coding gain, especially the classic serial one with Reed-Solomon code as the outer code and convolutional code as inner code. Because the hybrid code introduces a considerable amount of delay, it is normally suitable for extremely high data rates where the resulting delay is tolerable.

Turbo codes exploit the similar idea of concatenated code to achieve a low error rate with less decoding complexity than that required for a single code of the corresponding performance. Turbo codes can be in form of two- or three-dimensional which consists of concatenation of two or three component codes separated by interleavers. Two-dimensional turbo codes normally consist of two elementary encoders, which construct turbo encoder and two soft-in-soft-out (SISO) decoders, which construct turbo decoder and interleaver/de-interleavers. According to how information bits are encoded and decoded, there are three kinds of turbo codes: parallel, serial and hybrid turbo codes (PCCC, SCCC and HCCC)[26]. Illustrated by some simulation results[26], SCCCs outperform PCCCs at

lower BER region but PCCC has better performance at higher BER values. In what follows we take a two-dimensional turbo code to illustrate the basic concepts of parallel turbo code. The parallel turbo encoder and decoder are shown in Figure 2.1.

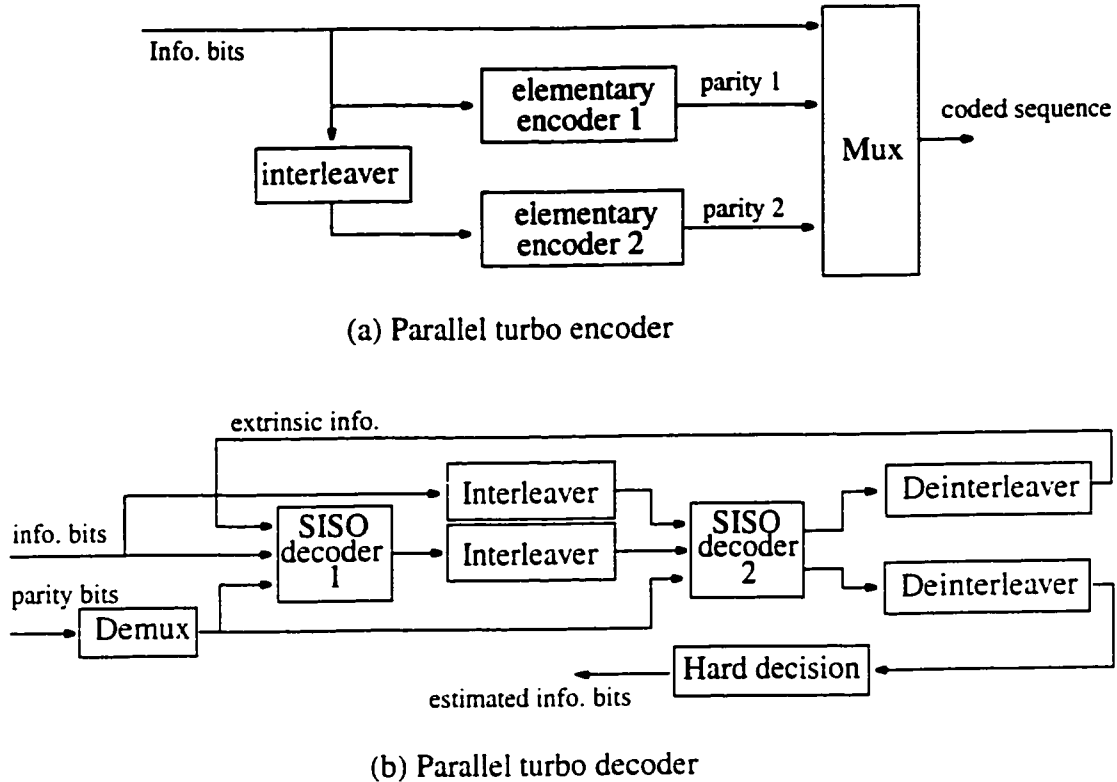


Figure 2.1 Block diagram of parallel turbo code

Although the term “turbo code” is used to refer to a wide variety of concatenated coding schemes, its initial version is solely a parallel concatenation of convolutional codes (PCCC) whose encoder is formed by parallel concatenation of two recursive systematic convolutional codes joined by an interleaver. In parallel turbo encoder shown in Figure 2.1(a), two elementary encoders are connected in parallel by an interleaver. The information bits and its interleaved sequence are input to elementary encoders 1 and 2, respectively. Each encoder encodes the information bits and output the coded sequence consisting of the information and parity check bits. The information bits for the encoder 2

are not transmitted because it is same as that of the encoder 1. This is different from conventional concatenated codes. Therefore, turbo encoder output is made up of three parts: information bits, parity 1 and parity 2 bits, which will be transmitted over the physical channel. One advantage of the parallel scheme is that the two encoders can work clock-synchronously.

In the parallel turbo decoder shown in Figure 2.1(b), two soft-in-soft-out (SISO) decoders linked by an interleaver realize the iterative decoding of the received sequence. The soft output of the received information and parity 1 bits from the demodulator and extrinsic information from the SISO decoder 2 as *a priori* value are taken as the inputs of the SISO decoder 1; the soft output of the interleaved received information and parity 2 bits as well as interleaved extrinsic information from decoder 1 are taken as inputs to the SISO decoder 2. Similarly, the extrinsic information generated by decoder 2 are deinterleaved and feedback to decoder 1 for next iterative decoding. After a presetting iterative decoding process, the hard decision is made according to the sign of the deinterleaved soft output of the decoder 2, and the magnitude of soft output is the reliability of this decision.

2.2 Reed-Muller turbo codes

The choice of the constituent code has a strong influence on the overall performance of turbo codes. Long block length (interleaver size) yield remarkable performance; however, in wireless applications, the block length is much smaller, on the order of a few hundred bits or less. The advantage of using a block code is that the memory requirements of the MAP decoding can be reduced significantly; also the decoding delay is set by the length of the block and does not depend on the transmission rate.

In our turbo code scheme, Reed-Muller codes are used as constituent code because they have modular structure for constructing longer codes with shorter ones. Their trellis can be easily constructed, further they can be decoded effectively with trellised-based decoding algorithms. In this section, we first present some basics of the Reed-Muller codes[19, 20], then discuss the construction of RM turbo codes. Finally the basics of RM turbo decoding is followed.

2.2.1 RM turbo encoder

Reed-Muller codes are a class of linear codes over GF(2). They can be constructed by procedures for constructing a generator matrix. For a binary r -th order $RM_{(r, m)}$ code of length $n = 2^m$ with $0 \leq r \leq m$, there exists m 2^{m-i} -tuple vectors v_i over GF(2), $1 \leq i \leq m$, in the following form:

$$v_i = \left(\underbrace{0 \dots 0}_{2^{i-1}}, \underbrace{1 \dots 1}_{2^{i-1}}, \underbrace{0 \dots 0}_{2^{i-1}}, \dots, \underbrace{1 \dots 1}_{2^{i-1}} \right) \quad (2.3)$$

that consists of 2^{m-i+1} alternate all-zero and all-one 2^{i-1} -tuples. Let

$\vec{a} = (a_1, a_2, \dots, a_n)$ and $\vec{b} = (b_1, b_2, \dots, b_n)$, and the product of \vec{a} and \vec{b} is defined as

$$\vec{a} \cdot \vec{b} = (a_1 \cdot b_1, a_2 \cdot b_2, \dots, a_n \cdot b_n) \quad (2.4)$$

For $1 \leq i_1 \leq i_2 \leq \dots \leq i_l \leq m$, the product $v_{i_1} \cdot v_{i_2} \cdot \dots \cdot v_{i_l}$ is said to have degree l .

we also denote $\vec{1}$ the all-one 2^m -tuple, $\vec{1} = (1, 1, \dots, 1)$.

Now the generator matrix G of r -th order RM code, $RM(r, m)$ of length 2^m is generated by the following set of vectors:

$$G(r, m) = \{ \overset{\rightarrow}{1}, v_1, v_2, \dots, v_m, v_1 \cdot v_2, v_1 \cdot v_3, \dots, v_{m-1} \cdot v_m, \dots \} \quad (2.5)$$

up-to-products-of-degree-r

The $RM(r, m)$ codes have the following parameters:

The code length $n = 2^m$ and the dimension k of G matrix is

$$k_{r,m} = 1 + \binom{m}{1} + \binom{m}{2} + \dots + \binom{m}{r}$$

For convenience, from now on in this thesis, $RM(n, k)$ form is used instead of

$RM_{(r,m)}$.

In an example of $RM(16, 11)$ code, the 2-th RM code of length 16 is generated by the following G matrix:

$$G_{RM(16, 11)} = \begin{bmatrix} (v_0 = 1) \\ v_4 \\ v_3 \\ v_2 \\ v_1 \\ v_3 v_4 \\ v_2 v_4 \\ v_1 v_4 \\ v_2 v_3 \\ v_1 v_3 \\ v_1 v_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.6)$$

The above matrix is not in “systematic” form, so the code generated is not systematic code. In order to simplify the encoding and decoding, G matrix can be reduced by row operations and column permutations to a systematic-like form which clearly indicates the position of information and parity bits. Because turbo codes are linear block codes, the

encoding operation can be viewed as modulo-2 multiplication of information vectors with the generator matrix.

In this thesis, two dimensional $RM(32,26)^2$ codes are used as constituent codes and parallel concatenated scheme is adopted. Therefore, two elementary encoders in Figure2.1(a) are instantiated with two $RM(32,26)$ encoders. The information block and its permuted version after block interleaver are encoded by two $RM(32,26)$ encoders, namely horizontal and vertical encoder corresponding to elementary encoder 1 and 2, respectively. Because we only transmit the original information bits once, this turbo code has the following re-ordered pattern shown in Figure 2.2.

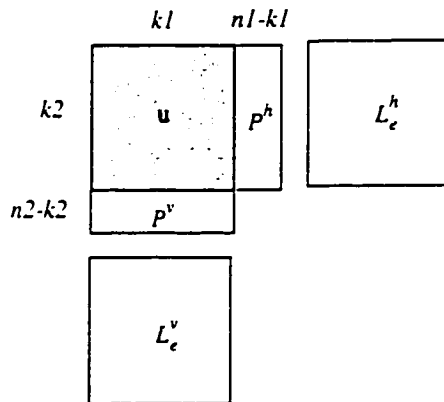


Figure 2.2 Re-ordered two dimensional RM turbo code

The horizontal encoder generates $C_1(n_1, k_1)$ and vertical one generates $C_2(n_2, k_2)$ linear systematic-like RM code, where n_i and k_i are code length and information length of code C_i , $i = 1, 2$. Horizontal parity bits P^h and vertical parity bits P^v are added by horizontal and vertical RM encoder accordingly based on the information data $u = k_1 \times k_2$ and its interleaved version. This code has no parity bits on another parity part.

The Figure 2.3 has the ideal systematic form through re-ordering information and parity bits. When decoding, we need to perform the inverse operation to obtain the original code word order.

2.2.2 RM turbo decoder

Before discussing the RM turbo decoder, we first present some basics of the log-likelihood algebra and SISO decoder as well as the iterative decoding principle[11,17].

Assume BPSK modulation and natural logarithm are used in the following discussion. Let a binary random variable X be in $GF(2)$ with elements $\{+1, -1\}$. The log-likelihood ratio of x , $L_X(x)$, is defined as $L_X(x) = \log \frac{P_X(x=1)}{P_X(x=-1)}$, which will be denoted as

the soft value. When an (n,k) systematic code is transmitted over a Gaussian channel, the log-likelihood ratio of bit x conditioned on the match filter output y is

$$\begin{aligned}
 L(x|y) &= \log \frac{P_X(x=1|y)}{P_X(x=-1|y)} \\
 &= \log \left(\frac{p(y|x=1)}{p(y|x=-1)} \cdot \frac{P(x=1)}{P(x=-1)} \right) \\
 &= \log \frac{\exp\left(-\frac{E_s}{N_0}(y-a)^2\right)}{\exp\left(-\frac{E_s}{N_0}(y+a)^2\right)} + \log \frac{P(x=1)}{P(x=-1)} \\
 &= L_c \cdot y + L(x)
 \end{aligned} \tag{2.7}$$

where $L_c = 4 \cdot a \cdot \frac{E_s}{N_0}$ is called the reliability value of the channel, and a denotes

the fading amplitude whereas for Gaussian channel $a = 1$. The soft-output of the bit x_k ,

$L(\hat{x}_k)$ of the *maximum a posteriori* (MAP) decoder is defined as *a posteriori* log-likelihood ratio of bit x_k conditioned on the received sequence \hat{y} as follow

$$\begin{aligned}
 L(\hat{x}_k) &= L(x_k|\hat{y}) \\
 &= \log \frac{P_X(x_k = 1|\hat{y})}{P_X(x_k = -1|\hat{y})} \\
 &= \log \frac{p(\hat{y}|x_k = 1)}{p(\hat{y}|x_k = -1)} \cdot \frac{P(x_k = 1)}{P(x_k = -1)} \\
 &= \log \left(\frac{p(y_k|x_k = 1)}{p(y_k|x_k = -1)} \cdot \prod_{i=1, i \neq k}^n \frac{p(y_i|x_k = 1)}{p(y_i|x_k = -1)} \cdot \frac{P(x_k = 1)}{P(x_k = -1)} \right) \quad (2.8) \\
 &= \log \frac{p(y_k|x_k = 1)}{p(y_k|x_k = -1)} + \log \left(\prod_{i=1, i \neq k}^n \frac{p(y_i|x_k = 1)}{p(y_i|x_k = -1)} \right) + \log \frac{P(x_k = 1)}{P(x_k = -1)} \\
 &= L_c \cdot y_k + L_e(\hat{x}_k) + L(x_k)
 \end{aligned}$$

The above equation consists of three independent estimates for the log-likelihood ratio of the information bit x_k : channel value $L_c \cdot y_k$ for all coded bits; the extrinsic information $L_e(\hat{x}_k)$ obtained from all the other code bits in the code sequence except x_k ; the *a priori* value $L(x_k)$ which equals to zero for the first iteration under assumption of equally likely information bits.

Soft-in-soft-out (SISO) decoder block diagram is shown in Figure 2.3. The log-likelihood input value $L(x)$ feeds to SISO, which decodes received sequence y with MAP algorithm or SOVA algorithm and outputs the log-likelihood ratio of *a posteriori* $L(\hat{x}_k)$ and extrinsic values $L_e(\hat{x})$ for all information bits.

Iterative decoding scheme is an important feature of turbo codes. Turbo codes use an iterative decoding algorithm where the BER performance improves after each iteration.

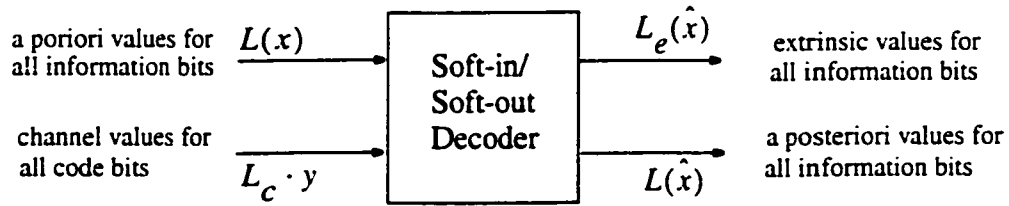
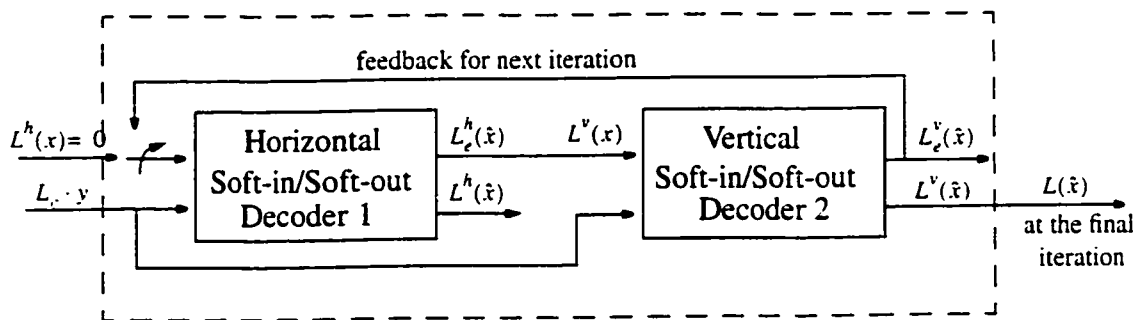


Figure 2.3 Soft-in-Soft-out decoder

In order to optimize the global decoding performance of two concatenated decoder, named horizontal and vertical, the output extrinsic value of the vertical decoder is fed back to the horizontal decoder for further efficient utilization as a diversity effect, and this extrinsic information makes the horizontal decoder to obtain additional redundant information that may significantly improve its performance. However, this performance improvement is limited by the interleaver length and the code structure. The longer the interleaver, the more coding gain can be obtained from increasing the number of iterations. For a given interleaver length, the coding gain becomes negligible after a certain number of iterations. Reference [5] proves that when the block interleaver was used, the coding gain is saturated at 2nd, 4th and 5th iteration for $RM(8,4)^2$, $RM(16,11)^2$ and $RM(32,26)^2$, respectively.

The iterative feedback decoding scheme with two soft-in-soft-out decoders is shown in Figure 2.4. Its decoding procedure is as follows.

Figure 2.4 Iterative decoding procedure with initial $L^h(x) = 0$

1. Assuming equally likely input information bits, so set the *a priori* $L^h(x)$ to zero at first iteration.
2. Decoding of horizontal code C^h by using the corresponding $L_c \cdot y$ of information part and horizontal parity check part. The horizontal extrinsic information $L_e^h(\hat{x})$ of C^h on information bit x is:

$$\begin{aligned} L_e^h(\hat{x}) &= L^h(\hat{x}) - L_c \cdot y - L^h(x) \\ &= L^h(\hat{x}) - L_c \cdot y \end{aligned} \quad (2.9)$$

3. Set $L^v(x) = L_e^h(\hat{x})$, i.e. the horizontal extrinsic information on information bit x is fed to the vertical decoder as the *a priori* value for decoding vertical code C^v while using the corresponding $L_c \cdot y$ of the interleaved information part and vertical parity check part. The vertical extrinsic information $L_e^v(\hat{x})$ of code C^v on information bit x is:

$$L_e^v(\hat{x}) = L^v(\hat{x}) - (L_c \cdot y) - L_e^h(\hat{x}) \quad (2.10)$$

4. Set $L^h(x) = L_e^v(\hat{x})$, i.e. the vertical extrinsic information bit x is fed back to horizontal decoder as *a priori* value for decoding of horizontal code C^h in the next iteration.

5. After the last iteration, the soft output of turbo decoder on the information bit x is:

$$L(\hat{x}) = L_c \cdot y + L_e^h(\hat{x}) + L_e^v(\hat{x}) \quad (2.11)$$

6. According to the sign of the soft output $L(\hat{x})$, the hard decision is made to obtain the estimated transmitted information bits.

In this thesis, RM turbo decoder takes parallel concatenated form of turbo codes shown in Figure 2.1(b). The component decoder uses SISO decoder with modified Log-MAP algorithm based on the minimal trellis diagram. Interleaver will take block interleaving techniques which is discussed in the next section.

2.2.3 Interleaver

Another key feature of turbo codes is the interleaver. It plays an important role in achieving good performance of turbo codes. There are four major interleaving structures widely used in conjunction with error-correcting schemes: block interleaver, convolutional interleaver, random interleaver and code matched interleaver[18,28].

The interleaver in a turbo code scrambles the bits in each block of data before it enters the second encoder, so that the inputs to the two encoders are not correlated. For the burst error channel, this process spreads out the burst error. After correction of some of errors in the first component decoder, some of the remaining errors become correctable in the second decoder. Therefore, by de-coupling the inputs to the two encoders, the interleaver provides a good codeword weight distribution, which improves the decoder performance.

Turbo codes have a feedback path for extrinsic information to be used in the next iteration, and iterative decoding algorithm is used. Therefore, the decoding performance of turbo decoder depends on the structure and the size of the interleaver. For a given set of component codes, the turbo code with a longer interleaver has a better performance. However, longer interleaver introduces long delays which are not desirable in low data rate. That's why turbo codes are particularly attractive to higher data rate application.

A block interleaver formats the input sequence in a matrix of $m \times n$, in which the input sequence is written into matrix row-wise and read out column-wise. It is a simple interleaving method with sound performance. For our RM-BTC, the delay of block interleaver is fixed.

2.3 Minimal trellis construction of RM code

A trellis is a compact method of representing all of the q^k codeword of a linear code. A path from the root to a terminal node is referred to as a path through the trellis, and each distinct codeword corresponds to a distinct path in the trellis. Thus the total number of paths in the trellis is $N_p = q^k$. It has been found that an arbitrary linear (n, k, d_{min}) block code over $GF(q)$, $q \geq 2$, can be represented by its trellis diagram which contains $N_c = n + 1$ columns and $N_s \leq q^{\min\{k, n-k\}}$ states[21]. In $RM(32, 26)$ block code, we have the following state profile corresponding to 33 column:

State profile = {0, 1, 2, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 3, 3, 2, 1, 0}

where each number in parenthesis is the exponential value of base 2 at its own position. Consequently, we may obtain the state number at each time instant. For instance, at time instant 5, we have $2^4 = 16$ states and at time 9 have $2^5 = 32$ states.

Trellis structure determines the complexity of the trellis-based decoding for a linear block code. This can be expressed in terms of some quantities of the trellis such as the logarithms of the maximum numbers of states and branches at any time index. The trellis structure of block encoder is irregular in comparison to the trellis of a convolutional code

the decoding complexity is equivalent to finding the minimal trellis diagram of a given code. Among the many trellis representations of block code, a minimal trellis diagram of a code C of length n is the one that has the minimal state numbers than in any other trellis for C at the time index i , $i = 0, 1, \dots, n$. In this thesis, the Massey construction method is used to construct the minimal trellis of the RM code because of its systematic property. Some basic knowledge related to minimal trellis construction are presented below[22].

- Left index $L(X)$ of X : Given a nonzero vector $X = (x_1, x_2, \dots, x_n)$ over $GF(q)$, $L(X)$ denotes the smallest integer i such that $x_i \neq 0$.
- Row-reduced echelon matrix: Given a $k \times n$ matrix $G = [x_{i,j}]$ over $GF(q)$, G is in row-reduced echelon form if the rows X_1, X_2, \dots, X_k of G are such that $L(X_1) < L(X_2) < \dots < L(X_k)$, and the k columns found at positions $L(X_i)$: $i = 1, 2, \dots, k$ in G are all of weight one.

In what follows we discuss Massey construction of minimal trellis. Let linear code $C(n, k)$ be constructed by row-reduced echelon generator matrix $G[k, n]$. The left indices of its rows is denoted as $\gamma_1, \gamma_2, \dots, \gamma_k$. This implies that $\gamma_1 < \gamma_2 < \dots < \gamma_k$ and the k positions $\gamma_1, \gamma_2, \dots, \gamma_k$ form the information bits for code C .

The Massey trellis $T = (V, B, L)$ for code C is constructed by identifying the vertices in V_i at time i : $i = 0, 1, 2, \dots, n$. Let m be the largest integer such that $\gamma_m \leq i$. then V_i is defined as

$$V_i = \{(c_{i+1}, \dots, c_n) \cdot (c_1, \dots, c_n) = (u_1, \dots, u_m, 0, \dots, 0)G\} \quad (2.12)$$

By convention, $V_0 = \{0\}$ and $V_n = \{\emptyset\}$ where \emptyset is the empty string.

The branch set of T is defined as follows in two cases:

- In the case $i > \gamma_m$, there is a branch $b \in B_i$ from a vertex $v \in V_{i-1}$ to a vertex

$v' \in V_i$ if and only if there exists a codeword $(c_1, c_2, \dots, c_n) \in C$, such that

$$\begin{aligned} (c_i, c_{i+1}, \dots, c_n) &= v \\ (c_{i+1}, \dots, c_n) &= v' \end{aligned} \quad (2.13)$$

the branch label is c_i . In this case, there is exactly one branch beginning at v for each vertex $v \in V_{i-1}$.

- In the case $i = \gamma_m$, there is a branch $b \in B_i$ from a vertex $v \in V_{i-1}$ to a vertex

$v' \in V_i$ if and only if there exists a pair of codeword $c = (c_1, c_2, \dots, c_n)$ and

$c' = (c'_1, c'_2, \dots, c'_n)$ in C such that

$$\begin{aligned} (c_i, c_{i+1}, \dots, c_n) &= v \\ (c'_{i+1}, \dots, c'_n) &= v' \end{aligned} \quad (2.14)$$

and either $c' = c$ or $\beta(c' - c)$ equals the m -th row of G for some constant $\beta \in GF(q)$. The branch label is c'_i , and the number of out-going branch for each $v \in V_{i-1}$ is q .

In this thesis, $RM(32,26)$ code is used as constituent code. According to Section 2.2.1, the generator matrix G of $RM(32,26)$ is constructed, and modified to be a row-reduced echelon form as follows.

$$G_{RM(32,26)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \quad (2.15)$$

As indicated in $G_{RM(32,26)}$, the information bit can be directly obtained in the position of k_i in which the weight is equal to one.

The $RM(32,26)$ trellis diagram is too complex with 638 vertices (states) and 1180 branches to draw in this size paper. However, we can follow the state profile at each time instant to get a simple illustrative drawing. Figure 4.7 in page 61 can be used as a reference.

2.4 Trellis-based MAP algorithm

Since the MAP algorithm for trellis codes was proposed by [14], there has been many papers published concerning the MAP and its sub-optimum variants. Following

[17], we discuss the MAP algorithm in detail in order to provide a guide for later modified algorithm and implementation in this thesis.

It is well known that codeword of a linear binary (N, K) block code can be represented as the paths through a trellis of depth n with at most 2^{N-K} states [17, 21]. Also due to the irregularity of the trellis of linear block code and the complexity of trellis of $RM(32, 26)$ code, one section of a trellis diagram is shown in Figure 2.5 to facilitate the description of the structure.

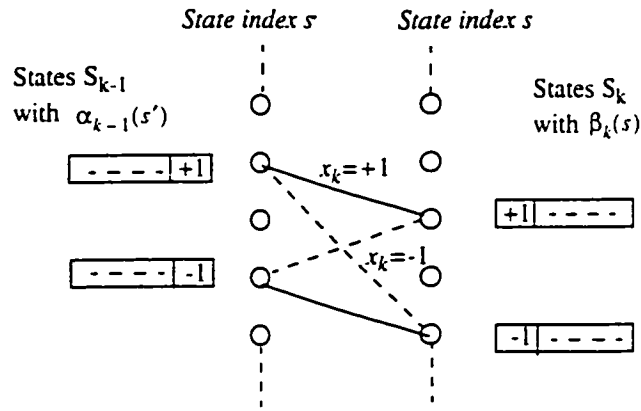


Figure 2.5 One section of trellis structure of block code

Let S_k and S_{k-1} denote the state set at times k and $k-1$; the trellis states at times $k-1$ and k are indexed by s' and s . The coded bit x_k at time k is the label of the branch linking the states at times $k-1$ and k . The soft-output of MAP algorithm conditioned on information sequence \hat{y} is

$$L(\hat{x}_k) = \log \frac{P(x_k = 1 | \hat{y})}{P(x_k = -1 | \hat{y})} = \log \frac{\sum_{\substack{(s', s) \\ x_k = 1}} p(s', s, \hat{y})}{\sum_{\substack{(s', s) \\ x_k = -1}} p(s', s, \hat{y})} \quad (2.16)$$

The sums in the numerator and denominator of $p(s', s, \hat{y})$ are taken over all existing transition from state s' to s labeled with the coded bit $x_k = +1$ and $x_k = -1$, respectively. Under the memoryless channel assumption, the joint probability $p(s', s, \hat{y})$ can be written as

$$\begin{aligned} p(s', s, \hat{y}) &= p(s', \hat{y}_{j < k}) \cdot p(s, \hat{y}_k | s') \cdot p(\hat{y}_{j > k} | s) \\ &= p(s', \hat{y}_{j < k}) \cdot P(s | s') \cdot p(\hat{y}_k | (s', s)) \cdot p(\hat{y}_{j > k} | s) \\ &= \alpha_{k-1}(s') \cdot \gamma_k(s', s) \cdot \beta_k(s) \end{aligned} \quad (2.17)$$

Here $\hat{y}_{j < k}$ denotes the part of the received sequence from bit 0 up to bit $k-1$, and $\hat{y}_{j > k}$ corresponds to the sequence part from bit $k+1$ up to bit $n-1$. The $\alpha_k(s)$ and $\beta_{k-1}(s')$, namely forward recursion and backward recursion accordingly, of the MAP algorithm are

$$\alpha_k(s) = \sum_{s'} \gamma_k(s', s) \cdot \alpha_{k-1}(s') \quad (2.18)$$

$$\beta_{k-1}(s') = \sum_s \gamma_k(s', s) \cdot \beta_k(s) \quad (2.19)$$

In the above two equations, we assume that at time $k=0$ and $k=n$, $\alpha_0(0) = 1$ and $\beta_n(0) = 1$. The branch transition probabilities $\gamma_k(s', s)$ between s' and s for systematic block code with statistically independent information bits are given

$$\gamma_k(s', s) = P(s | s') \cdot p(y_k | s', s) = p(x_k; y_k) \quad (2.20)$$

with $p(x_k; y_k)$ defined as

$$p(x_k; y_k) = \begin{cases} p(y_k | x_k) \cdot P(x_k) & 1 \leq k \leq K \\ p(y_k | x_k) & K+1 \leq k \leq N \end{cases} \quad (2.21)$$

Furthermore, the log-likelihood ratio associated with $p(x_k; y_k)$ can be obtained by defining following equation.

$$L(x_k; y_k) = \begin{cases} L_c \cdot y_k + L(x_k) & 1 \leq k \leq K \\ L_c \cdot y_k & K + 1 \leq k \leq N \end{cases} \quad (2.22)$$

By omitting the common terms for all transition from time $k-1$ to k , the branch transition probabilities can rewritten as

$$\begin{aligned} \gamma_k(s', s) &= e^{(L(x_k; y_k) \cdot x_k / 2)} \\ &= \begin{cases} e^{[L_c \cdot y_k + L(x_k)] \cdot x_k / 2} & 1 \leq k \leq K \\ e^{L_c \cdot y_k \cdot x_k / 2} & K + 1 \leq k \leq N \end{cases} \end{aligned} \quad (2.23)$$

Finally, the soft-output of the trellis-based MAP algorithm for block code can be written as

$$\begin{aligned} L(\hat{x}_k) &= L_c \cdot y_k + L(x_k) + \log \frac{\sum_{(S', S)} \alpha_{k-1}(s') \cdot \beta_k(s)}{\sum_{\substack{(S', S) \\ x_k = -1}} \alpha_{k-1}(s') \cdot \beta_k(s)} \\ &= L_c \cdot y_k + L(x_k) + L_e(\hat{x}_k) \end{aligned} \quad (2.24)$$

The above Equation is the foundation of our design. We also look into the complexity of the computing process. Because it is computation-intensive, it is impractical to implement it using current techniques. Therefore, the simplified sub-optimal algorithm, called Max-Log-MAP, and its implementation are derived from the above Log-MAP algorithm.

2.5 Summary

On the basis of linear block turbo code, the Reed-Muller turbo code used in this thesis was discussed in detail. The construction of minimal trellis of RM code and trellis-based MAP decoding algorithm were presented. The PCCC turbo code scheme used in this thesis and iterative decoding approach as well as SISO decoder were explained in detail. Taking complexity of MAP algorithm into account, we will discuss the alternative decoding algorithms suitable for implementation in next chapter.

Chapter 3

Algorithms for implementation

3.1 Introduction

In turbo decoding, two kinds of algorithms have been commonly used, the SOVA and the MAP algorithm[19].

For estimating the states or the outputs of a Markov process observed in a memoryless channel, the symbol-by-symbol maximum a posteriori (MAP) algorithm is optimal. However, MAP algorithm is likely to be considered too complex to implement in practice; basically because MAP decoding involves the non-linear function, and it requires a large number of memories and operations for exponentiation and multiplication. Log-MAP algorithm can be used to convert the multiplication into addition and to avoid exponential function, Log-MAP is equivalent to MAP and has the same performance as the MAP algorithm. The Log-MAP can be further simplified using Max-Log-MAP algorithm, which has been derived for simplifying the representation of probabilities and for reducing the computational complexity. It is sub-optimal compared to MAP but easy to implement without significant performance degradation [23-26]. In the additive white Gaussian noise (AWGN) channels, the Log-MAP is about 0.5 dB better than SOVA at low SNR region[65]. In this Chapter, we first discuss Max-Log-MAP and Log-MAP algorithms fol-

lowing [15,16,24], then a modified version of Max-Log-MAP with look-up table correction used in this thesis will be given.

3.2 Max-Log-MAP algorithm

In order to simplify the computation of Equation(2.24), to avoid the logarithmic operation and multiplication in the numerator and denominator, logarithmic function of forward recursion $\alpha_k(s)$, backward recursion $\beta_{k-1}(s')$ and transition probabilities $\gamma_k(s', s)$ are used.

First we give the notation and definitions used in following Equations: subscript k refers to the time instant k ; s' and s represent state or node at time instant $k - 1$ and k ; S' and S represent the state set at time instant $k - 1$ and k ; K and N are turbo code information bit lengths and coded word lengths, respectively.

From Equation(2.23) we obtain

$$\log \gamma_k(s', s) = \frac{L(x_k, y_k) \cdot x_k}{2} = \begin{cases} \frac{[L_c \cdot y_k + L(x_k)] \cdot x_k}{2} & 1 \leq k \leq K \\ \frac{L_c \cdot y_k \cdot x_k}{2} & K + 1 \leq k \leq N \end{cases} \quad (3.1)$$

In the logarithmic domain, the following approximation is used to simplify the computation of $\alpha_k(s)$ and $\beta_{k-1}(s')$. That is

$$\log(e^{\delta_1} + e^{\delta_2} + \dots + e^{\delta_n}) \approx \text{Max}_{i \in \{1, 2, \dots, n\}} \delta_i \quad (3.2)$$

where $\text{Max}_{i \in \{1, 2, \dots, n\}} \delta_i$ can be calculated by successively computing (n-1) maximum function over only two values.

Referring to Equation (3.2), we can obtain the logarithm of forward recursion

$$\begin{aligned}
 \log \alpha_k(s) &= \log \sum_{s'} \gamma_k(s', s) \cdot \alpha_{k-1}(s') \\
 &= \text{Max}_{s'} [\log \gamma_k(s', s) + \log \alpha_{k-1}(s')] \\
 &= \text{Max}_{s'} \left[\frac{L(x_k, y_k) \cdot x_k}{2} + \log \alpha_{k-1}(s') \right]
 \end{aligned} \tag{3.3}$$

with the initial condition $\log \alpha_0(0) = 0$ at time $k = 0$.

Similarly, the logarithm of backward recursion is obtained as follow

$$\begin{aligned}
 \log \beta_{k-1}(s') &= \sum_s \gamma_k(s', s) \cdot \beta_k(s) \\
 &= \text{Max}_s [\log \gamma_k(s', s) + \log \beta_k(s)] \\
 &= \text{Max}_s \left[\frac{L(x_k, y_k) \cdot x_k}{2} + \log \beta_k(s) \right]
 \end{aligned} \tag{3.4}$$

with the initial condition $\log \beta_n(0) = 0$ at time $k = n$.

For the extrinsic information output, the approximate form is given as:

$$\begin{aligned}
 L_e(x_k) &= \log \frac{\sum_{(s', s)} \alpha_{k-1}(s') \cdot \beta_k(s)}{\sum_{(s', s)} \alpha_{k-1}(s') \cdot \beta_k(s)} \\
 &= \text{Max}_{(s', s)} [\log \alpha_{k-1}(s') + \log \beta_k(s)] \\
 &\quad - \text{Max}_{(s', s)} [\log \alpha_{k-1}(s') + \log \beta_k(s)]
 \end{aligned} \tag{3.5}$$

Consequently, we obtain the logarithm of soft-output of MAP algorithm in the approximate form as follows:

$$\begin{aligned}
L(\hat{x}_k) &= L_c \cdot y_k + L(x_k) + L_e(\hat{x}_k) \\
&= L_c \cdot y_k + L(x_k) + \log \frac{\sum_{(S', S)} \alpha_{k-1}(s') \cdot \beta_k(s)}{\sum_{\substack{(S', S) \\ x_k = -1}} \alpha_{k-1}(s') \cdot \beta_k(s)} \\
&= L_c \cdot y_k + L(x_k) + \log \sum_{\substack{(S', S) \\ x_k = 1}} \alpha_{k-1}(s') \cdot \beta_k(s) - \log \sum_{\substack{(S', S) \\ x_k = -1}} \alpha_{k-1}(s') \cdot \beta_k(s) \quad (3.6) \\
&= L_c \cdot y_k + L(x_k) + \text{Max}_{\substack{(S', S) \\ x_k = 1}} [\log \alpha_{k-1}(s') + \log \beta_k(s)] \\
&\quad - \text{Max}_{\substack{(S', S) \\ x_k = -1}} [\log \alpha_{k-1}(s') + \log \beta_k(s)]
\end{aligned}$$

Here, we have changed the exponentiation and multiplication into simple addition and comparison operations. This makes the hardware implementation easy and feasible.

3.3 Max-Log-MAP algorithm with correction

As the approximation in Equation(3.2) is used for the computation of forward/backward recursions and log-likelihood values of MAP, the performance of Max-Log-MAP is sub-optimal and yields an inferior soft-output than MAP algorithm. It can be improved by using the Jacobian algorithm.

$$\begin{aligned}
\log(e^{\delta_1} + e^{\delta_2}) &= \text{Max}(\delta_1, \delta_2) + \log(1 + e^{-|\delta_2 - \delta_1|}) \\
&= \text{Max}(\delta_1, \delta_2) + f_c(|\delta_2 - \delta_1|)
\end{aligned} \quad (3.7)$$

where $f_c(|\delta_2 - \delta_1|)$ is a correction function which can be implemented by a look-up-table. So the approximation Equation(3.6) can be computed exactly by recursively using Equation(3.7) as given below.

$$\text{Assume } \Phi = e^{\delta_1} + \dots + e^{\delta_{n-1}} = e^{\delta},$$

$$\begin{aligned}
 \log(e^{\delta_1} + e^{\delta_2} + \dots + e^{\delta_n}) &= \log(\Phi + e^{\delta_n}) \\
 &= \text{Max}(\log \Phi, \delta_n) + f_c(|\log \Phi - \delta_n|) \\
 &= \text{Max}(\delta, \delta_n) + f_c(|\delta - \delta_n|)
 \end{aligned} \tag{3.8}$$

By applying Equation(3.7) to the calculation of $\alpha_k(s)$ and $\beta_{k-1}(s')$ as well as $L_e(\hat{x}_k)$, the performance of Max-Log-MAP approaches that of MAP.

However, calculating the $f_c(|\delta - \delta_n|)$ at each step sacrifices the low complexity of Max-Log-MAP. Therefore, the $f_c(|\delta - \delta_n|)$ is normally stored in pre-computed table. In [14,15], 8 compensation values for $|\delta - \delta_n|$ ranging between 0 and 5 is used.

In this thesis, according to our software simulation, 5 compensation values is used for $|\delta - \delta_n|$ ranging between 0 and 2. Considering the difference $|\delta - \delta_n|$ and the characteristics of logarithmic function, we have the following compensation range

$$0 < [f_c(|\delta_2 - \delta_1|) = \log(1 + e^{-|\delta_2 - \delta_1|})] < 0.7$$

Because of the non-linear nature of the exponential function, the compensation interval is divided unevenly. Simulation results of Figure 3.2 show that the performance of the modified Max-Log-MAP is between Log-MAP and Max-Log-MAP at lower SNR region, and approaches that of Log-MAP after 3 dB. Performance degradation 0.2 dB from MAP is accepted.

Figure 3.1 illustrates the Log-MAP decoding flow chart diagram in our software/hardware simulation.

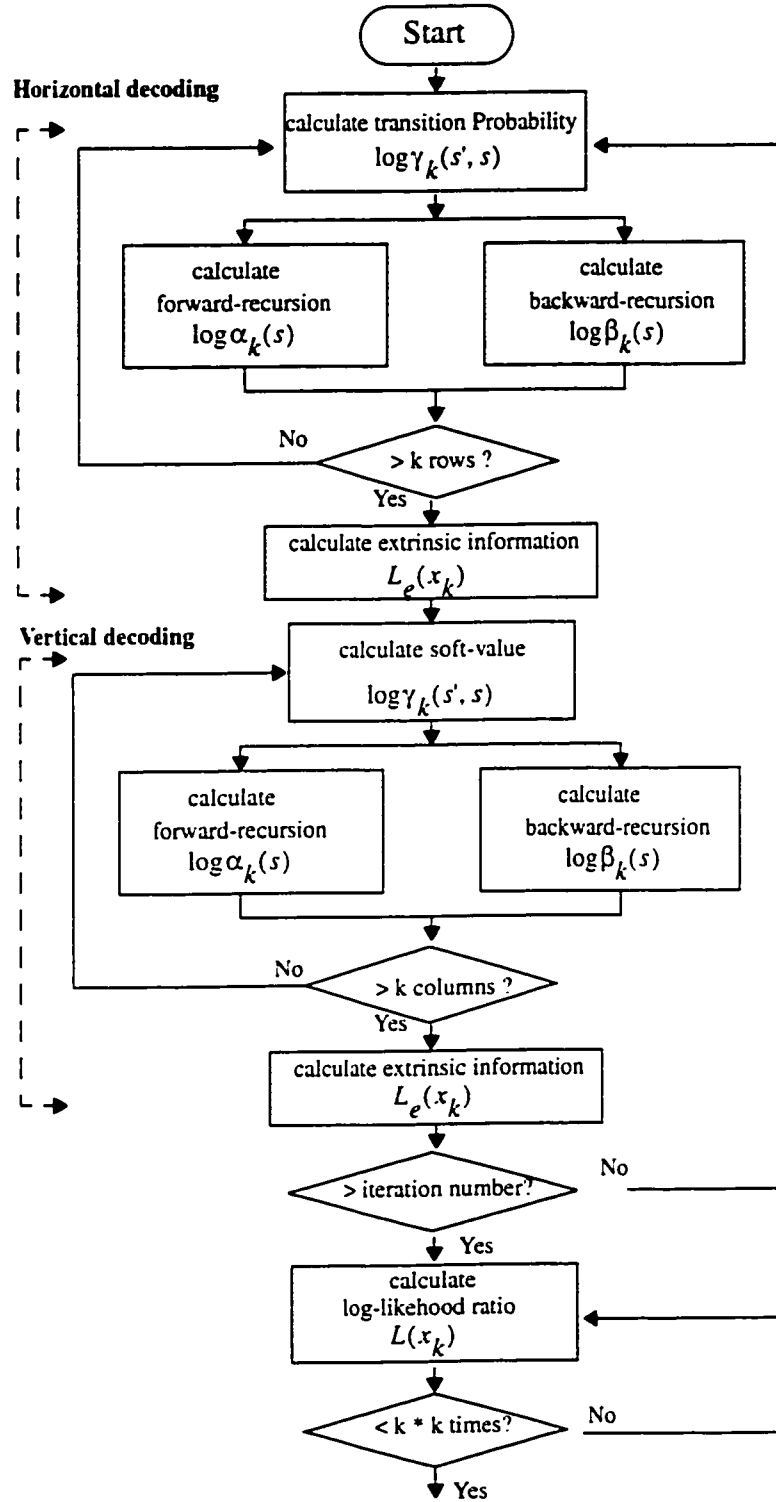


Figure 3.1 Log-MAP decoding flow chart diagram

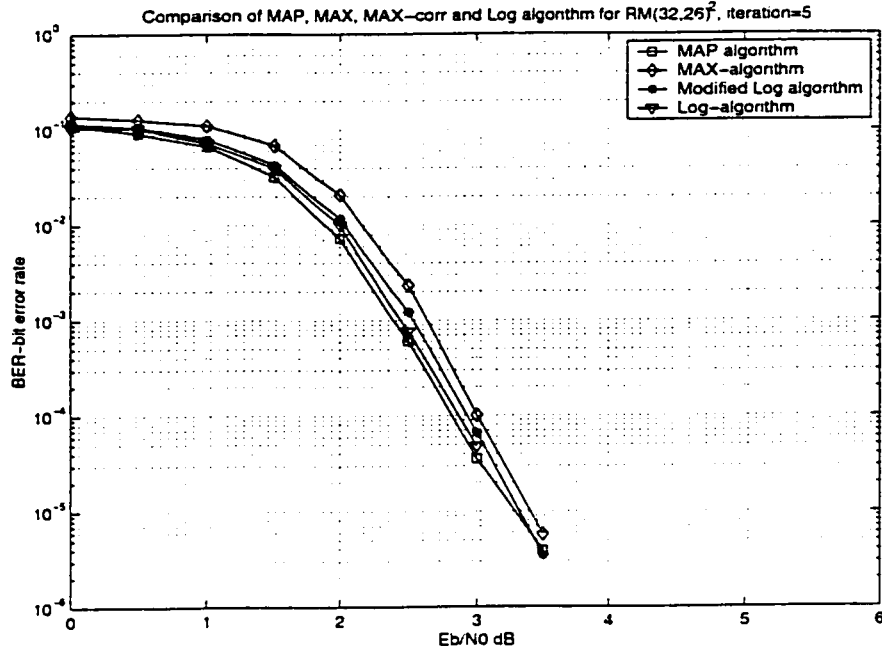


Figure 3.2 Comparison of different algorithms on performance for iteration=5, $RM(32, 26)^2$ code

3.4 Arithmetic units

In this section, we further analyze the relevant details of the calculation of branch transition probabilities, forward/backward recursion. Some pre-processing operations also are given to be the basis for successive metric computation.

3.4.1 Pre-processing model

From Equation 3.1 we can see that channel reliability value L_c is needed for calculation of $\log \gamma_k(s', s)$. For AWGN channel, $L_c = 4\alpha E_s/N_0$. It is related to channel fading amplitude. In two-dimensional code $RM(n, k)^2$, the total code rate is given by Equation 3.9.

$$R = \frac{k^2}{n^2 - (n - k)^2} \quad (3.9)$$

For AWGN channel, $\alpha = 1$. Our design uses two $RM(32, 26)$ as component code so that $n = 32$ and $k = 26$. Corresponding to different SNR value ranging from 5 to 0.5 dB steps of 0.5 dB, we obtain different L_c values according to Equation 3.10.

$$L_c = 4 \cdot R \cdot 10^{\frac{E_b/N_0}{10}} \quad (3.10)$$

The other parameter needed for calculation of Equation 3.1 is $L_c \cdot y_k$. When the decoder starts its operation, it receives the channel values that are output from the match filter. For one transmitted block, all values of coded bits are fixed during decoding stage, no matter how many iterations are used and in horizontal or vertical decoding stage. Therefore, one multiplier is adopted at the interface buffers, and result $L_c \cdot y_k$ are stored in one RAM for the transition probabilities calculation. During horizontal and vertical decoding stage, the RAM is accessed either directly or interleavingly.

3.4.2 Gamma calculation model

Depending on the transmitted bits between two states in trellis diagram, branch transition probabilities have only two possible values for transmitting bit 0 or 1 in a fixed time transition. In BPSK modulation, we have $x_t = 2v_t - 1$, $t = 0, 1, \dots, n - 1$, where x_t is the modulated bit and v_t is the code bit. Consequently, x_t has two values -1 and +1. Refer to Equation 3.1, it's necessary to store only two values for a fixed time instant corre-

sponding to branches labeling +1 or -1. As a result, two RAM memories each with 988 cells are needed in our design.

3.4.3 Alpha/Beta calculation model

According to the property of trellis diagram, there are two branches starting from time $k-1$ and ending at time k corresponding to the transmitted information bit; but only one branch corresponding to the parity check bit. So when applying the Log-MAP algorithm to the calculation of alpha and beta of each state at time instant k , according to the Equation 3.3 and 3.4, there are at most two items in the maximum comparison operation, e.g.

$$\begin{aligned}\log \alpha_k(s) &= \log \sum_{s'} \gamma_k(s', s) \cdot \alpha_{k-1}(s') \\ &= \text{Max}([\log \gamma_k(s_1', s) + \log \alpha_{k-1}(s_1')], [\log \gamma_k(s_2', s) + \log \alpha_{k-1}(s_2')]) \\ &\quad + \log(1 + e^{-|x_\alpha - y_\alpha|})\end{aligned}\quad (3.11)$$

$$\begin{aligned}\log \beta_{k-1}(s') &= \sum_s \gamma_k(s', s) \cdot \beta_k(s) \\ &= \text{Max}([\log \gamma_k(s', s_1) + \log \beta_k(s_1)], [\log \gamma_k(s', s_2) + \log \beta_k(s_2)]) \\ &\quad + \log(1 + e^{-|x_\beta - y_\beta|})\end{aligned}\quad (3.12)$$

where s_1' and s_2' are two independent states at time $k-1$, and s_1 and s_2 at time k , and

$$x_\alpha = [\log \gamma_k(s_1', s) + \log \alpha_{k-1}(s_1')]$$

$$y_\alpha = [\log \gamma_k(s_2', s) + \log \alpha_{k-1}(s_2')]$$

$$x_\beta = [\log \gamma_k(s', s_1) + \log \beta_k(s_1)]$$

$$y_\beta = [\log \gamma_k(s', s_2) + \log \beta_k(s_2)]$$

As we have obtained the state profile of $RM(32, 26)$, the possibility of maximum operation for calculating forward/backward recursion of each state can be analyzed through the state profile present in Section 2.3. For those branches, not only corresponding to information bits but also linking two time instants which have the same state index, and all branches corresponding to parity check bits, i.e. state index at time k is one less than at time $k - 1$, maximum operation should be done in calculation of $\log \alpha_k(s)$ because there are two branches ending at state s from s_1' and s_2' . On the contrary, maximum operation should be done for the calculation of $\log \beta_{k-1}(s')$ for all branches corresponding to information bits.

Following the above analysis of calculation of $\log \alpha_k(s)$ and $\log \beta_{k-1}(s')$, we may use two sets of “add-compare-correct” units which execute the function of the summation and comparison and correction.

3.4.4 Extrinsic information calculation model

Similarly, approximation formula is used in extrinsic information calculation. When calculating the maximum value, we need exhaustively search all state-pairs for each time transition. As shown in Equation 3.5, there are two groups of state transition corresponding to $x_k = 1$ and $x_k = -1$, $k = 1 \dots 32$. This operation is a recursive process. In our implementation, it is divided into two maximum operation which gives two maximum values among the branches labelled 1 and branches labelled -1. According to this recursive requirement, two sets of add-compare-correct unit are used to exhaustively search the final maximum values, and get the extrinsic output from the subtracter.

Following the trellis diagram and state profile, there are totally 1180 branches existing for all state transitions. However, in any next iteration, L_e is not needed as *a priori* probabilities for all parity check bits. Therefore, only L_e values of information bits are calculated in the implementation.

3.4.5 Log-likelihood soft-output calculation model

Following Equation 3.6, after final iteration, the log-likelihood ratio of *a posteriori* probabilities can be implemented by an adder and a comparator. The most significant bit (MSB) of summation determines the estimated transmitted bits. If the MSB is greater than zero, estimated transmitted bit is assumed to be 1; If the MSB is smaller than zero, estimated transmitted bit is assumed to be 0. The amplitude of summation shows the reliability of this judgement.

3.5 Summary

Based on the optimal MAP decoding algorithm, possible sub-optimal decoding algorithms are discussed. Due to a better performance and less complexity of Log-MAP, we selected it for hardware implementation. The software simulation block diagram and performance comparison between optimal and sub-optimal algorithms are presented. Decomposition of algorithms to functional units suitable for implementation also are described. In next chapter, we will discuss further design issues and architecture of implementation related to these function units.

Chapter 4

Design and implementation of RM-BTC

Although turbo codes exhibit an excellent error-correcting performance, the practical implementation faces a lot of challenges relating to computational complexity, power consumption and memory limitations as well as decoding speed etc.. Currently, most turbo codes implementation focus on Viterbi and SOVA decoding algorithms and convolutional code as constituent code, which have regular trellis structure which are relatively easy to be implemented. However, for the block turbo codes and MAP algorithm introduced in previous chapters, the irregular trellis diagram and more computation-intensive algorithm makes it more difficult for practical implementation. Our motivation is to explore and measure the performance of block turbo codes for satellite ATM as implemented by FPGA while minimizing the chip area. The results of this will be a good foundation for further improvement on decoding speed and power consumption.

In this chapter, we first review some works recently published by other researchers, then the hardware platform Virtex-E FPGA used in this design is briefly introduced. Some key issues of design and implementation are discussed in Section 3. Implementation of the turbo encoder and decoder are given in Section 4, 5 and 6. Finally, the chapter summary section concludes the contents involved in this chapter.

4.1 Review of turbo code implementation

Since turbo codes were introduced by C. Berrou in 1993, there has been a lot of papers to discuss its hardware implementation. Because of the highly data dominated MAP algorithm, most of papers pay attention to the sub-optimal algorithm--soft-output Viterbi algorithm (SOVA). Meanwhile, due to the regular trellis diagram of convolutional code, convolutional code is often selected as constituent code of turbo codes. In this section, we trace back some implementations and issues of turbo codes no matter what constituent code and decoding algorithm are used in those implementations[27~30, 34~57].

In [27], a block turbo decoding scheme which allows variable block lengths is presented. In this paper, due to the quantization, a SNR losses of 0.5dB between floating point simulation and hardware measurement under the same coding conditions is presented. The method of SOVA decoding which obtains a maximum throughput of 14 Mbit/s is realized in Altera FPGA. However, no measure of area was reported in this implementation. Through reducing test patterns, methods described in [28,29] obtain a low complexity scheme that use one decoder. It is reported that the coding gain degrades 0.7 dB at iteration 4 with $BCH(64,57,4)^2$ code, where 4-bit quantization contributes 0.1 dB[28].

References[34-42] discuss the quantization and normalization of input data or/and internal signals. Quantization of 3- to 6-bits for input and 5- to 9-bit for internal signals are discussed. Common 4-bit input quantization are accepted and optimal bit-width of internal signals is defined for each parameter. Among many normalization methods, modulo arithmetic and subtracting technique are often used in practice.

References[43-53, 68] discuss some implementation methods both in hardware and software of a turbo decoder with 1/2 or 1/3 rate RSC code or BCH block code. Decod-

ing speed and required area or FPGA utilization vary significantly depending on different realization. Authors of [44] take advantage of sliding window algorithms to reduce storage requirement; Method described in [45] presents a turbo decoder that runs up to 356 Kbps with 4 XC3100A FPGA chip for data processing and control units. Storage memories of intermediate values are implemented by external memories. Yet another method [48, 49] discuss the parallel and pipeline implementation, however, no area utilization is reported even though up to 3.5 Mbps decoding speed is obtained under certain conditions. Authors of [50] give a realization of turbo decoder on DSP chip that achieved at throughput 16.8 Kbps on a 40 MIPS version of Analog Devices ADSP-2181.

Architecture experiments have shown that up to 50-80% of area cost in (application specific) architectures is due to memory unit. So in [55,56] attention is given to memory reduction. Three methods of reducing the whole block processing, over sliding window and on halfway structure are presented.

In order to overcome the complexity of MAP algorithm, [57–67] presents some simplified sub-optimal proposal and VLSI architecture.

4.2 Hardware choices and FPGA

There are some hardware choices for mapping the algorithm to circuits. For the time being, DSPs, ASICs and reconfigurable architecture devices are commonly used to implement FEC (Forward Error Correction) techniques. The general-purpose digital signal processors (DSPs) are characterized by their flexibility and low cost, they are able to offer a wide variety of applications and perform a variety of computations and logic tasks. However, their general flexibility result in inefficient and lack of performance in specific

applications. On the other hand, fully custom design application-specific-integrated-circuits (ASICs) execute the specific task, and offer designer excellent performance including maximum speed, minimum chip area and power consumption, but inflexibility and the demands for a faster time to market in some cases make this alternative unattractive, especially for the research purpose, long development cycle of ASIC is not suitable for prototyping. As a balance, FPGAs are suitable for FEC because they have efficient parallel architectures and are reconfigurable. This parallel reconfigurable architectures suits particularly pipeline design, which corresponds to the parallel architecture of MAP algorithms. Meanwhile, the performance of FPGAs are approaching that of ASICs and FEC cores for FPGA implementation are inexpensive. Therefore, the advantages of field programmable gate array (FPGA) makes it an ideal solution for our design.

Field programmable gate array was introduced in the mid-1980's as a single-purpose technology with no function beyond processing digital logic[32]. Its initial role was to be used as "glue logic" to interconnect more complex system components. Today, this adaptable technology has virtually revolutionized the fields of computation and digital logic due to their increased gate density and computational ability.

According to the reprogrammability, there are mainly three kinds of FPGAs: SRAM-based, antifuse-based and EPROM/EEPROM-based FPGA. Since static random access memory (SRAM) FPGAs are easily reprogrammable through fast in-circuit reconfiguration, they are most ideal medium for prototyping the digital designs and has become more popular in practical applications.

Digital design with reconfigurable architectures is often a daunting task because good designs demand sound design practices as well as an intimate understanding of both

the design tools and the target devices. To optimize the use of programmable logic devices, in the next section, we present the structure and some features of Xilinx Virtex-E family FPGA which is used in our implementation.

4.2.1 Virtex-E FPGA architecture

Evolving from Virtex, the Virtex-E FPGA family delivers a high-speed and high-capacity programmable logic solution by optimizing the new architecture for place-and-route efficiency and exploiting an aggressive 6-layer metal 0.18 μ m CMOS process, which results in smaller dice, faster speed and lower power consumption[33]. The current functionality of SRAM-based FPGAs is configured to logic elements and interconnect resources through the values stored in static memory cells; function change can be achieved by reloading configuration values into SRAM cells. Some main features related to our design are listed below:

- Densities range from 58 Kb to 4 Mb system gates;
- 130 MHz internal performance (four LUT levels);
- Up to 804 singled-ended I/Os or 344 differential I/O pairs for an aggregate bandwidth of > 100 Gb/s;
- True Dual-Port™ BlockRAM capability;
- Up to 832 Kb of synchronous internal block RAM;
- Clock multiply and Divide;
- Dedicated carry logic for high-speed arithmetic;
- Dedicated multiplier support;
- SRAM-based in-System configuration, unlimited re-programmability;

Virtex-E Architecture overview is shown in Figure 4.1. There are three main parts with two major configurable elements: configurable logic blocks (CLBs), input/output blocks (IOBs) and programmable routing matrix (GRM). CLBs are the functional elements for constructing the user logic; IOBs provide the interface between the package pins on the chip and the CLBs signal lines; the GRM provides the interconnection route between CLBs and IOBs.

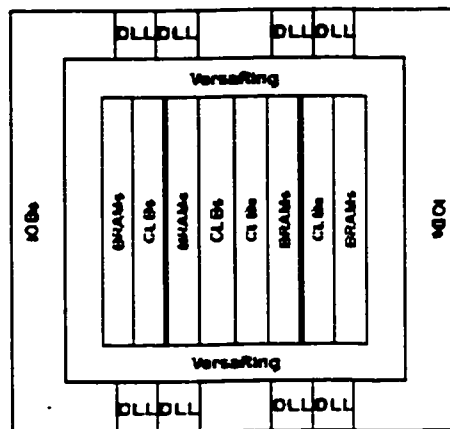


Figure 4.1 Virtex-E architecture overview[33]

The basic building block of the Virtex-E CLB is the logic cell (LC). An LC contains a 4-input function generator, carry logic and a storage element. Each Virtex-E CLB includes four LCs organized in two similar slices as shown in Figure 4.2. The detailed single Virtex-E slice is illustrated in Figure 4.3.

As shown in Figure 4.2, the function generators are implemented as 4-input look-up tables (LUTs). Each LUT also can be used as a 16 x 1-bit synchronous RAM; two LUTs within one slice can create a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16 x 1-bit dual-port synchronous RAM. Through combining the function generator outputs by multiplexers in slice, CLB can implement any 5- or 6-input function, 4:1 or 8:1 multiplexer or selected functions of up to 19 inputs.

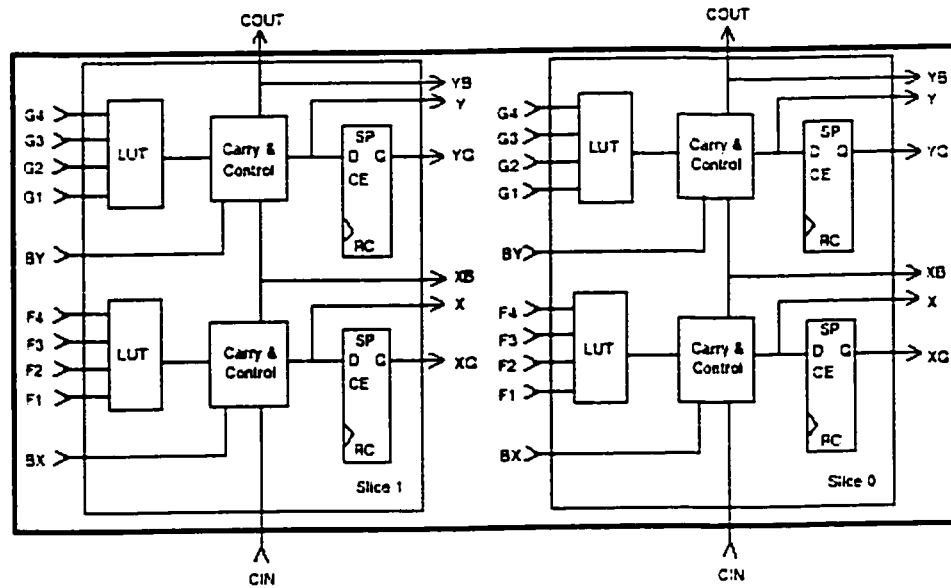


Figure 4.2 A 2-slice Virtex-E CLB[33]

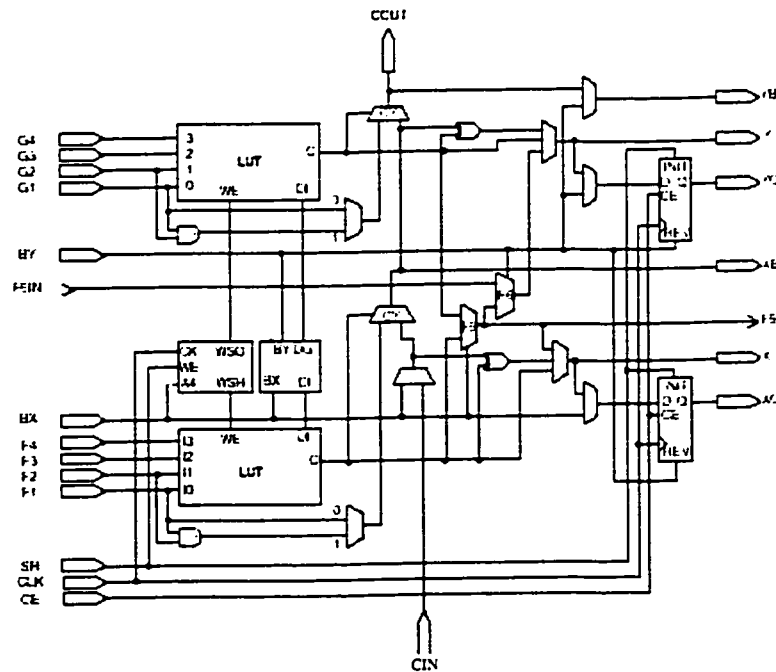


Figure 4.3 A detailed view of Virtex-E slice[33]

Adder and multiplier implementation depend on the FPGA architecture. The Virtex-E CLB supports two separate dedicated and fast carry chains as well as dedicated

AND gate, which provide fast arithmetic capability for high-speed arithmetic functions, and improve the efficiency of multiplier implementation. Another important feature of Virtex-E FPGAs is that it incorporates large block SelectRAM memories. Each of them is fully synchronous dual-port 4096-bit RAM. This feature offers a complement for built-in memories.

In the next section, some basic arithmetic function cores used in our design will be given.

4.2.2 Arithmetic function cores

As mentioned in the last section, FPGA architecture efficiently influence the implementation of adder/subtractor and multiplier. To accelerate and condense arithmetic functions, Virtex-E FPGAs incorporate dedicated carry chains and AND gate for adder and multiplier implementation. The advantages of these carry chains are that they are faster than most other fast-carry schemes and using them eliminates the need to use the FPGA's look-up table. Hence, the availability of these components optimizes both speed and size parameters critical to implementation.

Xilinx Coregen library provides a wide variety of standard functional core which are applied in different fields. In our decoding algorithm, many large bit-width adders/subtractors are used. Meanwhile, because there are a lot of intermediate results that should be stored for recursive calculation, a large amount of memory is also used. In order to benefit from the standard library, adder/subtractor, multiplier and memory (RAM and ROM) cores are employed in the design. For detail information of these cores about interface and timing refer to help sheet of Xilinx Coregen library.

4.3 Design issues

The actual implementation of turbo encoder/decoder involves works from algorithm mapping to circuits. The optimal finite accuracy representation not only obtains the required decoding performance but also reduces the hardware cost in the design. The system and functional module level optimization can simplify the circuit structure, and the data and the resources that is common in decoders hence reduce the memory requirement. Some design issues are discussed in this section and the conclusion will be taken as specification parameters for implementation.

4.3.1 Quantization and finite accuracy

For implementation, the quantization of the variables and their fixed point representation is a necessary step. When mapping an algorithm onto an actual architecture, quantization and fixed point representation have major influence on performance, area and energy consumption. The normal practice is to quantize the received sequence and send these values to decoder. While few levels of quantization minimizes the cost of A/D converter and also the number of bits that are required to represent each quantization level[1], it also introduces some noise to the system. If fixed-point arithmetic is employed, then a degradation in the BER performance of the turbo decoding algorithm is inevitable. Therefore, optimal word lengths of variables must be determined that optimizes the performance and reduces hardware costs.

The quantization of received signals is simulated for different bit-widths by software. For BPSK and an AWGN channel, the received values are spread with a Gaussian distribution around the transmitted symbols $\{-1, 1\}$. More than 99% of the received values

are located in the range of -4 to 4. In the simulation, the continuous signal within the range (V_{low}, V_{high}) is mapped into integer numbers between $(-2^{n-1}, 2^{n-1}-1)$, where n is the number of bits of quantizer[35]. The quantization is realized by dividing the range region into 2^n evenly spaced zones, which are numbered with integer number. The zone width is $\Delta = (V_{high} - V_{low})/2^n$, and the zone boundaries are at $\{-\infty, V_{low} + \Delta/2, \dots, V_{low} + (2m-1)\Delta/2, \dots, V_{high} - 3\Delta/2, \infty\}$, $m = 1, \dots, 2^{n-1}$.

Figure 4.4 shows simulation result for $RM(32, 26)^2$ turbo code. The 4-bit and 5-bit quantization have good performance which is close to infinite accuracy. Simulation in Figure 4.5 shows the results of 4-bit quantization for different channel SNR values. The results are insensitive to choice of SNR in higher SNR value. So 4-bit quantization is accepted in this design as a good trade-off between decoding performance and hardware cost.

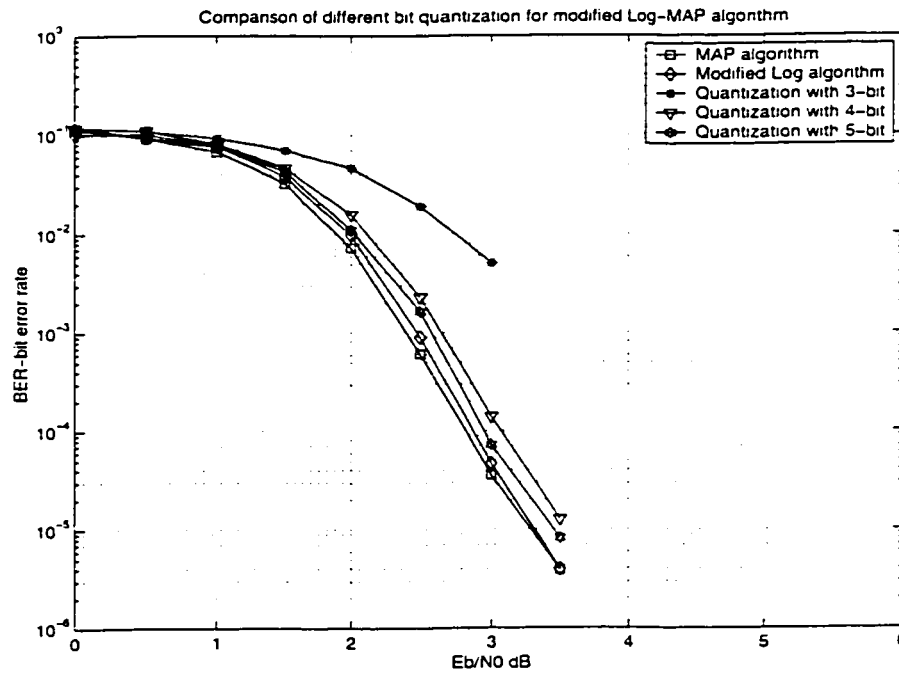


Figure 4.4 Different bit quantization effects on performance for modified Log-MAP algorithm, iteration=5, $RM(32, 26)^2$ code

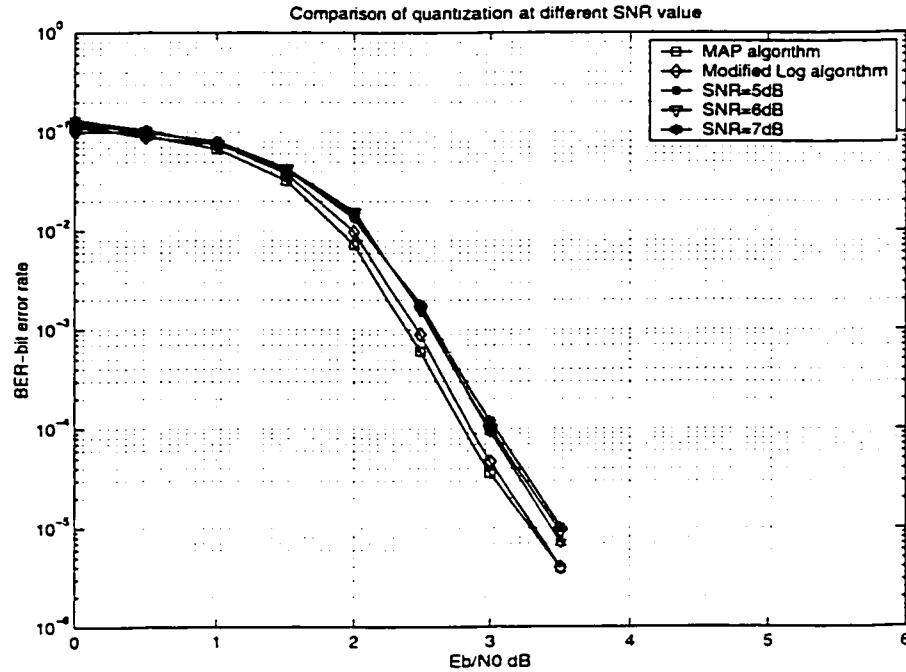


Figure 4.5 4-bit quantization on different SNR values for modified Log-MAP algorithm, iteration=5, $RM(32, 26)^2$ code

For all internal signals, a combination of theoretical analysis and algorithm simulation is used to determine the bit-widths of parameters. When the input is 4-bit, the internal values of forward and backward recursion calculation never exceeded a range that is represented by 17-bit width in the worse case; likewise, the values of branch metric and extrinsic information is within 12-bit width; and the channel reliability values is represented by 8-bit width.

4.3.2 Data- and hardware-sharing

Log-MAP algorithm is a computation-intensive decoding method. Although its recursive search for survival trellis path improves the decoding accuracy, it also generates a great amount of intermediate results that has to be stored for successive computation. If storage memory for each codeword is configured repeatedly, the required memory has an

exponential growth along with the increase of block size. This is unacceptable for implementation of large block size turbo codes. However, an important feature of iterative decoding is that its first and second decoding stage is at different time period even though they do the same operation on a specific data block. These characteristics suggest that we may share some hardware units and calculation results to save chip area and reduce calculation time in the design.

As analyzed in Chapter 3, the channel value for all transition in one block are the same for both decoders; only the order of data sequence is interleaved. So the data-sharing leads to a common-sharing structure of channel value memory in two decoders. Because all channel values are ready after pre-processing units, it is no longer needed to re-calculate them for each coming horizontal and vertical decoding, no matter how many iteration is required. This reduces channel value memory to half of that needed for two SISO structures.

A main design strategy is to employ one SISO decoder to execute two stages of decoding according to inherent property of iterative turbo decoding. The arithmetic function units are reused in horizontal and vertical decoding. Meanwhile, because the calculation of forward and backward recursion is executed one by one codeword, the sizes of internal storage memory for forward and backward recursion calculation hence are determined by the node (state) number of minimal trellis diagram of $RM(32, 26)$. Because the forward and backward recursion values of one codeword are no longer needed for other words, the memory location can therefore be reused for each codeword operation within a specific block. As a result, the memories are shared in different time within and between

horizontal and vertical decoding. This measure significantly decreases the memory size for internal results of forward and backward recursion.

Likewise, extrinsic information of horizontal and vertical stage also uses the same storage memory. Since the extrinsic information of each bit is generated and located in a fixed address of memory, we can access them by interleaving way in both decoding stages.

From above discussion, it shows that how sharing of arithmetic function units and memory units reduce to half the hardware circuits; consequently, chip area is reduced considerably.

4.3.3 Parallelism and pipeline

In the Log-MAP algorithm, many computations are repetitive and parallel. They include the computations of branch transition and forward/backward recursion as well as extrinsic information. From Equation 3.1 to 3.5, we know that for each bit of one codeword, we need to calculate $\log \gamma_k(s', s)$ then $\log \alpha_k(s)$ and $\log \beta_{k-1}(s')$ and finally the $L_e(x_k)$. All computations can be divided into three relatively independent parts in terms of processing time. For the block turbo code, in order to decrease the overhead of decoding one codeword by one, a specific block will be treated as a whole body, and the calculation of branch transition $\log \gamma_k(s', s)$ of each bit in the specific block is done in pipeline. Because one SISO decoder instead of two decoders is used to perform all the iterations including horizontal and vertical decoding, one set of memory for storing intermediate results of forward and backward recursion must be also reused in horizontal and vertical decoding stage. This leads to separate calculation of $\log \alpha_k(s)$ and $\log \beta_{k-1}(s')$ for individual codeword and further pipeline operation within one codeword. Similarly, the opera-

tions on $L_e(x_k)$ is in parallel for branch labelled +1 and -1. Especially for $\log \alpha_k(s)$ and $\log \beta_{k-1}(s')$ calculation, besides the pipeline process for both of them, they are implemented in parallel by following their property. After all branch transition are obtained, $\log \alpha_k(s)$ and $\log \beta_{k-1}(s')$ can be calculated at same time from both directions of trellis, forwardly and backwardly. The benefit of this parallel computation is a reduced decoding time at an acceptable cost of duplicating the recursion calculation unit.

Horizontal and vertical decoding run with the same decoding procedures at different stages with interleaved data block. Except the results of $\log \gamma_k(s', s)$ are the same for both of them, the other intermediate results are restricted in one stage and no longer needed for another stage. As a result, the hardware resource can be reused and, consequently, horizontal and vertical decoding are multiplexed on the same physical hardware circuits. This results in a factor of 2 decrease in turbo decoder circuits at a negligible cost of increasing the size of the control unit. Because of reduction of decoding circuits, the used memories are also reduced to half as compared to the traditional approach. The details of memory configuration is given in Section 4.6.

In conclusion, module level optimization assigns the algorithm into three parts which will be executed in parallel and pipeline. Meantime, system level optimization replaces two SISO structure with one SISO scheme. Both measures not only increase the decoding speed but also reduce the chip area for circuits layout.

4.3.4 Iteration stop criterion

Feedback loop is an important feature of turbo decoding algorithms. In turbo decoders, the first decoder uses only a fraction of the available redundant information.

Through the feedback loop with interleaver, most of redundant information supplied by second decoder can be used to improve the performance of turbo decoding[11]. This is the principle of iterative decoding and the term turbo-code is given for this feedback decoder.

Although increasing the number of iterations has a considerable effect on the performance, the coding gain has a diminishing returns along with the increase of the number of iteration. Most of the gain in iterative decoding is achieved by the first several iterations, and there is no more improvement of coding gain for the later successive iterations. The reason for this saturation limit is that the extrinsic information L-values will become more and more correlated due to same information being used indirectly. In order to avoid unnecessary computations and reduce the decoding delay, normally there are two ways to terminate the iterative decoding process, cross entropy criterion stop and systematic cyclic redundancy checking. This two ways effectively stop the iteration process with very little performance degradation[17,18].

Following Usa's simulation[5], we have the performance with different iterations on an AWGN channel of $RM(32, 26)^2$ turbo code, which will be implemented in the thesis. Simulation results show that for this code length, the coding gain will be saturated at the 5th iterations and no more improvement after 5 iterations is achieved. Consequently, we use 5 iterations as a reference of iteration input design with negligible performance degradation.

4.3.5 (De)-Interleaving technique

As discussed in Section 2.2.3, both interleaver size and structure effectively influence the code performance. When component code was determined, one may choose dif-

ferent interleaver size and a particular interleaver structure which might perform better. In our design, the block interleaver is selected and the size is determined to be 26×26 due to the property of component $RM(32,26)$ code.

According to iterative decoding principle, the extrinsic information of one decoder module is (de)interleaved and is used as an *a priori* values in the next decoder module. Normally, there are two buffer memories for storage of these value in traditional two SISO decoder approach.

In our interleaving technique, because only one SISO decoder is used to decode horizontally and vertically, only one memory for extrinsic information is used. This memory acts as interleaver and deinterleaver at different times through interleaved access mode. For instance, during horizontal decoding, memory is read out in normal order as input signal sequence, and extrinsic information is written back at the same order to correspond to original position of memory; during vertical decoding, extrinsic information is read out in interleaved order, and also written back at interleaved order. This accessing method always ensure that extrinsic information of each information bit is stored exactly in a fixed cell of memory. The mapping relationship between bit and memory address is fixed for known $RM(32, 26)^2$ turbo code which uses block interleaver, so look-up-table addressing of the internal memory is used in the design. This interleaving technique reduces the buffer memory and simplify the circuits. Because the fixed mapping relationship enables implementation of address generator of extrinsic information memory by look-up table, we can efficiently take advantage of BlockRAM of FPGA. Meantime, the combinational circuit of address generator is avoided.

4.4 Turbo encoder implementation

Our turbo encoder uses parallel turbo encoder scheme as shown in Figure 2.1(a). Since $RM(32, 26)$ code is a systematic-like code, following this characteristic, the information bits are apparent and only parity bits need to be calculated, and encoder 1 and 2 can run on the same information block at the same time based on our interleaving technique. The transmission order of one coded block output from encoder is set as information bits followed by parity bits 1 then parity bits 2.

In this section, details of turbo encoder are given for I/O interface, finite state machine and data path.

4.4.1 Interface input/output

The preliminary specification of turbo encoder is shown in Figure 4.6, which defines the pin names and I/O signals. Table 4.1 describes the pin name and bit-width of each signal.

Even though we have defined the output as “codeword”, it actually is not an individual codeword. It should be understood as one coded block sequence in form of information bits + parity bits 1 + parity bits 2, where information bits is completely the same as the input information sequence.

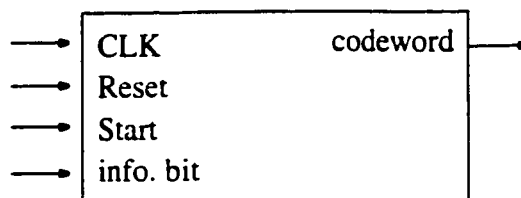


Figure 4.6 I/O Interface of turbo encoder

Table 4.1: Input/Output interface definition of turbo encoder

Pin name	I/O	Bits	Description
CLK	input	1	Information bit encoding clock
Reset	input	1	Encoder reset signal
Start	input	1	Encoder start receiving and encoding
infobit	input	1	information bits be encoded
codeword	output	1	codeword after encoding

4.4.2 Generator matrix and trellis

From Equation 2.15, we have systematic-like generator matrix $G_{RM(32, 26)}$, which clearly indicates the positions of information bits. Although the generator matrix is quite large for $RM(32, 26)$ code, we only need to calculate parity bits of each codeword to generate the complete codeword. This not only saves calculation time but also limits the required memory size. As shown in Table 4.2, only 6 cells of generator memory are needed to store the generator vectors corresponding to parity bits. During encoding process, information bits of each horizontal and vertical codeword separately multiply with every vector of generator memory, and then modulo-2 addition is used to obtain the respective parity bit.

Corresponding to generator matrix $G_{RM(32, 26)}$, the butterfly-like trellis diagram of $RM(32, 26)$ code is shown in Figure 4.7. Because its complexity, which contains 638

nodes and 1180 branches connecting node pair, we just give here the simplified illustrative diagram.

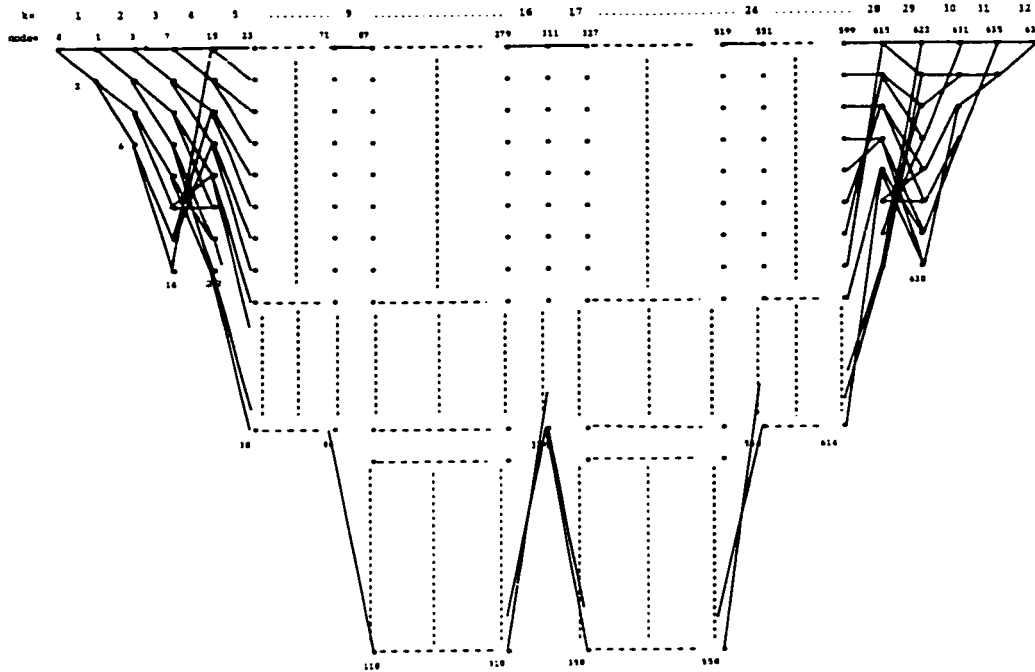


Figure 4.7 The illustrative trellis diagram of RM(32,26) code

Table 4.2: Generator memory address and cell content

Cell No.	Memory address	Cell content (Hex)
1	000	0007FFF
2	001	03F80FF
3	010	1C78F0F
4	011	2D9B333
5	100	36AD555
6	101	3B4E996

4.4.3 Finite state machine

The finite state machine (FSM) of turbo encoder is shown in Figure 4.8. Starting from state “idle”, when signal $\text{start} = '1'$, information sequence shift into serial-in-serial-output register; when one information block is ready, state change to “encode” to encode the codeword one by one. After 26 codewords are finished, the generated parity bits stored in shift register 2 and 3 will shift out in state “shift-out”. Then state returns to “shift” to receive new information block and new encoding process starts.

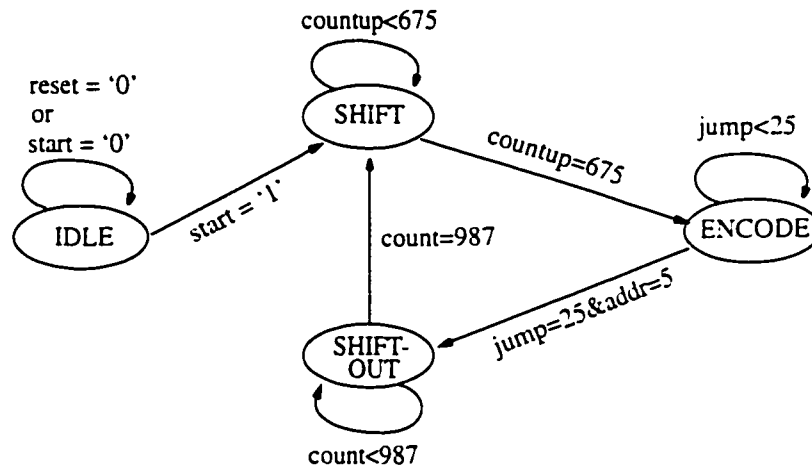


Figure 4.8 Finite state machine of turbo encoder

4.4.4 Encoder data path

For parallel turbo encoder, two elementary encoders are connected in parallel by an interleaver. They both work on the information bits and its interleaved version respectively. As discussed in Section 4.3.2, only parity bits need be calculated for both encoders and information bits can be passed through directly. For block interleaver, the input sequence is written into the matrix row-wise and read out column-wise. In order to interleavingly access 26 information bits of one code word at same time, the input buffer is

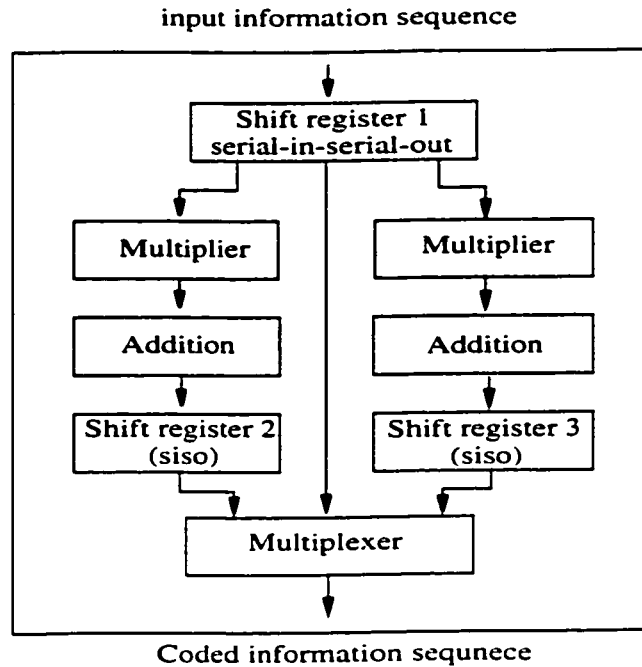


Figure 4.9 Datapath of turbo encoder

made of serial-in-serial-out shift register. When one block information bits is ready in shifter register 1, horizontal and vertical elementary encoder start to encode in both direction at same time. The encoding process is executed by boolean multiplier and modulo-2 adder. After the calculated parity bits of each codeword are ready, they shifted into shift register 2 and 3 separately for horizontal and vertical codeword. All shift registers are first-in-first-out (FIFO) and encoding starts from first-in sequence. When all parity bits for all horizontal and vertical code words are obtained, the total coded sequence of one block information are located in shift register 1, 2 and 3, which are accordingly the information part, horizontal parity bits part and vertical parity bits part. Finally, the three parts mentioned shift out through the multiplexer as output of turbo encoder one by one.

The size of shift register 1 is determined by code information length of one block.

For $RM(32, 26)^2$ turbo code, this is 26×26 bits. The size of shift register 2 and 3 is the same and equal to 26×6 bits.

4.5 Turbo decoder implementation

As a general rule, the more powerful a code, the more difficult the decoder. Likewise, turbo decoder design is more complex than turbo encoder. In this section, the component function units and finite state machine of turbo decoder are given in detail. Following the property of modified Log-MAP algorithm, the total data path structure is shown in Figure 4.10, which is divided into four independent calculation modules connected by intermediate memories. The individual module is discussed in the order of data flow.

4.5.1 Interface input/output

All input and output ports of turbo decoder are defined in Table 4.3 and illustrated in Figure 4.11. According to the specification, turbo decoder will take 4-bit quantized output from demodulator as a channel input. The channel SNR will be treated as presetting values which could be chosen in compliance with the actual estimated result. In this design, the presetting SNR ranges from 0.5 dB to 5 dB in steps of 0.5 dB. As a research model, the iteration number of decoding also can be selected from input port. Because one SISO decoder is used to perform both stage decoding in all iterations, the input must be odd number between 1 to 15 corresponding to iteration 1 to 8. This range is selected based

on the iteration stop criterion, which has shown that the coding gain is saturated at 5th iteration for $RM(32, 26)^2$ turbo code.

Since the transmitted sequence in the channel is continuous, two buffer memories are used before the decoder module. Buffer memories work on different clock signal from decoder module for a smooth switching between them. CLK0 for buffer memories is the same with channel transmission rate, and CLK1 for decoding procedure is determined by decoding speed.

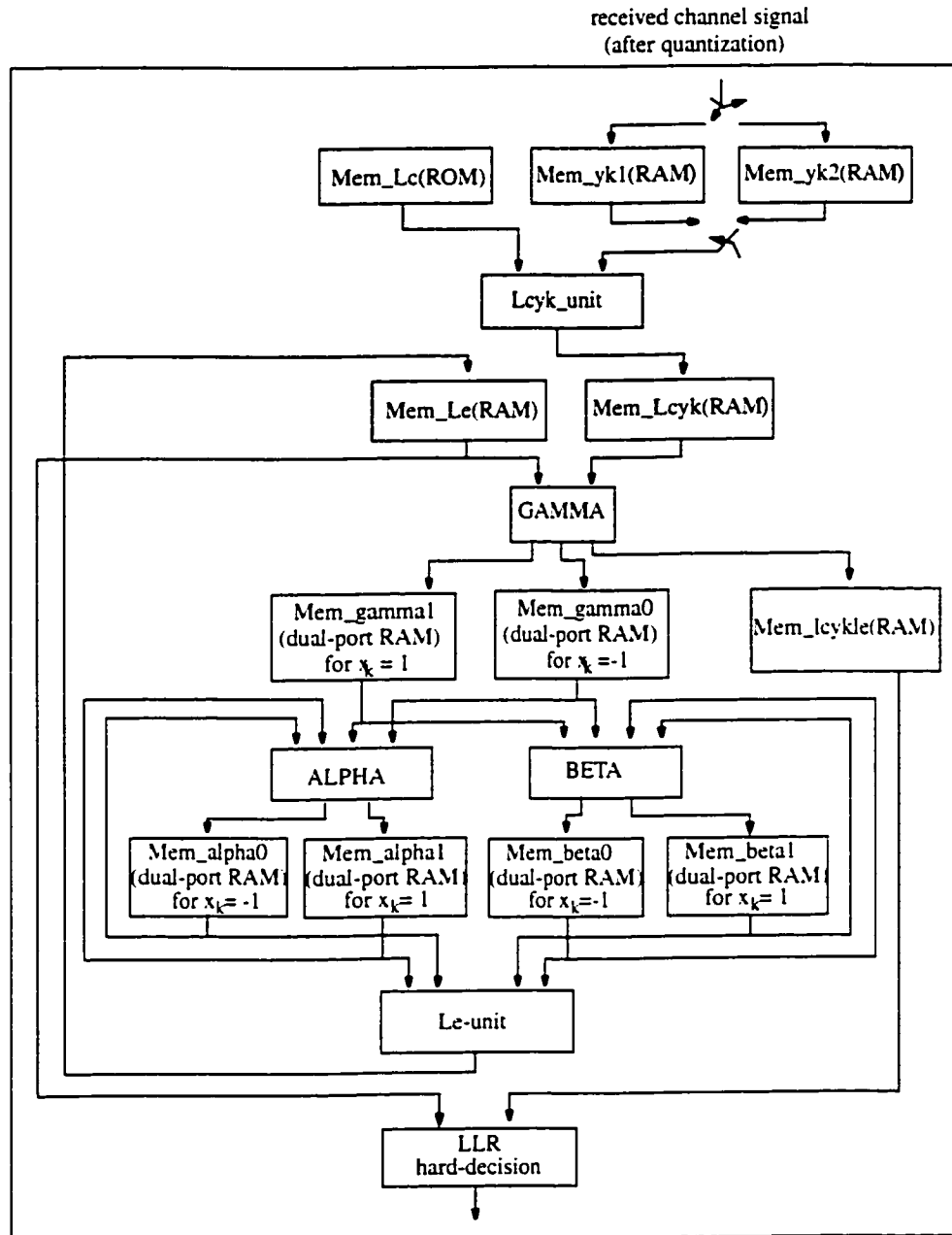


Figure 4.10 Datapath architecture of Log-MAP turbo decoder

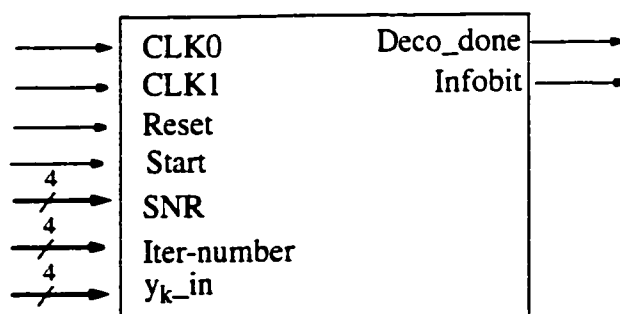


Figure 4.11 I/O Interface of turbo decoder

Table 4.3: Input/Output interface definition of turbo decoder

Pin name	I/O	Bits	Description
CLK0	input	1	Data transmission channel clock
CLK1	input	1	Decoder data decoding clock
Reset	input	1	Decoder reset signal
Start	input	1	Decoder start receiving and decoding
iter-number	input	4	Presetting iteration number of decoding (1, 3, 5, ..., 15) 0001: one time iteration; 0011: two time iteration;; 1111: eight time iteration (Max. iteration number)
SNR	input	4	Presetting signal-to-noise values (E_b/N_0) 1010 : 5.0 dB, 1001 : 4.5 dB, 1000 : 4.0 dB, 0111 : 3.5 dB, 0110 : 3.0 dB, 0101 : 2.5 dB, 0100 : 2.0 dB, 0011 : 1.5 dB, 0010 : 1.0 dB, 0001 : 0.5 dB
y_k -in	input	4	Decoder quantized input from demodulator
Deco-done	output	1	One block decoding finish signal
infobit	output	1	estimated transmitted information bit

4.5.2 Finite state machine

The finite state machine of turbo decoder is an important function unit in the design. The advantage of present design is the optimal FSM approach. Because the decoding operation on each codeword is the same, and received channel values of one codeword or block are fixed for all iterations, we can calculate all branch metrics at one state. Further, the soft-output log-likelihood ratio of SISO also can be computed in one state after all extrinsic information of one block are ready. Consequently, the state assignment is shown in Figure 4.12: starting from state “IDLE”, the pre-processing of channel values and computation of branch metrics of one transmitted packages (block) are finished in state “GAMMA”; in the following state “ALPHABETA” and “LEUNIT”, the forward/backward recursion values and extrinsic information of one codeword are computed one codeword by one codeword; when all horizontal and vertical extrinsic information outputs are ready through 26 “ALPHABETA-to-LEUNIT-to-ALPHABETA” recycle, the final soft-output log-likelihood ratios of one block are obtained in state “LLR”, in which the hard decision of estimated transmitted sequence is made.

In state “GAMMA”, treating one block as a whole body saves the overhead of switching over codeword one by one. Because the forward/backward recursion $\log \alpha_k(s)$ and $\log \beta_{k-1}(s')$ of one codeword are no longer needed for others, their storage memories can be reused by others after computation of extrinsic information $L_e(x_k)$ has finished. Further more, since the computation of extrinsic information $L_e(x_k)$ is closely connected to forward/backward recursion values, computing one codeword by one codeword significantly saves a great amount of internal memory of intermediate results with a

minor cost of control circuit. That's the reason why we have combined the state "ALPHA-BETA" and "LEUNIT" into one loop.

Due to the current extrinsic information output that is used as a priori value in the next decoding process, and depending on the presetting iteration number, state "LEUNIT" will change to "GAMMA" for computation of a new $\log \gamma_k(s', s)$. This leads to a new decoding stage namely horizontal or vertical decoding. After presetting iteration number, hard decisions are made based on the horizontal and vertical extrinsic information and return to state "IDLE" to wait for the next transmitted package whenever it is ready.

Within all states, the computation is designed to execute in pipeline manner to increase the throughput. In addition, the address generator of the address memory is combined into the FSM unit.

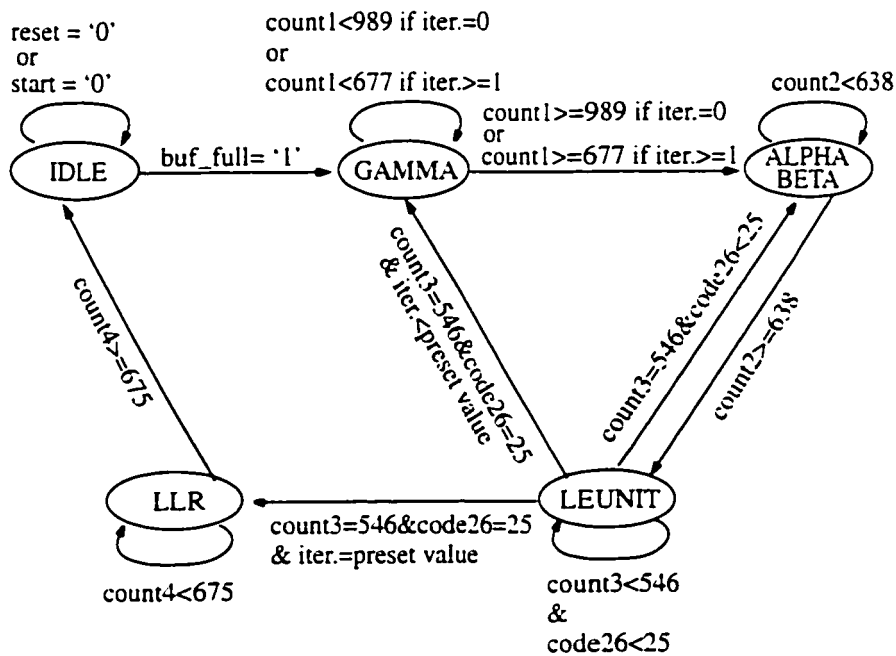


Figure 4.12 Finite state machine of turbo decoder

4.5.3 Pre-processing component

The pre-processing unit is realized by a 8-bit \times 4-bit multiplier in state “GAMMA”. According to the chosen E_b/N_0 input, the corresponding channel reliability value L_c multiplied by the quantized input y_k at first horizontal decoding stage and the results is stored in the intermediate memory Mem-Lcyk for later computation. In the following decoding stages, we no longer calculate these products and just directly read them out from memory for computation of $\log \gamma_k(s', s)$. The L_c values corresponding to different SNR are listed in Table 4.4 and stored in read only memory Mem-Lc. Column “ L_c in binary” gives the binary representation with 3 bits of fraction. When SNR of channel is determined, decoder will choose the respective Lc value.

Table 4.4: Lc values for E_b/N_0 at rate $R=0.684$

E_b/N_0 (dB)	L_c	L_c in binary
5.0	8.652	01000101
4.5	7.711	00111110
4.0	6.873	00110111
3.5	6.125	00110001
3.0	5.459	00101100
2.5	4.865	00100111
2.0	4.336	00100011
1.5	3.865	00011111
1.0	3.444	00011100
0.5	3.070	00011001

4.5.4 Gamma unit

Figure 4.13 is the data path of branch metric computation. The binary arithmetic function addition, multiplication and division are implemented by the adder, two's complementer and shift operation accordingly. Gamma unit reads in both the extrinsic information of previous decoding stage and channel signal from internal memories Mem-Le and Mem-Lcyk, and outputs two type of $\log \gamma_k(s', s)$ from state S' to S with branch labeling +1 and -1. Meantime, the intermediate results, the sum from the adder, is output directly to the decoding stage and is used in the computation of final log-likelihood ratio.

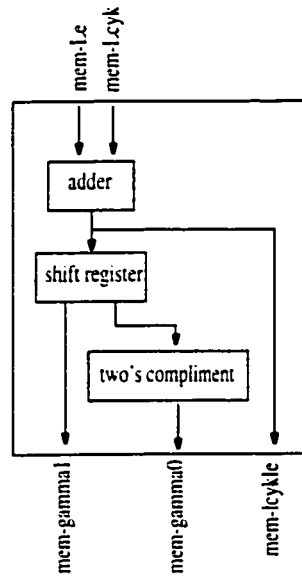


Figure 4.13 Datapath of branch metric calculation

4.5.5. Alpha/Beta unit

In the computation of forward and backward recursion, the operation is the same for both of them except that each starts from both terminals of the trellis diagram. Therefore, the data path for them is duplicated and they only have different data input as shown

in Figure 4.14. Following arithmetic model given in Chapter 3, all $\log \alpha_k(s)$ and $\log \beta_{k-1}(s')$ for each node (state) are recursively calculated from first node at the starting time instant to the last one. This computation is exhaustive for all branches finally converging to a node.

From the property of trellis diagram for one codeword, there are at most two branches ending at one node no matter what the transition is caused by the information bit or the parity bit. So two adders are designed to calculate the summation of previous forward/backward recursion and branch metrics, and to operate in parallel. This will maximize the throughput. According to the modified Log-MAP algorithm discussed in Chapter 3, the correct values for approximate formula are stored and compensated by a look-up table. This reduces the implementation complexity significantly. The correction values stored in Table 4.5 are also applied to the computation of extrinsic information.

Table 4.5: Look-up table for correction term

lDifl	≤ 0.0625	0.5	1.0	1.5	2.0	≥ 2.0
$\ln(1+e^{-lDifl})$	0.625	0.5	0.375	0.25	0.125	0

The multiplexer select signals are generated by FSM. They choose the right sum according to the number of branch converging to one node, which decides if the correction operation is needed or not.

The Alpha/Beta unit reads in data from memories Mem-alpha0, Mem-alpha1 and Mem-beta0 and Mem-beta1 and writes the results back to them.

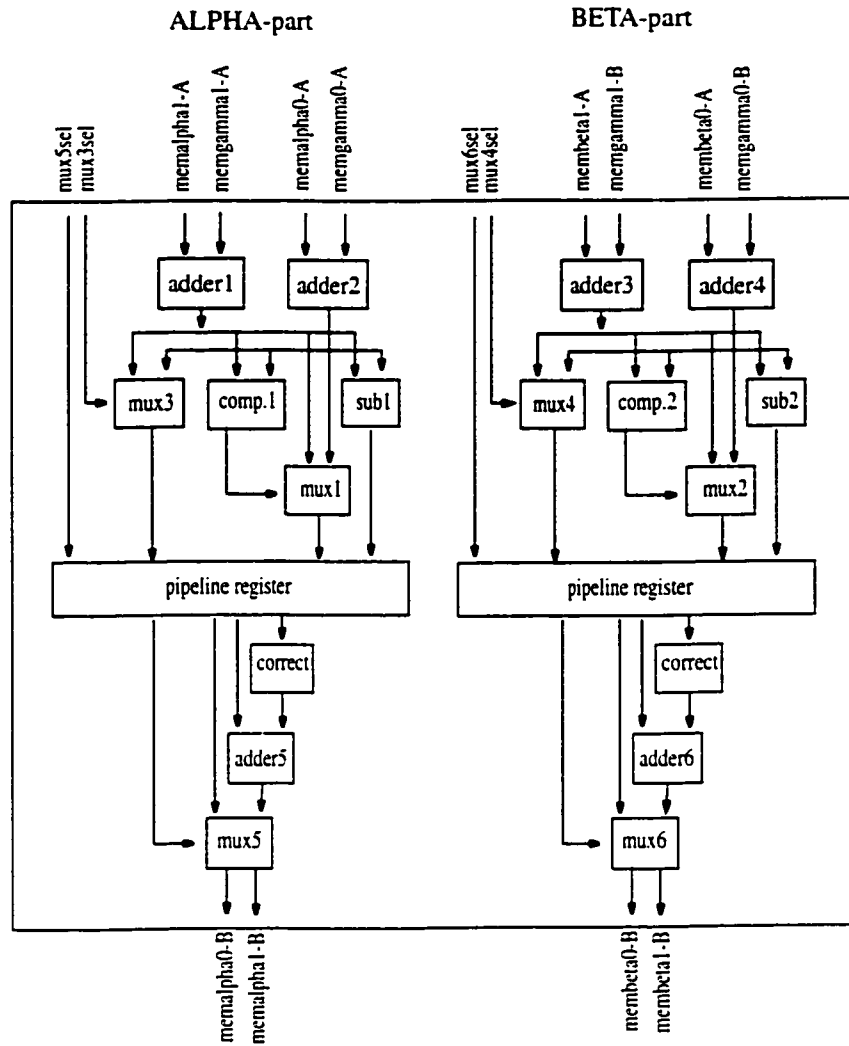


Figure 4.14 Datapath of forward/backward recursion calculation

4.5.6 Extrinsic information unit

The Equation 3.5 requires the exhaustive comparison of the sum of $\log \alpha_{k-1}(s')$ and $\log \beta_k(s)$ of all node pairs connected by branch labelled +1 or -1. So the computation is implemented in parallel for branches labelled +1 and -1. When all branches at a time

instant are computed, the subtractor gives the output of extrinsic information of the transmitted bit at this time.

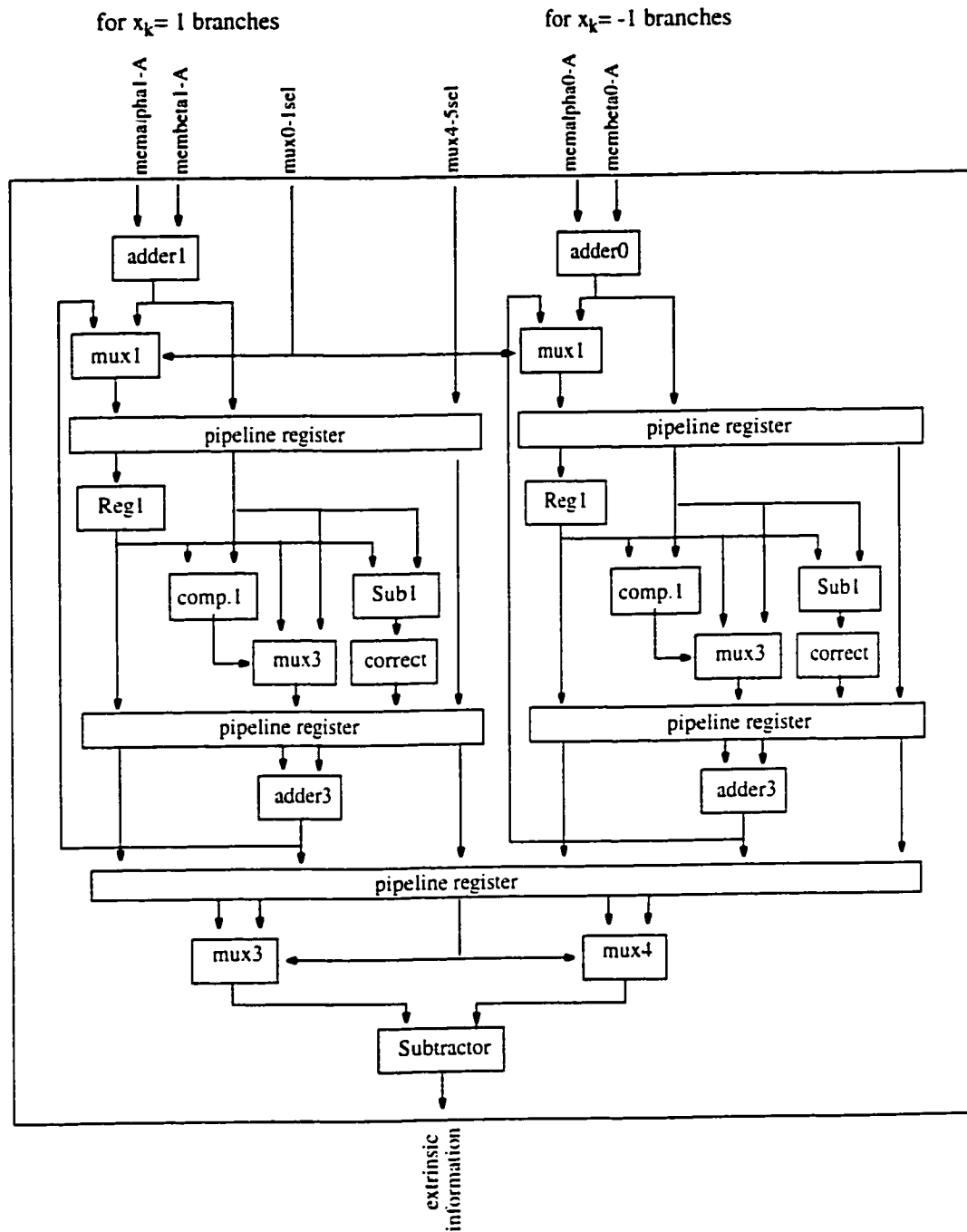


Figure 4.15 Datapath of extrinsic information calculation

The data path of the extrinsic information calculation is shown in Figure 4.15. The circuits labelled +1 and -1 branch are the same. Likewise, the multiplexer select signals are generated by FSM according to the trellis diagram of $RM(32, 26)$.

4.5.7 LLR unit

LLR unit performs the computation of log-likelihood ratio when all iterations are completed. It reads in the data from memories Mem-lcyle and Mem-Le and makes a hard decision according to the sign of sum of both inputs. Its output is the estimated transmitted bits of turbo decoder. LLR unit data path is shown in Figure 4.16.

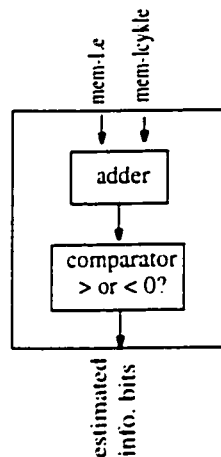


Figure 4.16 Datapath of log-likelihood ratio calculation

4.6 Memory architecture

The size of memories required depend on the interleaver size, word length of data input and intermediate results, and number of trellis states.

In order to take advantage of BlockRAM of FPGA, all of receiver buffer, memories for intermediate results and memories of address table utilize standard memory core

of Xilinx Coregen library. In total 21 groups of memories are used in turbo encoder and decoder. Details are given below and summarized in Table 4.6.

In turbo encoder, one 6 x 26-bit single-port ROM is used to store the generator matrix vectors.

A large amount of single- or dual-port RAM or ROM are used in turbo decoder from receiver buffer to internal memories. Two single-port RAMs Mem-yk are used for receiver buffer, whose size is determined by the transmitted package and bit width of the quantized input. For $RM(32, 26)^2$ turbo code, it is 988 x 4-bit. The ROM Mem-Lc stores the channel reliability values used in pre-processing unit. The RAM Mem-Le stores the extrinsic information outputs of every decoding stage and is refreshed in horizontal and vertical decoding. Because only the extrinsic information of information bits are used in next iteration, the size of Mem-Le is 676 x 12-bit. Most of the used memories are for intermediate results of branch metrics and forward/backward recursion values. All of them are dual-port RAM. Mem-gamma1, Mem-alpha1 and Mem-beta1 are used for branch labelled +1 and Mem-gamma0, Mem-alpha0 and Mem-beta0 for labelled -1. According to node number of trellis diagram of $RM(32, 26)$, the memory size for each $\log \alpha_k(s)$ and $\log \beta_{k-1}(s')$ is 638 x 17-bit. Especially, RAM Mem-lcycle saves the summation of channel data values and extrinsic information in the last vertical decoding to avoid the repetitive calculation in the stage of computing the soft-output of decoder. By this way not only we reduce the decoding delay, but also we eliminate an additional adder circuit. Viewed from total decoding flow of turbo decoder, Mem_lcycle saves the extrinsic information output of last horizontal decoding too. This is important for one

Table 4.6 Memory description of turbo encoder and decoder

Mem. Block	Type	Bitwidth (bits)	Addr. bus (bits)	Num.	Size (bits)	Description
reg_gen3226	s.-port ROM	26	3	1	156	storage of generator vectors for parity bits
memyktop	s.-port RAM	4	10	2	7904	used as receiver buffer
memlctop	s.-port ROM	8	4	1	80	storage memory of L.e values
memlctop	d.-port RAM	12	10	1	8112	storing extrinsic information L.e for horizontal and vertical decoding
memleyktop	s.-port RAM	12	10	1	11856	storing pre-processing results of channel signal
memleyktop	s.-port ROM	13	10	1	8788	storing sum of (L.e+y+L.e) in last vertical decoding
memloggamma0top	d.-port RAM	12	10	2	23712	storing Log values of branch transition probability separately for $x_k=1$ or -1
memalphabetatop	d.-port RAM	17	10	4	43384	storing Log values of forward/backward recursion separately for $x_k=1$ or -1
addrmemlctop	s.-port ROM	10	10	1	6760	addr. table for accessing memlctop and memleyk in vertical decoding
addrmemalphabetatop	s.-port ROM	10	10	1	6360	addr. table for accessing memalphabetatop in alphabet state
addrmemalphabetatop	s.-port ROM	10	10	1	6360	addr. table for accessing memalphabetatop in alphabet state

Mem. Block	Type	Bitwidth (bits)	Addr. bus (bits)	Num.	Size (bits)	Description
addrmembeta0top	s.-port ROM	10	10	1	6360	addr. table for accessing membeta0 in alphabet0 state
addrmembeta1top	s.-port ROM	10	10	1	6360	addr. table for accessing membeta1 in alphabet0 state
addralpha0top	s.-port ROM	10	10	1	5430	addr. table for accessing memalpha0, 1 in leunit state
addrbeta0top	s.-port ROM	10	10	1	5430	addr. table for accessing membeta0 in leunit state
addrbeta1top	s.-port ROM	10	10	1	5430	addr. table for accessing membeta1 in leunit state

SISO decoder and one memory Mem-Le architecture. Because Mem-Le keeps the storage of extrinsic values in horizontal and vertical decoding by overwriting, after last vertical decoding, we don't have the previous horizontal extrinsic values L_e . Hence, the contents of memory Mem-lcycle can be used directly for addition with last vertical extrinsic information L_e in the computation of soft-output LLR.

Among all memories used, there are 8 memories dedicated for storing the address tables which are used to indirectly access the internal memories of the intermediate values. The counters for state change also are used to generate addresses for these memories in each corresponding decoding stage. Their size information is given in Table 4.6.

4.7 Summary

This chapter presents the detail of RM-BTC design and implementation. For fast prototyping, reconfigurable FPGA chip is selected to map the implementation. Besides the turbo encoder implementation, important design issues and hierarchical architecture of decoder are described. Simulation results of quantization are presented and data path design is given. Due to the use of one SISO decoder, data and hardware sharing scheme is designed that minimizes the chip area. In order to increase decoding speed and throughput, all operations are designed with parallelism and pipeline principles. Specific interleaver access technique was used to eliminate the demand for interleaver buffer. Meanwhile, details on finite state machine, data path and interface port of turbo encoder and decoder are also given in this chapter.

Chapter 5

Simulation and synthesis

This chapter will present the results of implementation of turbo encoder and decoder on FPGA. All component parts of turbo encoder and decoder are written in VHDL language and logically synthesized to map to Xilinx FPGA. Test model and simulation environment are described. The functional simulation results at the RTL level and logic synthesis reports of turbo encoder and decoder are presented in Section 2 and 3. Since this is a preliminary hardware implementation, we have paid more attention on correct functionality of turbo encoder and decoder implemented and on chip area minimization. The timing reports also are given as reference for future improvement.

5.1 Test model

The designed turbo encoder and decoder are separately simulated for correct functionality and timing before a combination test is executed. Figure 5.1 shows the block diagram of the combination test model.

The signal generator offers the input information sequence using both fix and random input data patterns, which are sent to turbo encoder for encoding and meanwhile written into a file for later bit error comparison. The encoded information sequences after turbo encoder is corrupted by addition of white noise $N(0, \sigma^2)$ during transmission in

Gaussian channel, which will input to turbo decoder. The estimated transmitted codeword sequences output from turbo decoder are stored in another file. This file will be used in the last comparator that will give bit error rate between transferred information sequence and the estimated transmitted information sequence.

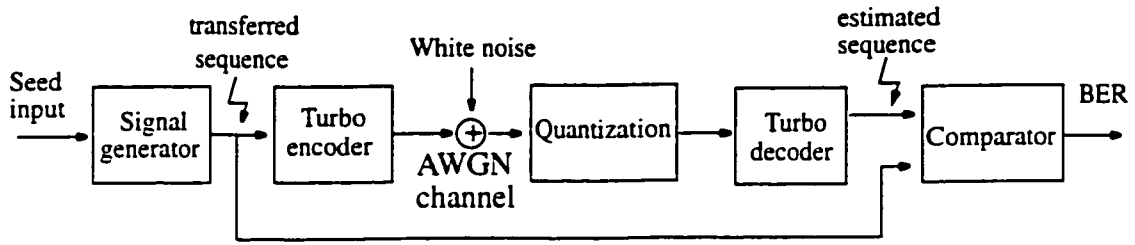


Figure 5.1 Block diagram of test model

All simulations are executed on Sun ultra10 workstation with Solaris 2.5.7 operating system. When implemented by VHDL-87, some arithmetic cores of Xilinx library are used to reduce area. Functional simulation is done with Synopsys VHDL system simulator (VSS); Logic synthesis is through Synopsys design compiler and design analyzer; and Place and Route uses Xilinx Design Manager (Xilinx Alliance M3.1i) targeting to device XCV1000EFG1156-6.

5.2 Simulation

We will denote the fixed test pattern $fix(0^i, 1^j)$ and random test pattern $ran(0^i, 1^j)$ where i means number of bit '0' and j number of bit '1' in one test block.

Although turbo encoder is simpler than turbo decoder, some test patterns are applied to turbo encoder to test its correct functionality independently. The test patterns of $fix(0^{676})$, $fix(1^{676})$, $fix(0, 1)^{338}$, $fix(1, 0)^{338}$, $fix(0^{13}, 1^{13})^{26}$, $fix(1^{13}, 0^{13})^{26}$,

$fix[(0^{13}, 1^{13})(1^{13}, 0^{13})]^{13}$ and $ran(0^{339}, 1^{337}), ran(0^{347}, 1^{329}), ran(0^{318}, 1^{358}), ran(0^{311}, 1^{365})$ are applied to turbo encoder. Partial outputs of them from turbo encoder are verified by manual calculation. For simplicity, further tests of turbo encoder would be combined with decoder.

Here we give the clock cycles required to encode one block of information bits.

$$\frac{\text{clock cycles}}{\text{block}} = 676 + 6 \times 26$$

Where 676 is for shifting information bits into encoder buffer, 6 is the encoding cycles of one codeword and 26 is the codeword number of one block.

Test of turbo decoder is done combining with encoder under two cases of without and with additional noise. The test patterns of encoder listed above are also applied to the turbo decoder. The outputs of these patterns from the encoder are entered to the decoder after 4-bit quantization. For the cases of without additional noise, all estimated outputs of turbo decoder are completely correct, i.e. we obtain the same output as the input patterns of turbo encoder; For case with additional noise, the correct rate or BER of turbo decoder depends on the signal-to-noise ratio of transferred sequences. Here are some random test patterns with additional noise: $ran(0^{357}, 1^{319}), ran(0^{345}, 1^{331}), ran(0^{342}, 1^{334})$ with SNR = 1 dB; $ran(0^{330}, 1^{346}), ran(0^{338}, 1^{338}), ran(0^{342}, 1^{334})$ with SNR = 1.5 dB; $ran(0^{308}, 1^{368}), ran(0^{328}, 1^{348}), ran(0^{326}, 1^{350})$ with SNR = 2 dB. In what follows we present an average BER obtained from a few of blocks under different SNR values.

From 0.5 dB to 2.0 dB of SNR, average bit error rate per block size is in compliance with the one that we obtained from the software simulation. For higher SNR region

from 2.5 to 5.0 dB, because the block size is small, the reported error is zero, so we assume that the BER/block is zero for all of them. The BER/block versus SNR is listed in Table 5.1. The simulation results show that the present implementation of turbo encoder and decoder are working properly and reach expected performance.

Table 5.1: BER/Block versus SNR values

SNR (dB)	BER/Block
0.5	0.106509
1.0	0.0865385
1.5	0.0173815
2.0	0.00702675
2.5 ~ 5.0	0.00000

The number of cycles required to perform one block decoding is a function of the block size and iteration times. Computation performance in term of clock cycles per block can be determined by the following formula:

For the 1st iteration including horizontal and vertical decoding,

$$\frac{\text{clock cycles}}{\text{block}} = 990 + 678 + (639 + 547) \times 26 \times 2$$

where 990 and 678 is clock cycles required in calculation of branch transition in horizontal and vertical decoding, respectively. The 639 and 547 is clock cycles required in calculation of forward/backward recursion and extrinsic information. The 26 is the number of codeword in one block, and the 2 represents both horizontal and vertical decoding stage within one iteration.

For the 2nd and up iterations, every iteration will contribute more cycles to the above formula. Thus the total cycles required from 2nd iteration and up is

$$\frac{\text{clock cycles}}{\text{block}} = [678 + (639 + 547) \times 26] \times 2 \times (\text{iteration} - 1)$$

We have found that decoding process for one block size involves clock cycles which increase proportionally with codeword length (or block size) and iteration number.

5.3 Synthesis and comparison

Both implementation of turbo encoder and decoder are logic synthesized and mapped to Xilinx XCV1000EFG1156-6 FPGA chip. As we mentioned in the previous chapter, Virtex-E FPGA is selected due to its high gate density and fast clock. Table 5.2 and 5.3 list the mapping reports from Place and Route tool.

Normally, utilization of FPGA is defined as the number of logic blocks used for actual realization of the design. It is recommended not to use the device up to its maximum capacity as it would impair the routability of the design. From Table 5.2 and Table 5.3, we can see that both implementation of turbo encoder and decoder occupy only a small area of the chip, 8% and 21% of total slices. This not only makes place and route easy but also saves a lot of chip area for future design improvement. In order to increase the decoding throughput, multiple decoders often are arranged to work in parallel. So accommodating multiple decoders in one chip is very important in codec design. The unused chip area is enough to hold another two decoders.

Regarding timing statistics, post-layout timing reports from Place and Route tool has shown that the turbo encoder can work at the maximum frequency of 47.594 MHz

(21.011 ns) and decoder at the maximum frequency 22.404 MHz (44.634 ns). These two parameters are

Table 5.2: Design summary of turbo encoder

Number of Slices	1079 out of 12,288 (8%)
Number of Slice Registers	1076 out of 24,576 (4%)
Number of 4-input LUTs	1091 out of 24,576 (4%)
Number of Block RAMs	1 out of 96 (1%)
Number of bonded IOBs	5 out of 660 (1%)
Total equivalent gate count for design	31,766

Table 5.3: Design summary of turbo decoder

Number of Slices	2642 out of 12,288 (21%)
Number of Slice Registers	451 out of 24,576 (1%)
Number of 4-input LUTs	3736 out of 24,576 (15%)
Number of Block RAMs	58 out of 96 (60%)
Number of bonded IOBs	18 out of 660 (2%)
Total equivalent gate count for design	988,531

obtained under no timing constraints. These values would have been higher if timing constraint are used. According to these two clock frequencies, we may calculate a preliminary transmission rate of the encoder and the decoder.

For encoding, one block size before decoding equals 676 bits. So, we have the following value:

$$\begin{aligned}
 \text{Transmission rate} &= \frac{676}{832 \times 21.011} \\
 &= 38.67 \text{ Mbps}
 \end{aligned}$$

where 832 is the encoding cycles needed for one block.

For decoding, we assume that iterative decoding is preset to 5 iteration for one block size which is 988 bits after encoding. The total required clock cycles is 315452 deduced from equation of clock cycles per block. Hence the transmission rate of data entering turbo decoder is

$$\begin{aligned} \text{Transmission rate} &= \frac{988}{315452 \times 44.634} \\ &= 70.171 \text{ Kbps} \end{aligned}$$

Because of the recursive calculation of each node and iteration, decoding procedure for one block of information bits takes many clock cycles, which decreases the decoding speed considerable. The bigger the block size, the more the clock cycles required.

Although there are some reported implementations of turbo decoder in the literature, it is difficult to compare one design with another because of different algorithms, component codes, implementation architecture, target devices, etc., and often not all of the performance parameters are available for comparison. Here we give one realization implemented by Xilinx Virtex FPGA[68] as a reference to our implementation.

Table 5.4: Comparison between implementations

Implementation	Reference[44] convolutional code rate 1/3 to 1/7 Log-MAP normalization	Present design block code rate 0.684 Log-MAP Non-normalization
chip	XC3190A-5(1 piece) XC3142A-5(2 piece) XC3130A-5s(2 piece)	XCV1000E-6(1 piece) less than 30% used
clock frequency	10MHz	22.404MHz
decoding speed	v(number of memory)=4 356.8Kbit/s v=2, 624.7Kbit/s	70.171Kbit/s

From Table 5.4 we can see that our implementation outperforms reference[44] in clock frequency. However, comparing with convolutional turbo code, the decoding speed of our implementations is slower than that of reference [44]. This is determined by the different property of convolutional and block code. For the convolutional turbo code, it has regular trellis structure and fixed state number. But for block turbo codes, their state and column number of trellis diagram is proportional to code length and redundancy bit length.

5.4 Summary

In the chapter the simulation scheme and environment for the implemented turbo encoder and decoder are introduced. Through a wide variety of test patterns, we have shown that the implementation of RM-BTC is functionality correct and have similar performance of bit error rate with software model in magnitude. Synthesis report also have shown that the chip area is reasonable. However, the throughput of decoder is lower than prevailing commercial standard.

Chapter 6

Conclusion

6.1 Summary

In this thesis, FPGA implementation of Reed-Muller block turbo encoder and decoder is presented, and its correct functionality is proven by simulation.

Based on knowledge of the linear block code, Reed-Muller block code is constructed with row-reduced echelon generator matrix. Trellis structure determines the complexity of trellis-based decoding for a linear block code. The minimal trellis diagram of Reed-Muller code is constructed by Massey method and is built on row-reduced echelon generator matrix. To take advantage of efficient trellis-based decoding method, trellis-based MAP algorithm for linear block code is introduced. Meantime, iterative decoding and parallel RM turbo code scheme are also illustrated.

For reducing the complexity of MAP decoding algorithm, sub-optimal Max-Log-MAP and Log-MAP algorithms are discussed so as to avoid the complex operations such as exponential and logarithmic computations. Compensation by correction value of Log-MAP is realized with look-up table and simulation result on performance comparison between different algorithms is given. For the sake of implementation, decomposition of Log-MAP algorithm to functional units is discussed.

The hardware implementation of RM-BTC encoder and decoder is the main contribution of this thesis. Instead of cascading elementary decoders to iterate the decoding procedures, a specific SISO decoder structure is proposed to perform iterative decoding. This data- and hardware-sharing scheme significantly reduced the required chip area without any performance degradation and has proven to be a very efficient and a low-complexity implementation. At the same time, parallelism and pipeline design of turbo encoder and decoder has increased the throughput of encoder and decoder.

Simulation of RTL level model of turbo encoder and decoder has shown the correct functionality of the proposed implementation, and synthesis results show that required chip area is small.

From proposed implementation, the following conclusions are obtained.

- Sub-optimal Log-MAP decoding algorithm is a good trade-off between less complexity hardware implementation and better decoding performance. Less than 0.5 dB coding gain degraded from MAP algorithm is observed.
- The small chip area implementation can be achieved through improved one SISO decoder structure and data reuse. Double receiver buffer smooth out the decoding process.
- The remaining chip area of FPGA is a big potential advantage for future improvement of RM-BTC codec.

6.2 Practical limitation and solutions

Although many simplifications (measures) have been incorporated in our design, the decoding delay is still large so that the transmission rate is not practical compared with

available commercial systems. We have the following practical limitations: for block turbo codes, when the code length and the number of redundancy bits are long, exhaustive calculation of forward and backward recursion of every node and branch transition probability of each branch will cost a lot of time and require a large amount of memory; Log-MAP decoding algorithm has better decoding performance but still is computation intensive which results in the lower data transmission rate; Further more, interleaving process makes the decoding operation serial and the high iteration number also exacerbate this decoding delay. So it is necessary to make modifications to the algorithm and the design in order to increase the throughput.

An alternative is the duplication of the functional units of forward/backward recursion and branch metric and decoder. This will increase the decoding throughput due to parallel decoding. If the newest implementation technology is used, the system clock frequency is further increased.

New design could be implemented by full custom application-specific integrated circuit (ASIC). Normalization of forward/backward recursion and extrinsic information should be done. Other modification may add to optimize the design of adders and multipliers.

Another way to increase the throughput could be done by optimizing the search of the trellis diagram. For the time being, MAP decoding algorithm that goes through all the nodes and the branches of trellis diagram during decoding takes long calculation time and introduces a big delay. More research is required on the possibility of reducing the computation node number which just searches part of nodes according to the priority of previous calculation. This should significantly shorten the decoding time.

To reduce power consumption, a great amount of power can be saved by shutting down a portion of inactive decoding stage. Alternatively, finding another architecture for the encoder buffer instead of register-chain used may reduce power consumption further because it is power hungry.

Of all possible alternatives, for the enhancement of the bit rate, selecting a different algorithm or code will have the greatest impact on the performance of proposed system.

Bibliography

- [1] George C. Clark and J. Bibb Cain, *Error-correction Coding for Digital Communications*, New York, Plenum Press, 1981.
- [2] Forney, G. D., *Concatenated Codes*, Cambridge, Massachusetts: MIT Press, 1966.
- [3] John G. Proakis, *Digital Communication*, 3rd edition, New York: McGraw-Hill, 1995.
- [4] Lin, S and Costello, DJ, *Error control Coding: Fundamentals and Applications*, 1983, Prentice-Hall, NJ.
- [5] Usa Vilaipornsawai, *Trellis-Based Iterative decoding of Block Codes for Satellite ATM*, MASc thesis, concordia university, Feb. 2001
- [6] C. E. Shannon, *A Mathematical Theory of Communications*, Bell System Technical Journal, Vol27, July 1948, pp379-423 and 623-656.
- [7] P. Sweeney, *Error Control Coding an Introduction*, New York, USA, Prentice Hall International, 1991
- [8] S. Lin, *An introduction to Error-Correcting Codes*, New Jersey, Prentice Hall, 1970.
- [9] E. R. Berlekamp. *Algebraic Coding Theory*, New York, McGraw-Hill.
- [10] C. Berrou, A. Glavieux and P. Thitimajshima, *Near Shannon limit error-correcting coding and decoding: Turbo codes*. ICC'93, Geneva, Switzerland, May 93, pp 1064-1070.
- [11] C. Berrou, A. Glavieux, "Near Optimum Error Correcting coding and Decoding: Turbo-Codes, IEEE Trans. on Comm. Vol44, No.10, Oct.1996, pp1261-1271.
- [12] R. Pyndiah, A. Glavieux, A. Picart and S. Jacq, *Near optimum decoding of products codes*, GLOBECOM'94, San Francisco, CA, Dec. 94, pp.339-343.
- [13] R. Pyndiah, *Near-Optimum Decoding of Product Codes: Block Turbo Codes*, IEEE trans. on comm. Vol.46, No.8, Aug. 1998, pp.1003-1010.
- [14] L. R. Bahl, J. Cocke, F. Jelinek and J. Raviv, *Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate*, IEEE trans. on Information Theory, March, 1974, pp.284-287.

- [15] P. Robertson, E. Villebrun and P. Hoeher, *A Comparison of Optimal and sub-optimal MAP Decoding Algorithm Operating in the Log Domain*, IEEE Int. Conf. on Comm., Seattle, WA, Jun. 1995, pp.1009-1013,
- [16] P. Robertson, P. Hoeher, *Optimal and Sub-optimal Maximum a Posteriori Algorithms Suitable for Turbo Decoding*, European trans. on Telecommun., Vol. 8, No.2, Mar-Apr. 1997, pp.119-125.
- [17] J. Hagenauer, E. Offer and L. Papke, *Iterative Decoding of Binary Block and Convolutional Codes*, IEEE trans. on Information Theory, Vol. 42, No.2, Mar. 1996, pp.429-445.
- [18] B. Vucetic, J. H. Yuan, *Turbo Codes: Principles and Applications*, Kluwer Academic Publishers, 2000.
- [19] S. Lin *Trellises and trellis-based decoding algorithms for linear block codes*,
- [20] R. E. Blahut, *Theory and practice of Error Control Codes*, Addison-Wesley, 1983.
- [21] B. Honary and G. Markarian, *Trellis Decoding of Block Codes: A Practical Approach*, Kluwer Academic Publishers, 1997.
- [22] V. S. Pless and W. C. Huffman, *Handbook of Coding Theory*, Elsevier science B.V., Vol II, Amsterdam, Netherland, pp2027-2029, 1998.
- [23] J. Hagenauer, P. Hoeher, *A Viterbi algorithm with soft-decision outputs and its applications*, Proc. GLOBECOM'89, Nov. 1989, pp1680-1686.
- [24] J. A. Erfanian, S. Pasupathy, G. Gulak, *Reduced Complexity Symbol Detectors with Parallel Structures for ISI Channels*, IEEE Trans. Commun. Vol. 42, Feb/Mar/Apr. 1994, pp1661-1671.
- [25] W. Koch, A. Baier, *Optimum and Sub-optimum Detection of Coded Data Disturbed by Time-varying Intersymbol interference*, Proc. GLOBECOM'90, Dec.1990, pp1679-1684.
- [26] R. Achiba, M. Mortazavi, W. Fizell, *Turbo Code Performance and Design Trade-offs*, 0-7803-6521-6/2000 IEEE.
- [27] K. Koora, S. Zeisberg and A. Finger, *New High Speed Flexible Turbo-Block-Decoding*, Proc. of ACTS Mobile Communications Summit, 8-11 Jun 1998, Rhodes-Greece, pp653-658.

- [28] P. Adde, R. Pyndiah and O. Raoul, *Performance and Complexity of Block Turbo Decoder Circuits*, Proc. of the third IEEE inter. conf. on Electronics, circuits and system. ICECS'96, pp.172-175.
- [29] R. Pyndiah, P. Combelles and P. Adde, *A Very Low Complexity Block Turbo Decoder for Product codes*, 07803-3336-5/96, 1996 IEEE
- [30] C. Schurgers, F. Catthoor and M. Engels, *Energy Efficient Data Transfer and Storage Organization for a MAP Turbo Decoder Module*, 1999 Inter. Sym. on Low Power Electronics and Design (ISLPED'99), San Diego, CA, Aug. 1999, pp76-81.
- [31] T. Y. Huwang, *Efficient Optimal Decoding of Linear Block Codes*, IEEE trans. on Info. theory, Vol.IT-26, No.5, Sept. 1980.. Schurgers, F. Catthoor and M. Engels, *Energy Efficient Data Transfer and Storage Organization for a MAP Turbo Decoder Module*, 1999 Inter. Sym. on Low Power Electronics and Design (ISLPED'99), San Diego, CA, Aug. 1999, pp76-81.
- [32] S. Hauck, *the Role of FPGA's in Reprogrammable Systems*, Proc. of the IEEE, Vol.86, No.4, Apr. 1998, pp613-637. (0018-9219/98).
- [33] Xilinx Inc., *Virtex-E 1.8V Field Programmable Gate Arrays Data Sheet*, DS022-1~4, Apr. 2001.
- [34] D. E. Cress, W. J. Ebel, *Turbo Code Implementation Issues for Low Latency, Low Power Application*, Proc. Virginia Tech. Symp. on Wireless Personal Comm.: Emerging technologies for Enhanced Communications, Jun. 1998, pp191-200.
- [35] Y. Wu and B. D. Woerner, *The Influence of Quantization and Fixed Point Arithmetic upon the BER Performance of Turbo Codes*, Proc. IEEE inter. conference on Vehicular Tech., VTC spring'99, May, 1999, pp1683-1687.
- [36] F. Berens, A. Worm, H. Michel and N. Wehn, *Implementation Aspects of Turbo-Decoders for Future Radio Applications*, Proc. VTC'99 Fall, Amsterdam, Netherlands, Sep. 1999, pp2601-2605.
- [37] G. Jeong, D. Hsia, *Optimal Quantization for Soft-Decision Turbo Decoder*, Proc. VTC'99 Fall, Amsterdam, Netherlands, Sep. 1999, pp1620-1624.
- [38] H. Michel, A. Worm and N. Wehn, *Influence of Quantization on the Bit-Error Performance of Turbo-Decoders*, Proc. IEEE inter. conference on Vehicular Tech., VTC spring'00, Tokyo, Japan, May 2000, pp581-585.

- [39] H. Michel and N. Wehn, *Turbo-Decoder Quantization for UMTS*, submitted to IEEE communications letter.
- [40] Z. Wang, H. Suzuki and K. K. Parhi, *VLSI Implementation Issues of Turbo Decoder Design for Wireless Applications*, Proc. IEEE workshop on Signal Processing Systems, Design and Implementation, Taipei, Taiwan, Oct20-22, 1999, pp503-512.
- [41] S. Hong, J. Yi and W. E. Stark, *VLSI Design and Implementation of Low-complexity Adaptive Turbo-code Encoder and Decoder for Wireless Mobile Communication Applications*, IEEE workshop on Signal Processing System: Design and Implementation, Cambridge, Massachusetts, Oct. 1998, pp233-242.
- [42] C. B. Shung, G. Ungerboeck and H. K. Thapar, *VLSI Architectures for Metric Normalization in the Viterbi Algorithm*, IEEE inter. Conf. on Communication : ICC'90, pp1723-1728.
- [43] S. Halter, M. Oberg, P. M. Chau and P. H. Siegel, *Reconfigurable signal Processor for Channel coding & Decoding in Low SNR Wireless Communications*, Proc. IEEE workshop on Signal Processing System: Design and Implementation. (SiPs'98) Cambridge, Massachusetts, Oct. 1998, pp260-274.
- [44] S. S. Pietroben, *Implementation and Performance of A Turbo/MAP Decoder*, International Journal of Satellite Communications, 16(1998), pp23-46.
- [45] G. Masera, G. Piccinini, M. R. Roch and M. Zamboni, *VLSI Architectures for Turbo Codes*, IEEE Trans. on Very Large Scale Integration(VLSI) System, Vol. 7, No.3 Sep. 1999, pp369-379.
- [46] A. Worm, H. Lamm and N. Wehn, *VLSI Architectures for High-Speed MAP Decoders*, VLSI Design, 2001: Fourteenth International Conference, pp446-453.
- [47] H. Dawid, G. Gehnen and H. Meyr, *MAP Channel Decoding: Algorithm and VLSI Architecture*, VLSI Signal Processing VI, 1993, pp141-149.
- [48] N. V. Stralen and S. Hladik, *Design of Turbo Decoder ASIC*, 2nd Inter. Symposium on Turbo Codes&Related Topics, Brest, France, 2000, pp275-278.
- [49] C. Schurgers, F. Catthoor and M. Engels, *Optimized MAP Turbo Decoder*, IEEE workshop on Signal Processing System: Design and Implementation, SIPs 2000, Oct, 2000, pp245-254.

- [50] K. Gracie, S. Crozier, A. Hunt and J. Lodge, *Performance of a Low -Complexity Turbo Decoder and its Implementation on a Low-cost, 16-Bit Fixed-Point DSP*, email:ken.gracie@crc.ca
- [51] T. Ngo and I. Verbauwhede, *Fixed Point Implementation for Turbo Codes*, Final report 1998-99 for Micro Project 98-162. EE. Univ. of CA.
- [52] A. Raghupathy and K. J. R. Liu, *A Transformation for Computational Latency reduction in Turbo-MAP Decoding*, 0-7803-5471-0/99/, 1999 IEEE, ppIV402-IV405.
- [53] A. Goalic, N. Chapalain and R. Pyndiah, *Implementation of a Low Complexity Algorithm for BCH Block Turbo Decoding on Digital Signal Processor*, email:andre.golic@enst-bretagne.fr.
- [54] E. Boutillon, W. J. Gross and G. Gulak, *VLSI Architectures for the Forward-Backward Algorithm*, Submitted as a transaction paper to the IEEE Trans. on Comm..
- [55] F. Raouafi, A. Dingninou and C. Berrou, *Saving memory in turbo-decoders using the MAX-Log-MAP Algorithm*, Turbo Codes in Digital Broadcasting - Could it double Capacity, IEE Colloquium on,1999, pp14/1-14/4.
- [56] J. Vogt, J. Ertel and A. Finger, *Reducing bit width of extrinsic memory in turbo decoder realizations*, Electronic Letter, 28 Sep. 2000, Vol.36, No.20. pp1714-1716.
- [57] W. J. Gross and P. G. Gulak, *Simplified MAP algorithm suitable for implementation of turbo decoders*, Electronic Letter, 6 Aug. 1998, Vol.34 No.16, pp1577-1578.
- [58] H. Sugimoto, T. Wang and X. Ping, *A sub-MAP Decoder performance in IMT2000 mobile environment*, 2nd Inter. Symposium on Turbo Codes&Related Topics, Brest, France,2000.
- [59] J. Yuan, B. Vucetic and W. Feng, *Combined Turbo Codes and Interleaver Design*, IEEE trans. on communication, Vol.47, No.4, Apr. 1999, pp484-487.
- [60] C. Yi and J. H. Lee, *Interleaveing and Decoding Scheme for a Product Code for a Mobile Data Communication*, IEEE trans. on communication, Vol.45, No.2 Feb, 1997,pp144-147.
- [61] J. K. Wolf, *Efficient Maximum Likelihood Decoding of Linear Block Codes Using a Trellis*, IEEE trans. on information theory, Vol.IT-24, No.1 Jan. 1978, pp76-80.
- [62] M. C. Valenti, *Inserting Turbo Code Technology into the DVB Satellite Broadcasting System*, email:mvalenti@wvu.edu.

- [63] Y. Kaji, R. Shibuya, T. Fujiwara, T. Kasami and S. Lin, *MAP and LogMAP Decoding Algorithm for Linear Block Codes Using a Code Structure*, IEICE Trans. fundamentals, Vol. E83-A, No.10, Oct. 2000, pp1884-1890.
- [64] J. Vogt and A. Finger, *Improving the max-log-MAP turbo decoder*, Electronic Letter, 9th Nov. 2000, Vol.36, No.23, pp1937-1939.
- [65] P. Jung and M. M. Nabhan, *Comprehensive comparison of Turbo-Code decoders*, IEEE 45th Vehicular Conference, Chicago, IL, Jul.25-28,1995, pp624-628.
- [66] Z. Wang and K. K. Parhi, *Decoding Metrics and their Applications in VLSI Turbo Decoders*, Acoustics, Speech and Signal Processing , 2000, ICASSP'00, Proc. 2000 IEEE inter. conf. on Vol.6,2000, pp3370-3373.
- [67] SoftDSP Corporation, *Turbo Decoder Core for ASIC&System Development*, Designer's Guide, May 2000, Version 1.2.2.
- [68] W. B. Puckett, *Implementation and Performance of an IMproved Turbo Decoder on a Configurable Computing Machine*, Master thesis paper, Virginia Polytechnic Institute and State University, July 2000

Appendix A Synthesis Reports

Turbo encoder mapping report

Xilinx Mapping Report File for Design 'Synopsys_edif'
Copyright (c) 1995-2000 Xilinx, Inc. All rights reserved.

Design Information

Command Line : map -p xcv1000e-6-fg1156 -o map.ncd -detail encoder_c.ngd
encoder_c.pcf
Target Device : xcv1000e
Target Package : fg1156
Target Speed : -6
Mapper Version : virtexe -- D.27
Mapped Date : Mon Apr 1 14:17:41 2002

Design Summary

Number of errors: 0
Number of warnings: 11
Number of Slices: 1,079 out of 12,288 8%
Number of Slices containing
unrelated logic: 0 out of 1,079 0%
Total Number Slice Registers: 1,076 out of 24,576 4%
Number used as Flip Flops: 1,024
Number used as Latches: 52
Total Number 4 input LUTs: 1,091 out of 24,576 4%
Number used as LUTs: 1,069
Number used as a route-thru: 22
Number of bonded IOBs: 5 out of 660 1%
Number of Block RAMs: 1 out of 96 1%
Total equivalent gate count for design: 31,766
Additional JTAG gate count for IOBs: 240

Turbo encoder logic level timing summary

Xilinx TRACE, Version D.27
Copyright (c) 1995-2000 Xilinx, Inc. All rights reserved.

trce map.ncd encoder_c.pcf -a -v 1 -skew -o map.twr -xml map_trce.xml

Design file: map.ncd
Physical constraint file: encoder_c.pcf
Device speed: xcv1000e,-6 (PRELIMINARY 1.60 2001-05-08)
Report level: verbose report, limited to 1 item per constraint

Timing summary:

Timing errors: 0 Score: 0

Constraints cover 26136 paths, 0 nets, and 6215 connections (92.3% coverage)

Design statistics:

Minimum period: 11.039ns (Maximum frequency: 90.588MHz)

Maximum combinational path delay: 4.221ns

Minimum input arrival time before clock: 2.046ns

Maximum output required time before clock: 13.804ns

Analysis completed Mon Apr 1 14:18:50 2002

Turbo encoder post layout timing summary

Xilinx TRACE, Version D.27

Copyright (c) 1995-2000 Xilinx, Inc. All rights reserved.

trce encoder_c.ncd encoder_c.pcf -a -v 3 -skew -o encoder_c.twr -xml
encoder_c_trce.xml

Design file: encoder_c.ncd

Physical constraint file: encoder_c.pcf

Device.speed: xcv1000e-6 (PRELIMINARY 1.60 2001-05-08)

Report level: verbose report

Timing summary:

Timing errors: 0 Score: 0

Constraints cover 26136 paths, 0 nets, and 6215 connections (92.3% coverage)

Design statistics:

Minimum period: 21.011ns (Maximum frequency: 47.594MHz)

Maximum combinational path delay: 11.933ns

Minimum input arrival time before clock: 3.724ns

Maximum output required time before clock: 18.718ns

Analysis completed Mon Apr 1 14:20:51 2002

Turbo decoder mapping report

Xilinx Mapping Report File for Design 'Synopsys_edif'

Copyright (c) 1995-2000 Xilinx, Inc. All rights reserved.

Design Information

Command Line : map -p xcv1000e-6-fgl156 -o map.ncd decoder.ngd decoder.pcf

Target Device : xv1000e

Target Package : fgl156

Target Speed : -6

Mapper Version : virtexe -- D.27
 Mapped Date : Tue Apr 2 23:07:33 2002

Design Summary

```

-----
Number of errors:    0
Number of warnings: 133
Number of Slices:    2,642 out of 12,288  21%
Number of Slices containing
  unrelated logic:    0 out of 2,642  0%
Total Number Slice Registers: 451 out of 24,576  1%
  Number used as Flip Flops:    447
  Number used as Latches:        4
Total Number 4 input LUTs: 3,736 out of 24,576  15%
  Number used as LUTs:          3,070
  Number used as a route-thru:   666
Number of bonded IOBs:    18 out of 660  2%
Number of Block RAMs:     58 out of 96  60%
Number of RPM macros:     4
Total equivalent gate count for design: 988,531
Additional JTAG gate count for IOBs: 864
  
```

Turbo decoder logic level timing summary

```

-----
Xilinx TRACE, Version D.27
Copyright (c) 1995-2000 Xilinx, Inc. All rights reserved.

trce map.ncd decoder.pcf -v 1 -o map.twr -xml map_trce.xml

Design file:      map.ncd
Physical constraint file: decoder.pcf
Device, speed:    xcv1000e,-6 (PRELIMINARY 1.60 2001-05-08)
Report level:     verbose report, limited to 1 item per constraint
  
```

Timing summary:

 Timing errors: 0 Score: 0

Constraints cover 96616809 paths, 4988 nets, and 13096 connections (100.0% coverage)

Design statistics:

Minimum period: 26.146ns (Maximum frequency: 38.247MHz)
 Maximum net delay: 0.879ns

Analysis completed Fri Jun 28 13:42:50 2002

Turbo decoder post layout timing summary

```

-----
Xilinx TRACE, Version D.27
Copyright (c) 1995-2000 Xilinx, Inc. All rights reserved.
  
```

trce decoder.ncd decoder.pcf -e 3 -o decoder.twr -xml decoder_trce.xml

Design file: decoder.ncd
Physical constraint file: decoder.pcf
Device, speed: xcv1000e,-6 (PRELIMINARY 1.60 2001-05-08)
Report level: error report

Timing summary:

Timing errors: 0 Score: 0

Constraints cover 96616809 paths, 5687 nets, and 13096 connections (100.0% coverage)

Design statistics:
Minimum period: 44.634ns (Maximum frequency: 22.404MHz)
Maximum net delay: 13.468ns

Analysis completed Fri Jun 28 13:50:51 2002

Appendix B Turbo encoder VHDL source codes

File name: encoder_c.vhd

purpose: main program of turbo encoder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity encoder_c is
  generic (
    bitwidth : integer := 26;      -- RM(32,26) codeword information length;
    buffersize : integer := 676;   -- RM(32,26) input buffer size for test
    paritysize : integer := 156);
  port (
    input : in std_logic;  -- input: received sequence info.bit;
    reset : in std_logic;  --initiate signal for state and register Q;
    clk : in std_logic;
    start : in std_logic;
    codeword : out std_logic);  -- coded word output;
end encoder_c;

architecture rtl of encoder_c is

  type statetype is (idle, receive, encode, shift_out);
  signal state : statetype; -- initial state is waiting

  component reg_gen3226
    port (
      addr : in std_logic_vector(2 downto 0);  -- memory addressing
      clk : in std_logic;
      dout : out std_logic_vector(25 downto 0)); -- matrix vector output
  end component;

  signal g : std_logic_vector(bitwidth-1 downto 0); --codeword generator metrix vector;
  signal Q : std_logic_vector(buffersize-1 downto 0); --buffer length vector;
  signal mux_out_H, mux_out_V, m_out_H, m_out_V : std_logic_vector(bitwidth-1 downto 0); --multiplier
  output
  signal xor_out_H, xor_out_V : std_logic_vector(bitwidth-1 downto 1); --XOR gate output;
  signal paritybit_H : std_logic_vector(paritysize-1 downto 0); -- Hori. coded Par. bits length vector;
  signal paritybit_V : std_logic_vector(paritysize-1 downto 0); -- Ver. coded Par. bits length vector;
  signal R_shift, encoding, shifting_out : std_logic; --FSM output signals;
  signal enco1 : std_logic;
  signal addr : std_logic_vector(2 downto 0); --addr signal of codeword generator-metrix memory
  signal jump, jump_delay : std_logic_vector(4 downto 0); --counting signal of codeword number;
  signal countup, count : std_logic_vector(9 downto 0);
  signal QH0, QH1, QH2, QH3, QH4, QH5, QH6, QH7, QH8, QH9 : std_logic_vector(25 downto 0);
  signal QH10, QH11, QH12, QH13, QH14, QH15, QH16, QH17, QH18, QH19 : std_logic_vector(25 downto 0);
  signal QH20, QH21, QH22, QH23, QH24, QH25 : std_logic_vector(25 downto 0);
```

```

signal QV0, QV1, QV2, QV3, QV4, QV5, QV6, QV7, QV8, QV9 : STD_LOGIC_VECTOR(25 downto
0);
signal QV10, QV11, QV12, QV13, QV14, QV15, QV16, QV17, QV18 : STD_LOGIC_VECTOR(25
downto 0);
signal QV19, QV20, QV21, QV22, QV23, QV24, QV25 : STD_LOGIC_VECTOR(25 downto 0);
signal Sel : std_logic_vector(4 downto 0);

begin

QH0 <= Q(25 downto 0);
QH1 <= Q(51 downto 26);
QH2 <= Q(77 downto 52);
QH3 <= Q(103 downto 78);
QH4 <= Q(129 downto 104);
QH5 <= Q(155 downto 130);
QH6 <= Q(181 downto 156);
QH7 <= Q(207 downto 182);
QH8 <= Q(233 downto 208);
QH9 <= Q(259 downto 234);
QH10 <= Q(285 downto 260);
QH11 <= Q(311 downto 286);
QH12 <= Q(337 downto 312);
QH13 <= Q(363 downto 338);
QH14 <= Q(389 downto 364);
QH15 <= Q(415 downto 390);
QH16 <= Q(441 downto 416);
QH17 <= Q(467 downto 442);
QH18 <= Q(493 downto 468);
QH19 <= Q(519 downto 494);
QH20 <= Q(545 downto 520);
QH21 <= Q(571 downto 546);
QH22 <= Q(597 downto 572);
QH23 <= Q(623 downto 598);
QH24 <= Q(649 downto 624);
QH25 <= Q(675 downto 650);

QV0 <=
Q(650)&Q(624)&Q(598)&Q(572)&Q(546)&Q(520)&Q(494)&Q(468)&Q(442)&Q(416)&Q(390)&Q(364
)&Q(338)&Q(312)&Q(286)&Q(260)&Q(234)&Q(208)&Q(182)&Q(156)&Q(130)&Q(104)&Q(78)&Q(52
)&Q(26)&Q(0);
QV1 <=
Q(651)&Q(625)&Q(599)&Q(573)&Q(547)&Q(521)&Q(495)&Q(469)&Q(443)&Q(417)&Q(391)&Q(365
)&Q(339)&Q(313)&Q(287)&Q(261)&Q(235)&Q(209)&Q(183)&Q(157)&Q(131)&Q(105)&Q(79)&Q(53
)&Q(27)&Q(1);
QV2 <=
Q(652)&Q(626)&Q(600)&Q(574)&Q(548)&Q(522)&Q(496)&Q(470)&Q(444)&Q(418)&Q(392)&Q(366
)&Q(340)&Q(314)&Q(288)&Q(262)&Q(236)&Q(210)&Q(184)&Q(158)&Q(132)&Q(106)&Q(80)&Q(54
)&Q(28)&Q(2);
QV3 <=
Q(653)&Q(627)&Q(601)&Q(575)&Q(549)&Q(523)&Q(497)&Q(471)&Q(445)&Q(419)&Q(393)&Q(367
)&Q(341)&Q(315)&Q(289)&Q(263)&Q(237)&Q(211)&Q(185)&Q(159)&Q(133)&Q(107)&Q(81)&Q(55
)&Q(29)&Q(3);
QV4 <=
Q(654)&Q(628)&Q(602)&Q(576)&Q(550)&Q(524)&Q(498)&Q(472)&Q(446)&Q(420)&Q(394)&Q(368

```

```

)&Q(342)&Q(316)&Q(290)&Q(264)&Q(238)&Q(212)&Q(186)&Q(160)&Q(134)&Q(108)&Q(82)&Q(56)
)&Q(30)&Q(4);
  QV5 <=
Q(655)&Q(629)&Q(603)&Q(577)&Q(551)&Q(525)&Q(499)&Q(473)&Q(447)&Q(421)&Q(395)&Q(369)
)&Q(343)&Q(317)&Q(291)&Q(265)&Q(239)&Q(213)&Q(187)&Q(161)&Q(135)&Q(109)&Q(83)&Q(57)
)&Q(31)&Q(5);
  QV6 <=
Q(656)&Q(630)&Q(604)&Q(578)&Q(552)&Q(526)&Q(500)&Q(474)&Q(448)&Q(422)&Q(396)&Q(370)
)&Q(344)&Q(318)&Q(292)&Q(266)&Q(240)&Q(214)&Q(188)&Q(162)&Q(136)&Q(110)&Q(84)&Q(58)
)&Q(32)&Q(6);
  QV7 <=
Q(657)&Q(631)&Q(605)&Q(579)&Q(553)&Q(527)&Q(501)&Q(475)&Q(449)&Q(423)&Q(397)&Q(371)
)&Q(345)&Q(319)&Q(293)&Q(267)&Q(241)&Q(215)&Q(189)&Q(163)&Q(137)&Q(111)&Q(85)&Q(59)
)&Q(33)&Q(7);
  QV8 <=
Q(658)&Q(632)&Q(606)&Q(580)&Q(554)&Q(528)&Q(502)&Q(476)&Q(450)&Q(424)&Q(398)&Q(372)
)&Q(346)&Q(320)&Q(294)&Q(268)&Q(242)&Q(216)&Q(190)&Q(164)&Q(138)&Q(112)&Q(86)&Q(60)
)&Q(34)&Q(8);
  QV9 <=
Q(659)&Q(633)&Q(607)&Q(581)&Q(555)&Q(529)&Q(503)&Q(477)&Q(451)&Q(425)&Q(399)&Q(373)
)&Q(347)&Q(321)&Q(295)&Q(269)&Q(243)&Q(217)&Q(191)&Q(165)&Q(139)&Q(113)&Q(87)&Q(61)
)&Q(35)&Q(9);
  QV10 <=
Q(660)&Q(634)&Q(608)&Q(582)&Q(556)&Q(530)&Q(504)&Q(478)&Q(452)&Q(426)&Q(400)&Q(374)
)&Q(348)&Q(322)&Q(296)&Q(270)&Q(244)&Q(218)&Q(192)&Q(166)&Q(140)&Q(114)&Q(88)&Q(62)
)&Q(36)&Q(10);
  QV11 <=
Q(661)&Q(635)&Q(609)&Q(583)&Q(557)&Q(531)&Q(505)&Q(479)&Q(453)&Q(427)&Q(401)&Q(375)
)&Q(349)&Q(323)&Q(297)&Q(271)&Q(245)&Q(219)&Q(193)&Q(167)&Q(141)&Q(115)&Q(89)&Q(63)
)&Q(37)&Q(11);
  QV12 <=
Q(662)&Q(636)&Q(610)&Q(584)&Q(558)&Q(532)&Q(506)&Q(480)&Q(454)&Q(428)&Q(402)&Q(376)
)&Q(350)&Q(324)&Q(298)&Q(272)&Q(246)&Q(220)&Q(194)&Q(168)&Q(142)&Q(116)&Q(90)&Q(64)
)&Q(38)&Q(12);
  QV13 <=
Q(663)&Q(637)&Q(611)&Q(585)&Q(559)&Q(533)&Q(507)&Q(481)&Q(455)&Q(429)&Q(403)&Q(377)
)&Q(351)&Q(325)&Q(299)&Q(273)&Q(247)&Q(221)&Q(195)&Q(169)&Q(143)&Q(117)&Q(91)&Q(65)
)&Q(39)&Q(13);
  QV14 <=
Q(664)&Q(638)&Q(612)&Q(586)&Q(560)&Q(534)&Q(508)&Q(482)&Q(456)&Q(430)&Q(404)&Q(378)
)&Q(352)&Q(326)&Q(300)&Q(274)&Q(248)&Q(222)&Q(196)&Q(170)&Q(144)&Q(118)&Q(92)&Q(66)
)&Q(40)&Q(14);
  QV15 <=
Q(665)&Q(639)&Q(613)&Q(587)&Q(561)&Q(535)&Q(509)&Q(483)&Q(457)&Q(431)&Q(405)&Q(379)
)&Q(353)&Q(327)&Q(301)&Q(275)&Q(249)&Q(223)&Q(197)&Q(171)&Q(145)&Q(119)&Q(93)&Q(67)
)&Q(41)&Q(15);
  QV16 <=
Q(666)&Q(640)&Q(614)&Q(588)&Q(562)&Q(536)&Q(510)&Q(484)&Q(458)&Q(432)&Q(406)&Q(380)
)&Q(354)&Q(328)&Q(302)&Q(276)&Q(250)&Q(224)&Q(198)&Q(172)&Q(146)&Q(120)&Q(94)&Q(68)
)&Q(42)&Q(16);
  QV17 <=
Q(667)&Q(641)&Q(615)&Q(589)&Q(563)&Q(537)&Q(511)&Q(485)&Q(459)&Q(433)&Q(407)&Q(381)
)&Q(355)&Q(329)&Q(303)&Q(277)&Q(251)&Q(225)&Q(199)&Q(173)&Q(147)&Q(121)&Q(95)&Q(69)
)&Q(43)&Q(17);

```

```

QV18 <=
Q(668)&Q(642)&Q(616)&Q(590)&Q(564)&Q(538)&Q(512)&Q(486)&Q(460)&Q(434)&Q(408)&Q(382)
)&Q(356)&Q(330)&Q(304)&Q(278)&Q(252)&Q(226)&Q(200)&Q(174)&Q(148)&Q(122)&Q(96)&Q(70)
)&Q(44)&Q(18);
QV19 <=
Q(669)&Q(643)&Q(617)&Q(591)&Q(565)&Q(539)&Q(513)&Q(487)&Q(461)&Q(435)&Q(409)&Q(383)
)&Q(357)&Q(331)&Q(305)&Q(279)&Q(253)&Q(227)&Q(201)&Q(175)&Q(149)&Q(123)&Q(97)&Q(71)
)&Q(45)&Q(19);
QV20 <=
Q(670)&Q(644)&Q(618)&Q(592)&Q(566)&Q(540)&Q(514)&Q(488)&Q(462)&Q(436)&Q(410)&Q(384)
)&Q(358)&Q(332)&Q(306)&Q(280)&Q(254)&Q(228)&Q(202)&Q(176)&Q(150)&Q(124)&Q(98)&Q(72)
)&Q(46)&Q(20);
QV21 <=
Q(671)&Q(645)&Q(619)&Q(593)&Q(567)&Q(541)&Q(515)&Q(489)&Q(463)&Q(437)&Q(411)&Q(385)
)&Q(359)&Q(333)&Q(307)&Q(281)&Q(255)&Q(229)&Q(203)&Q(177)&Q(151)&Q(125)&Q(99)&Q(73)
)&Q(47)&Q(21);
QV22 <=
Q(672)&Q(646)&Q(620)&Q(594)&Q(568)&Q(542)&Q(516)&Q(490)&Q(464)&Q(438)&Q(412)&Q(386)
)&Q(360)&Q(334)&Q(308)&Q(282)&Q(256)&Q(230)&Q(204)&Q(178)&Q(152)&Q(126)&Q(100)&Q(74)
)&Q(48)&Q(22);
QV23 <=
Q(673)&Q(647)&Q(621)&Q(595)&Q(569)&Q(543)&Q(517)&Q(491)&Q(465)&Q(439)&Q(413)&Q(387)
)&Q(361)&Q(335)&Q(309)&Q(283)&Q(257)&Q(231)&Q(205)&Q(179)&Q(153)&Q(127)&Q(101)&Q(75)
)&Q(49)&Q(23);
QV24 <=
Q(674)&Q(648)&Q(622)&Q(596)&Q(570)&Q(544)&Q(518)&Q(492)&Q(466)&Q(440)&Q(414)&Q(388)
)&Q(362)&Q(336)&Q(310)&Q(284)&Q(258)&Q(232)&Q(206)&Q(180)&Q(154)&Q(128)&Q(102)&Q(76)
)&Q(50)&Q(24);
QV25 <=
Q(675)&Q(649)&Q(623)&Q(597)&Q(571)&Q(545)&Q(519)&Q(493)&Q(467)&Q(441)&Q(415)&Q(389)
)&Q(363)&Q(337)&Q(311)&Q(285)&Q(259)&Q(233)&Q(207)&Q(181)&Q(155)&Q(129)&Q(103)&Q(77)
)&Q(51)&Q(25);

```

```

--Finite state machine for encoder3226-----
FSM: process(clk,reset,start)
begin
  if reset='0' or start='0' then      --low level active:
    state <= idle;
  elsif clk'event and clk='1' then
    case state is
      when idle => if start = '1' then
        state <= receive;
      else
        state <= idle;
      end if;

      when receive => if countup < "1010100011" then --shift number=675;
        state <= receive;
      else
        state <= encode;
      end if;

      when encode => if jump < "11001" then -- codeword number<25;
        state <= encode;
      end if;
    end case;
  end if;
end process;

```

```

        elsif jump="11001" and addr="101" then
            state <= shift_out;
        end if;

    when shift_out => if count < "1111011011" then -- coded word shift
        -- out <987="1111011011";
        state <= shift_out;
    else
        state <= receive;
    end if;

    when others => null;
end case;
end if;
end process;

R_shift <= '1' when state = receive else '0';
encoding <= '1' when state = encode else '0';
shifting_out <= '1' when state = shift_out else '0';

--input buffer RIGHT shift process, used for HORIZONTAL and VERTICAL ENCODING-----
process(clk.reset)
begin
    if reset='0' then
        Q <= (others=>'0');           --asynchronous clear register of input sequence;
    elsif clk'event and clk='1' then
        if R_shift='1' or (shifting_out='1' and count <"1010100100")then --shift enable "1010100100"=676;
            Q(bufferize-1) <= input;
            for i in bufferize-1 downto 1 loop
                Q(i-1) <= Q(i);         --R-shift register one bit/clock rise-edge;
            end loop; -- i
        end if;
    end if;
end process;

-----shifting number counter process-----
num1: process (clk, reset)
begin -- process count
    if reset = '0' then                -- asynchronous reset (active low)
        countup <= (others => '0');
    elsif clk'event and clk = '1' then -- rising clock edge
        if R_shift = '1' then
            if countup < "1010100011" then --shift number<675;
                countup <= countup + 1;
            else
                countup <= (others => '0');
            end if;
        end if;
    end if;
end process num1;

----Generate address of G metrix for both HORIZONTAL and VERTICAL ENCODING-----
process(clk,reset)
begin

```

```

if reset = '0' then          --initiate address value
    addr <= "000";
elsif clk'event and clk='1' then
    if encoding='1' then
        if addr < "101" then    --G' addr < 5;
            addr <= addr+1;
        else
            addr <= "000";
        end if;
    end if;
end if;
end process;

-----counting the switching number for each codeword encoding-----
num2: process (clk, reset)
begin -- process num2
    if reset = '0' then        -- asynchronous reset (active low)
        jump <= (others => '0');
    elsif clk'event and clk = '1' then -- rising clock edge
        if addr = "101" then
            if jump < "11001" then    -- codeword switching number j < 25;
                jump <= jump + 1;
            else
                jump <= (others => '0');
            end if;
        end if;
    end if;
end process num2;

-- purpose: delay jump signal to synchronize with G output
process (clk, reset)
begin -- process
    if reset = '0' then        -- asynchronous reset (active low)
        jump_delay <= (others=>'0');
    elsif clk'event and clk = '1' then -- rising clock edge
        jump_delay <= jump;
    end if;
end process;

-----
genmatrix : reg_gen3226 port map (
    addr => addr,
    clk => clk,
    dout => g);    --26(word width)x6(word number)
                  --generator matrix;

mux_H: process
(jump_delay,QH0,QH1,QH2,QH3,QH4,QH5,QH6,QH7,QH8,QH9,QH10,QH11,QH12,QH13,QH14,QH15
,QH16,QH17,QH18,QH19,QH20,QH21,QH22,QH23,QH24,QH25)
begin -- process mux_H
    case jump_delay is
        when "00000" => mux_out_H <= QH0;
        when "00001" => mux_out_H <= QH1;
        when "00010" => mux_out_H <= QH2;
    end case;
end process;

```

```

when "00011" => mux_out_H <= QH3;
when "00100" => mux_out_H <= QH4;
when "00101" => mux_out_H <= QH5;
when "00110" => mux_out_H <= QH6;
when "00111" => mux_out_H <= QH7;
when "01000" => mux_out_H <= QH8;
when "01001" => mux_out_H <= QH9;
when "01010" => mux_out_H <= QH10;
when "01011" => mux_out_H <= QH11;
when "01100" => mux_out_H <= QH12;
when "01101" => mux_out_H <= QH13;
when "01110" => mux_out_H <= QH14;
when "01111" => mux_out_H <= QH15;
when "10000" => mux_out_H <= QH16;
when "10001" => mux_out_H <= QH17;
when "10010" => mux_out_H <= QH18;
when "10011" => mux_out_H <= QH19;
when "10100" => mux_out_H <= QH20;
when "10101" => mux_out_H <= QH21;
when "10110" => mux_out_H <= QH22;
when "10111" => mux_out_H <= QH23;
when "11000" => mux_out_H <= QH24;
when "11001" => mux_out_H <= QH25;
when others => null;
end case;
end process mux_H;

mux_V: process (jump_delay,QV0, QV1, QV2, QV3, QV4, QV5, QV6, QV7, QV8, QV9,QV10, QV11,
QV12, QV13, QV14, QV15, QV16, QV17, QV18,QV19, QV20, QV21, QV22, QV23,QV24, QV25)
begin -- process mux_V
case jump_delay is
when "00000" => mux_out_V <= QV0;
when "00001" => mux_out_V <= QV1;
when "00010" => mux_out_V <= QV2;
when "00011" => mux_out_V <= QV3;
when "00100" => mux_out_V <= QV4;
when "00101" => mux_out_V <= QV5;
when "00110" => mux_out_V <= QV6;
when "00111" => mux_out_V <= QV7;
when "01000" => mux_out_V <= QV8;
when "01001" => mux_out_V <= QV9;
when "01010" => mux_out_V <= QV10;
when "01011" => mux_out_V <= QV11;
when "01100" => mux_out_V <= QV12;
when "01101" => mux_out_V <= QV13;
when "01110" => mux_out_V <= QV14;
when "01111" => mux_out_V <= QV15;
when "10000" => mux_out_V <= QV16;
when "10001" => mux_out_V <= QV17;
when "10010" => mux_out_V <= QV18;
when "10011" => mux_out_V <= QV19;
when "10100" => mux_out_V <= QV20;
when "10101" => mux_out_V <= QV21;
when "10110" => mux_out_V <= QV22;

```

```

when "10111" => mux_out_V <= QV23;
when "11000" => mux_out_V <= QV24;
when "11001" => mux_out_V <= QV25;
when others => null;
end case;
end process mux_V;

```

```

--control signal delay process-----
delay1: process (clk, reset)
begin -- process delay
if reset = '0' then          -- asynchronous reset (active low)
    encol <= '0';
elsif clk'event and clk = '1' then -- rising clock edge
    encol <= encoding;
end if;
end process delay1;

```

```

-----HORIZONTAL ENCODING process -----
m_out_H <= mux_out_H and g;
loop1: for i in 0 to 24 generate
    bit1: if i=0 generate
        xor_out_H(i+1) <= m_out_H(i) xor m_out_H(i+1);
    end generate bit1;
    other: if i/=0 generate
        xor_out_H(i+1) <= m_out_H(i+1) xor xor_out_H(i);
    end generate other;
end generate loop1;

```

```

--output parity bits buffer for HORIZONTAL ENCODING-----
process(clk.reset)
begin
if reset = '0' then          --active low;
    paritybit_H <= (others => '0');
elsif clk'event and clk='1' then
    if encol = '1' or (shifting_out = '1' and count < "1101000000" and count>="1010100100") then --just
shift 156 times
        paritybit_H(paritysize-1)<=xor_out_H(bitwidth-1);
        for i in paritysize-1 downto 1 loop
            paritybit_H(i-1) <= paritybit_H(i);
        end loop; -- i
    end if;
end if;
end process;

```

```

-----VERTICAL ENCODING process -----
m_out_V <= mux_out_V and g;
loop2: for i in 0 to 24 generate
    bit1: if i=0 generate
        xor_out_V(i+1) <= m_out_V(i) xor m_out_V(i+1);
    end generate bit1;
    other: if i/=0 generate
        xor_out_V(i+1) <= m_out_V(i+1) xor xor_out_V(i);
    end generate other;
end generate loop2;

```

```
--output parity bits buffer for VERTICAL ENCODING-----
process(clk,reset)
begin
    if reset = '0' then          --active low;
        paritybit_V <= (others => '0');
    elsif clk'event and clk='1' then
        if encol = '1' or (shifting_out = '1' and count >= "1101000000" and count < "1111011100") then
--just shift 156 times
            paritybit_V(paritysize-1)<=xor_out_V(bitwidth-1);
            for i in paritysize-1 downto 1 loop
                paritybit_V(i-1) <= paritybit_V(i);
            end loop; -- i
        end if;
    end if;
end process;

-- output coded codeword serially-----
process (clk, reset)
begin -- process
    if reset = '0' then          -- asynchronous reset (active low)
        count <= (others=>'0');
    elsif clk'event and clk = '1' then -- rising clock edge
        if shifting_out = '1' then
            if count < "1111011011" then
                count <= count + 1;
            else
                count <= (others=>'0');
            end if;
        end if;
    end if;
end process;

codeword <= Q(0) when count < "1010100100" else -- <676
    paritybit_H(0) when count>="1010100100" and count<"1101000000" else -- 676<= count< 832
    paritybit_V(0) when count>="1101000000" and count<"1111011100" else -- <988
    '0';
end rtl;
```

File name : encoder_c_cfg.vhd
 Purpose: configure file of encoder_c.vhd

library Xilinxcorelib;
 use Xilinxcorelib.all;

configuration cfg_encoder_c of encoder_c is

```
    for rtl
        for all : reg_gen3226
            use configuration work.cfg_reg_gen3226;
        end for;
    end for;
```

```
end cfg_encoder_c;
```

```
File name: reg_gen3226.vhd
```

```
Purpose: parity vectors of generator matrix. ROM memory generated by Coregen.
```

```
library ieee;
```

```
use ieee.std_logic_1164.all;
```

```
entity reg_gen3226 is
```

```
port (
```

```
  addr : in std_logic_vector(2 downto 0); -- memory addressing
```

```
  clk : in std_logic;
```

```
  dout : out std_logic_vector(25 downto 0)); -- matrix vector output
```

```
end reg_gen3226;
```

```
architecture rtl of reg_gen3226 is
```

```
  component gen3226
```

```
  port (
```

```
    addr: IN std_logic_VECTOR(2 downto 0);
```

```
    clk : IN std_logic;
```

```
    dout: OUT std_logic_VECTOR(25 downto 0));
```

```
  end component;
```

```
begin -- rtl
```

```
  gen_matrix: gen3226
```

```
    port map (
```

```
      addr => addr,
```

```
      clk => clk,
```

```
      dout => dout);
```

```
end rtl;
```

```
File name: reg_gen3226_cfg.vhd
```

```
Purpose: configure file of reg_gen3226_cfg.vhd
```

```
Library XilinxCoreLib;
```

```
use XilinxCoreLib.all;
```

```
configuration cfg_reg_gen3226 of reg_gen3226 is
```

```
for rtl
```

```
-- synopsys translate_off
```

```
for all : gen3226 use entity XilinxCoreLib.blkmemsp_v3_2(behavioral)
```

```
generic map(
```

```
  c_has_en => 0,
```

```
  c_has_din => 0,
```

```
  c_has_limit_data_pitch => 0,
```

```
  c_has_sinit => 0,
```

```
  c_limit_data_pitch => 8,
```

```
c_width => 26,
c_sinit_value => "0",
c_addr_width => 3,
c_has_rfd => 0,
c_has_we => 0,
c_depth => 6,
c_write_mode => 0,
c_pipe_stages => 0,
c_has_nd => 0,
c_default_data => "0",
c_has_default_data => 0,
c_mem_init_file => "gen3226.mif",
c_reg_inputs => 0,
c_enable_rlocs => 0,
c_has_rdy => 0);

end for;
-- synopsys translate_on

end for;
end cfg_reg_gen3226;
```

Appendix C Turbo decoder VHDL source codes

File name: decoder.vhd

Purpose: turbo decoder main program

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity decoder is
  port (
    clk0    : in std_logic;
    clk1    : in std_logic;
    reset   : in std_logic;
    start   : in std_logic;          -- starting signal of decoding;
    SNR     : in std_logic_vector(3 downto 0); -- presetting SNR value;
    iter_number: in std_logic_vector(3 downto 0); -- presetting iteration number of decoding;
    yk_in   : in std_logic_vector(3 downto 0); -- 4-bit quantitative input to decoder from channel;
    deco_done : out std_logic;          -- one block decoding finishing signal;
    infobit  : out std_logic);          -- estimated info. code output;
end decoder;
```

architecture rtl of decoder is

```
  component memyktop          -- buffer component
  port (
    addr : IN std_logic_VECTOR(9 downto 0);
    clk  : IN std_logic;
    yk_in : IN std_logic_VECTOR(3 downto 0);
    yk_out: OUT std_logic_VECTOR(3 downto 0);
    we   : IN std_logic);
  end component;

  component memlctop          --
  port (
    clk : in std_logic;
    addr : in std_logic_vector(3 downto 0);
    Lc   : out std_logic_vector(7 downto 0));
  end component;

  component lcykunit          -- multiplier component
  port (
    Lc : in std_logic_vector(7 downto 0);
    yk : in std_logic_vector(3 downto 0);
    Lcyk : out std_logic_vector(11 downto 0));
  end component;
```

component memletop

```

port (
  addra : IN std_logic_VECTOR(9 downto 0); -- A port for reading out;
  addrb : IN std_logic_vector(9 downto 0); -- B port for writing in;
  clka : IN std_logic;
  clkb : IN std_logic;
  Le_in : IN std_logic_VECTOR(11 downto 0);
  web : IN std_logic;
  Le_out : OUT std_logic_VECTOR(11 downto 0));

```

end component;

component memlcyktop

```

port (
  addr : in std_logic_VECTOR(9 downto 0);
  clk : in std_logic;
  we : in std_logic;
  lcyk_in : in std_logic_VECTOR(11 downto 0);
  lcyk_out: out std_logic_VECTOR(11 downto 0));

```

end component;

component gamma -- gamma calculation component

```

port (
  lcyk_out : in std_logic_vector(11 downto 0);
  le_out : in std_logic_vector(11 downto 0);
  addr1 : in std_logic_vector(9 downto 0); -- addr for judge of Prity bit.
  sum : out std_logic_vector(12 downto 0); -- for LLR in final decision;
  gamma1 : out std_logic_vector(11 downto 0); -- corresponding to Xk=1;
  gamma0 : out std_logic_vector(11 downto 0)); -- corresponding to Xk=-1;

```

end component;

component memlcykletop -- component for storing Lcyk+Le

```

port (
  addr : in std_logic_vector(9 downto 0);
  clk : in std_logic;
  din : in std_logic_vector(12 downto 0);
  dout : out std_logic_vector(12 downto 0);
  we : in std_logic);

```

end component;

component memloggammatop

```

port (
  addra: IN std_logic_VECTOR(9 downto 0);
  addrb: IN std_logic_VECTOR(9 downto 0);
  clka : IN std_logic;
  clkb : IN std_logic;
  dina : IN std_logic_VECTOR(11 downto 0);
  douta: OUT std_logic_VECTOR(11 downto 0);
  doutb: OUT std_logic_VECTOR(11 downto 0);
  wea : IN std_logic);

```

end component;

component memalphabetatop -- A port for reading out, B for writing in

```

port (
  addra : in std_logic_vector(9 downto 0);

```

```

addrb : in std_logic_vector(9 downto 0);
clka  : in std_logic;
clkb  : in std_logic;
dinb  : in std_logic_vector(16 downto 0);
douta : out std_logic_vector(16 downto 0);
doutb : out std_logic_vector(16 downto 0);
web   : in std_logic;
end component;

component alphabeta          -- alphabeta unit component
port (
  logalpha_in1 : in std_logic_vector(16 downto 0);
  logalpha_in0 : in std_logic_vector(16 downto 0);
  loggamma0_in1 : in std_logic_vector(11 downto 0);
  loggamma0_in0 : in std_logic_vector(11 downto 0);
  logbeta_in1  : in std_logic_vector(16 downto 0);
  logbeta_in0  : in std_logic_vector(16 downto 0);
  loggamma1_in1 : in std_logic_vector(11 downto 0);
  loggamma1_in0 : in std_logic_vector(11 downto 0);
  mux3sel      : in std_logic;
  mux4sel      : in std_logic;
  mux5sel      : in std_logic;
  mux6sel      : in std_logic;
  clk          : in std_logic;
  reset        : in std_logic;
  logalpha_out : out std_logic_vector(16 downto 0);
  logbeta_out  : out std_logic_vector(16 downto 0));
end component;

component leunit            -- Leunit component
port (
  logalpha0 : in std_logic_vector(16 downto 0);
  logalpha1 : in std_logic_vector(16 downto 0);
  logbeta0  : in std_logic_vector(16 downto 0);
  logbeta1  : in std_logic_vector(16 downto 0);
  clk       : in std_logic;
  reset     : in std_logic;
  mux0_1sel : in std_logic;
  mux4_5sel : in std_logic;
  leunit_out : out std_logic_vector(16 downto 0));
end component;

component LLR_unit          -- harddecision LLR_unit component
port (
  le_h : in std_logic_vector(12 downto 0);
  le_v : in std_logic_vector(11 downto 0);
  infobit : out std_logic);
end component;

component FSM_buf           -- FSM and memory address generator component
port (
  start      : in std_logic;
  reset      : in std_logic;
  clk0       : in std_logic; -- channel clock

```

```

clk1      : in std_logic; -- decoder clock
iter_number : in std_logic_vector(3 downto 0);
SNR       : in std_logic_vector(3 downto 0); -- presetting SNR value;
buffer_sel : out std_logic;
deco_done  : out std_logic; -- one block decoding finishing signal;
mux3sel    : out std_logic;
mux4sel    : out std_logic;
mux5sel    : out std_logic;
mux6sel    : out std_logic;
mux0_1sel  : out std_logic;
mux4_5sel  : out std_logic;
we_memyk1  : out std_logic;
we_memyk2  : out std_logic;
we_memlcyk : out std_logic;
web_memle  : out std_logic;
we_memlcykle : out std_logic;
wea_memgamma : out std_logic;
web_memalphabet : out std_logic;
Le_in_sel  : out std_logic;
addr_memyk1 : out std_logic_vector(9 downto 0);
addr_memyk2 : out std_logic_vector(9 downto 0);
addr_memlc  : out std_logic_vector(3 downto 0);
addra_memle : out std_logic_vector(9 downto 0);
addrb_memle : out std_logic_vector(9 downto 0);
addr_memlcyk : out std_logic_vector(9 downto 0);
addr_memlcykle : out std_logic_vector(9 downto 0);
addra_memgamma : out std_logic_vector(9 downto 0); -- memgamma reading out addr. A port for
alpha;
addrb_memgamma : out std_logic_vector(9 downto 0); -- memgamma reading out addr. B port for
beta ;
addrb_memalphabet : out std_logic_vector(9 downto 0); -- memalpha and membeta write back addr.
addra_memalpha0 : out std_logic_vector(9 downto 0); -- memalpha0 read out addr.
addra_memalpha1 : out std_logic_vector(9 downto 0); -- memalpha1 read out addr.
addra_membeta0 : out std_logic_vector(9 downto 0); -- membeta0 read out addr.
addra_membeta1 : out std_logic_vector(9 downto 0); -- membeta0 read out addr.
end component;

signal memyk1_out, memyk2_out, yk_out : std_logic_vector(3 downto 0);
signal lc_out : std_logic_vector(7 downto 0);
signal Lcykunit_Out : std_logic_vector(11 downto 0);
signal memlcyk_out : std_logic_vector(11 downto 0);
signal le_in, le_out : std_logic_vector(11 downto 0);
signal gamma0, gamma1 : std_logic_vector(11 downto 0);
signal memgamma0_douta, memgamma0_doutb, memgamma1_douta, memgamma1_doutb :
std_logic_vector(11 downto 0);
signal addra_memgamma, addrb_memgamma : std_logic_vector(9 downto 0);
signal addrb_memalphabet : std_logic_vector(9 downto 0);
signal addra_memalpha0, addra_memalpha1, addra_membeta0, addra_membeta1 : std_logic_vector(9
downto 0);
signal memalpha1_dinb : std_logic_vector(16 downto 0);
signal memalpha0_douta, memalpha0_doutb, memalpha1_douta : std_logic_vector(16 downto 0);
signal membeta1_dinb : std_logic_vector(16 downto 0);
signal membeta0_douta, membeta0_doutb, membeta1_douta, membeta1_doutb : std_logic_vector(16 -
downto 0);

```

```

signal alpha0_in,beta0_in,beta1_in : std_logic_vector(16 downto 0);
signal addr_memlc : std_logic_vector(3 downto 0);
signal addr_memyk1, addr_memyk2 : std_logic_vector(9 downto 0);
signal addra_memle, addrb_memle : std_logic_vector(9 downto 0);
signal addr_memlcyk : std_logic_vector(9 downto 0);
signal addr_memlcykle : std_logic_vector(9 downto 0);
signal we_memyk1,we_memyk2,we_memlcyk,web_memle,we_memlcykle, web_memalphabeta,
wea_memgamma : std_logic;
signal mux3sel, mux4sel, mux5sel, mux6sel : std_logic;
signal mux0_1sel, mux4_5sel : std_logic;
signal Le_in_sel : std_logic;
signal leunit_out : std_logic_vector(16 downto 0);
signal sum,memlcykle_out : std_logic_vector(12 downto 0);
signal buffer_sel : std_logic;
signal clk_memyk1, clk_memyk2 : std_logic;

```

```
begin -- rtl
```

```

machine : FSM_buf port map (
  start      => start,
  reset      => reset,
  clk0       => clk0,
  clk1       => clk1,
  iter_number  => iter_number,
  SNR        => SNR,
  buffer_sel  => buffer_sel,
  deco_done  => deco_done,
  mux3sel    => mux3sel,
  mux4sel    => mux4sel,
  mux5sel    => mux5sel,
  mux6sel    => mux6sel,
  mux0_1sel  => mux0_1sel,
  mux4_5sel  => mux4_5sel,
  we_memyk1  => we_memyk1,
  we_memyk2  => we_memyk2,
  we_memlcyk => we_memlcyk,
  web_memle  => web_memle,
  we_memlcykle => we_memlcykle,
  wea_memgamma  => wea_memgamma,
  web_memalphabeta => web_memalphabeta,
  Le_in_sel  => Le_in_sel,
  addr_memyk1  => addr_memyk1,
  addr_memyk2  => addr_memyk2,
  addr_memlc   => addr_memlc,
  addra_memle  => addra_memle,
  addrb_memle  => addrb_memle,
  addr_memlcyk  => addr_memlcyk,
  addr_memlcykle  => addr_memlcykle,
  addra_memgamma  => addra_memgamma,
  addrb_memgamma  => addrb_memgamma,
  addrb_memalphabeta => addrb_memalphabeta,
  addra_memalpha0  => addra_memalpha0,
  addra_memalpha1  => addra_memalpha1,
  addra_membeta0  => addra_membeta0,

```

```

addr_membeta1    => addr_membeta1);

clk_memyk1 <= clk0 when buffer_sel='1' else clk1;
clk_memyk2 <= clk1 when buffer_sel='1' else clk0;

memorylc : memlctop port map (
    addr => addr_memlc,
    clk  => clk1,
    Lc   => lc_out);

memyk1 : memyktop port map (
    addr  => addr_memyk1,
    clk   => clk_memyk1,
    yk_in => yk_in,
    yk_out => memyk1_out,
    we    => we_memyk1);

memyk2 : memyktop port map (
    addr  => addr_memyk2,
    clk   => clk_memyk2,
    yk_in => yk_in,
    yk_out => memyk2_out,
    we    => we_memyk2);

yk_out <= memyk2_out when buffer_sel='1' else memyk1_out; -- select buffer output;

multiplier : lcykunit port map (
    Lc   => lc_out,
    yk   => yk_out,
    Lcyk => lcykunit_out);

memorylcyk : memlcyktop port map (
    addr  => addr_memlcyk,
    clk   => clk1,
    lcyk_in => lcykunit_out,
    lcyk_out => memlcyk_out,
    we    => we_memlcyk);

memoryle : memletop port map (
    addra => addra_memle,      -- A port for reading out;
    addrb => addrb_memle,      -- B port for writing in;
    clka  => clk1,
    clkb  => clk1,
    Le_in => le_in,
    Le_out => le_out,
    web   => web_memle);

ga : gamma port map (          -- log-gamma calculation unit;
    lcyk_out => memlcyk_out,
    le_out  => le_out,
    addr1   => addra_memle,      -- address value for judging jump
                                   -- during calculation of log-gamma;
    sum     => sum,
    gamma0  => gamma0,

```

```
gammal => gammal);
```

```
memorylcykle : memlcykletop port map (  
  addr => addr_memlcykle,  
  clk  => clk1,  
  din  => sum,  
  dout => memlcykle_out,  
  we   => we_memlcykle);
```

```
memgamma0 : memloggamma0top port map ( -- for xk=-1, gamma state:A write  
  -- in; alphabeta state: A and B port  
  -- read out for alpha and beta calculation;  
  addra => addra_memgamma,  
  addrb => addrb_memgamma,  
  clka  => clk1,  
  clkb  => clk1,  
  dina  => gamma0,  
  douta => memgamma0_douta,  
  doutb => memgamma0_doutb,  
  wea   => wea_memgamma);
```

```
memgamma1 : memloggamma0top port map ( -- for xk=1, gamma state:A write  
  -- in; alphabeta state: A and B port  
  -- read out for alpha and beta calculation;  
  addra => addra_memgamma,  
  addrb => addrb_memgamma,  
  clka  => clk1,  
  clkb  => clk1,  
  dina  => gammal,  
  douta => memgamma1_douta,  
  doutb => memgamma1_doutb,  
  wea   => wea_memgamma);
```

--B port of memalpha0 and memalpha1 are same for both write in with same address and data.

```
memalpha0 : memalphabetatop port map ( -- for xk=-1  
  addra => addra_memalpha0,  
  addrb => addrb_memalphabeta,  
  clka  => clk1,  
  clkb  => clk1,  
  dinb  => memalpha1_dinb,      -- write in at port B  
  douta => memalpha0_douta,     -- read out at port A for xk=-1  
  doutb => memalpha0_doutb,  
  web   => web_memalphabeta);
```

```
memalpha1 : memalphabetatop port map ( -- for xk=1  
  addra => addra_memalpha1,  
  addrb => addrb_memalphabeta,  
  clka  => clk1,  
  clkb  => clk1,  
  dinb  => memalpha1_dinb,      -- write in at port B  
  douta => memalpha1_douta,     -- read out at port A for xk=1  
  doutb => open,  
  web   => web_memalphabeta);
```

```

--B port of memalpha0 and membeta1 are same for both write in with same
--address and data. So memalpha0_wea and membeta1_wea = '0'.
membeta0 : memalphabetatop port map ( -- for xk=-1
  addra => addra_membeta0,
  addrb => addrb_memalphabet,
  clka  => clk1,
  clkb  => clk1,
  dinb  => membeta1_dinb,      -- write in at port B
  douta => membeta0_douta,     -- read out at port A for xk=-1
  doutb => membeta0_doutb,
  web   => web_memalphabet);

membeta1 : memalphabetatop port map ( -- for xk=1
  addra => addra_membeta1,
  addrb => addrb_memalphabet,
  clka  => clk1,
  clkb  => clk1,
  dinb  => membeta1_dinb,      -- write in at port B
  douta => membeta1_douta,     -- read out at port A for xk=1
  doutb => membeta1_doutb,
  web   => web_memalphabet);

process (addrb_memalphabet,memalpha0_douta,
memalpha0_doutb,membeta1_douta,membeta1_doutb,membeta0_douta,membeta0_doutb)
begin
  -- solve the data hazards by forwarding value
  if addrb_memalphabet="0000000010" then
    alpha0_in <= memalpha0_doutb;
    beta1_in  <= membeta1_doutb;
  else
    alpha0_in <= memalpha0_douta;
    beta1_in  <= membeta1_douta;
  end if;

  if addrb_memalphabet="1001111011" then -- 635="1001111011"
    beta0_in <= membeta0_doutb;
  else
    beta0_in <= membeta0_douta;
  end if;
end process;

alfa_beta : alphabeta port map (
  logalpha_in1 => memalpha1_douta,--alpha1_in,--memalpha1_douta,
  logalpha_in0 => alpha0_in,      --memalpha0_douta,avoiding write/read conflict;
  loggamma0_in1 => memgamma0_douta,
  loggamma0_in0 => memgamma0_doutb,
  logbeta_in1  => beta1_in,--membeta1_douta,
  logbeta_in0  => beta0_in,--membeta0_douta,
  loggamma1_in1 => memgamma1_douta,
  loggamma1_in0 => memgamma1_doutb,
  mux3sel      => mux3sel,
  mux4sel      => mux4sel,
  mux5sel      => mux5sel,
  mux6sel      => mux6sel,
  clk          => clk1,

```

```

reset      => reset,
logalpha_out => memalpha1_dinb,
logbeta_out  => membeta1_dinb);

le : leunit port map (
  logalpha0 => memalpha0_douta,
  logalpha1 => memalpha1_douta,
  logbeta0  => membeta0_douta,
  logbeta1  => membeta1_douta,
  clk       => clk1,
  reset     => reset,
  mux0_1sel => mux0_1sel,
  mux4_5sel => mux4_5sel,
  leunit_out => leunit_out);

memleinput: process (leunit_out,Le_in_sel) -- for clear memle to ZERO
-- whenever receive new block.
begin -- process memleinput
  if Le_in_sel = '0' then
    le_in <= leunit_out(16)&leunit_out(10 downto 0);
  else
    le_in <= (others => '0');
  end if;
end process memleinput;

llr : LLR_unit port map (
  le_h  => memlcycle_out,
  le_v  => le_out,
  infobit => infobit);

end rtl;

```

File name: decoder_cfg.vhd

Purpose: configure file of decoder.vhd

library xilinxcorelib;

use xilinxcorelib.all;

configuration cfg_decoder of decoder is

```

for rtl
  for all : memyktop
    use configuration work.cfg_memyktop;
  end for;

  for all : memlctop
    use configuration work.cfg_memlctop;
  end for;

  for all : lcykunit
    use configuration work.cfg_lcykunit;
  end for;

```

```
for all : memletop
  use configuration work.cfg_memletop;
end for;

for all : memlcyktop
  use configuration work.cfg_memlcyktop;
end for;

for all : gamma
  use configuration work.cfg_gamma;
end for;

for all : memlcykletop
  use configuration work.cfg_memlcykletop;
end for;

for all : memloggammatop
  use configuration work.cfg_memloggammatop;
end for;

for all : memalphabetatop
  use configuration work.cfg_memalphabetatop;
end for;

for all : alphabeta
  use entity work.alphabeta(rtl);
end for;

for all : leunit
  use entity work.leunit(rtl);
end for;

for all : LLR_unit
  use entity work.LLR_unit(rtl);
end for;

for all : FSM_buf
  use configuration work.cfg_FSM_buf;
end for;

end for;
end cfg_decoder;
```

File name: lcykunit.vhd
Purpose: pre-processing multiplication of Lc and Yk

```
library ieee;
use ieee.std_logic_1164.all;
```

```
--unit for multiplication of lc and yk,lc's format is in sign+XXXX+.XXX(8 bit),
--yk is in sign+xxx(4 bit)
```

```
entity lcykunit is
```

```
port (
  lc : in std_logic_vector(7 downto 0);
  yk : in std_logic_vector(3 downto 0);
  lcyk : out std_logic_vector(11 downto 0));
```

end lcykunit;

architecture rtl of lcykunit is

component multilcyktop

```
port (
  Lc : in std_logic_vector(7 downto 0);
  yk : in std_logic_vector(3 downto 0);
  Lcyk : out std_logic_vector(11 downto 0));
end component;
```

begin -- rtl

```
multi : multilcyktop port map (
  Lc => lc,
  yk => yk,
  Lcyk => lcyk);
```

end rtl;

File name: lcykunit_cfg.vhd

Purpose: configure file of lcykunit.vhd

```
library xilinxcorelib;
use xilinxcorelib.all;
```

configuration cfg_lcykunit of lcykunit is

```
for rtl
  for all : multilcyktop
    use configuration work.cfg_multilcyktop;
  end for;
end for;
```

end cfg_lcykunit;

File name: gamma.vhd

Purpose: branch transition calculation unit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
--unit for gamma calculation, le_out from memle in sign+XXXXXXXXX+XXX(12 bits), lcyk_out from
--memlyk in sign+XXXXXXXXX+XXX(12 bits);outputis in 13 bit(sign+XXXXXXXXXX.XXX)
```

entity gamma is

```
port (
  lcyk_out : in std_logic_vector(11 downto 0);
  le_out   : in std_logic_vector(11 downto 0);
  addr1    : in std_logic_vector(9 downto 0);
  sum      : out std_logic_vector(12 downto 0); -- for LLR unit in final decision;
  gamma1   : out std_logic_vector(11 downto 0); -- corresponding to Xk=1;
  gamma0   : out std_logic_vector(11 downto 0)); -- corresponding to Xk=-1;
```

end gamma;

architecture rtl of gamma is

component addergammatop

```
port (
  A : in std_logic_VECTOR(11 downto 0);
  B : in std_logic_VECTOR(11 downto 0);
  S : out std_logic_VECTOR(12 downto 0));
end component;
```

signal adder_out : std_logic_vector(12 downto 0); --13 bits format "sign+XXXXXXXXXX+.XXX";

begin -- rtl

add : addergammatop port map (

```
  A => le_out,
  B => lcyk_out,
  S => adder_out);
```

sum <= adder_out; -- directly for LLR unit;

pr1: process (addr1, adder_out, lcyk_out)

variable mux_out : std_logic_vector(11 downto 0);

begin -- process pr1

if addr1 < "1010100100" then -- 676="1010100100"

mux_out := adder_out(12 downto 1); --shift right one bit, 1/2 function;

else

mux_out := lcyk_out(11)&lcyk_out(11 downto 1); --halfcyk; --sign extend to 11 bits width;

end if;

gamma1 <= mux_out;

gamma0 <= ("111111111111" xor mux_out) + 1; --two's complement;

end process pr1;

end rtl;

File name: gamma_cfg.vhd

Purpose: configure file of gamma.vhd

library XilinxCoreLib;

use XilinxCoreLib.all;

configuration cfg_gamma of gamma is

```

for rtl

  for all : addergammatop
    use configuration work.cfg_addergammatop;
  end for;

end for;

end cfg_gamma;

```

File name: alphabeta.vhd

Purpose: forward and backward recursion calculation unit

library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_signed.all;

entity alphabeta is

```

port (
  logalpha_in1 : in std_logic_vector(16 downto 0); -- memalpha1-A port;
  logalpha_in0 : in std_logic_vector(16 downto 0); -- memalpha0-A port;
  logbeta_in1  : in std_logic_vector(16 downto 0); -- membeta1-A port;
  logbeta_in0  : in std_logic_vector(16 downto 0); -- membeta0-A port;
  loggamma0_in1 : in std_logic_vector(11 downto 0); -- memgamma0-A port;for alpha part;
  loggamma0_in0 : in std_logic_vector(11 downto 0); -- memgamma0-B port;for beta part
  loggamma1_in1 : in std_logic_vector(11 downto 0); -- memgamma1-A port;for alpha part
  loggamma1_in0 : in std_logic_vector(11 downto 0); -- memgamma1-B port;for beta part
  mux3sel       : in std_logic;
  mux4sel       : in std_logic;
  mux5sel       : in std_logic;
  mux6sel       : in std_logic;
  clk           : in std_logic;
  reset         : in std_logic;
  logalpha_out  : out std_logic_vector(16 downto 0);
  logbeta_out   : out std_logic_vector(16 downto 0));

```

end alphabeta;

architecture rtl of alphabeta is

```

  signal mux1_out,mux2_out,mux3_out,mux4_out,mux5_out,mux6_out : std_logic_vector(16 downto 0);
  signal Dif_a, Dif_b : std_logic_vector(16 downto 0);
  signal Reg1, Reg2, Reg3, Reg4, Reg_difa, Reg_difb : std_logic_vector(16 downto 0);-- pipeline registers;
  signal Reg_reset : std_logic;
  signal adder1_out,adder2_out,adder3_out,adder4_out,adder5_out,adder6_out : std_logic_vector(16
downto 0);
  signal corvalue_a, corvalue_b : std_logic_vector(4 downto 0);

```

begin -- rtl

```

add1_4: process
(logalpha_in1,loggamma0_in1,logalpha_in0,loggamma0_in0,logbeta_in1,logbeta_in0,loggamma1_in1,log
gamma1_in0)
begin
  adder1_out <= logalpha_in1 + loggamma1_in1;
  adder2_out <= logalpha_in0 + loggamma0_in1;
  adder3_out <= logbeta_in1 + loggamma1_in0;
  adder4_out <= logbeta_in0 + loggamma0_in0;
end process add1_4;

mux3_4: process (mux3sel, mux4sel, adder1_out, adder2_out, adder3_out, adder4_out)
begin -- process mux3
  if mux3sel = '1' then
    mux3_out <= adder1_out;
  else
    mux3_out <= adder2_out;
  end if;

  if mux4sel = '1' then
    mux4_out <= adder3_out;
  else
    mux4_out <= adder4_out;
  end if;
end process mux3_4;

comp_max_sub: process (adder1_out, adder2_out, adder3_out, adder4_out)
begin -- process comp_max_sub
  if adder1_out >= adder2_out then
    mux1_out <= adder1_out;
    Dif_a <= adder1_out - adder2_out;
  else
    mux1_out <= adder2_out;
    Dif_a <= adder2_out - adder1_out;
  end if;

  if adder3_out >= adder4_out then
    mux2_out <= adder3_out;
    Dif_b <= adder3_out - adder4_out;
  else
    mux2_out <= adder4_out;
    Dif_b <= adder4_out - adder3_out;
  end if;
end process comp_max_sub;

reset_delay: process (clk, reset)
begin -- process reset_delay
  if reset = '0' then -- asynchronous reset (active low)
    Reg_reset <= '0';
  elsif clk'event and clk = '1' then -- rising clock edge
    Reg_reset <= reset;
  end if;
end process reset_delay;

```

```

pipeline1: process (clk, Reg_reset)
begin -- process pipeline1
  if Reg_reset = '0' then          -- asynchronous reset (active low)
    Reg1 <= (others => '0');
    Reg3 <= (others => '0');
    Reg_difa <= (others => '0');

    Reg2 <= (others => '0');
    Reg4 <= (others => '0');
    Reg_difb <= (others => '0');
  elsif clk'event and clk = '1' then -- rising clock edge
    Reg1 <= mux1_out;
    Reg3 <= mux3_out;
    Reg_difa <= Dif_a;

    Reg2 <= mux2_out;
    Reg4 <= mux4_out;
    Reg_difb <= Dif_b;
  end if;
end process pipeline1;

correction: process (Reg_difa, Reg_difb)
begin -- process comp2
  if Reg_difa >= "0000000000000000" and Reg_difa <= "00000000000000100" then --
0=<Reg_difa<=0.5;
    corvalue_a <= "00101";          --0.625
  elsif Reg_difa > "00000000000000100" and Reg_difa <= "000000000000100000" then --
0.5<Reg_difa<=4.0;
    corvalue_a <= "00100";          -- 0.5
  elsif Reg_difa > "000000000000100000" and Reg_difa <= "000000000001000000" then --
4.0<Reg_difa<=8.0;
    corvalue_a <= "00011";          -- 0.375
  elsif Reg_difa > "000000000001000000" and Reg_difa <= "000000000001100000" then --
8.0<Reg_difa<=12;
    corvalue_a <= "00010";          -- 0.25
  elsif Reg_difa > "000000000001100000" and Reg_difa <= "000000000010000000" then --
12<Reg_difa<=16;
    corvalue_a <= "00001";          -- 0.125
  else
    -- 16 <=Reg_difa;
    corvalue_a <= "00000";          -- 0.0
  end if;

  if Reg_difb >= "0000000000000000" and Reg_difb <= "00000000000000100" then --
0=<Reg_difb<=0.5;
    corvalue_b <= "00101";          --0.625
  elsif Reg_difb > "00000000000000100" and Reg_difb <= "000000000000100000" then --
0.5<Reg_difb<=4.0;
    corvalue_b <= "00100";          -- 0.5
  elsif Reg_difb > "000000000000100000" and Reg_difb <= "000000000001000000" then --
4.0<Reg_difb<=8.0;
    corvalue_b <= "00011";          -- 0.375
  elsif Reg_difb > "000000000001000000" and Reg_difb <= "000000000001100000" then --
8.0<Reg_difb<=12;
    corvalue_b <= "00010";          -- 0.25

```

```

    elsif Reg_difb > "00000000001100000" and Reg_difb <= "00000000010000000" then --
12<Reg_difb<=16;
    corvalue_b <= "00001";          -- 0.125
    else                             -- 16 <=Reg_difb;
    corvalue_b <= "00000";          -- 0.0
    end if;
end process correction;

add5_6: process (Reg1, Reg2, corvalue_a, corvalue_b)
begin -- process add2
    adder5_out <= Reg1 + corvalue_a;
    adder6_out <= Reg2 + corvalue_b;
end process add5_6;

mux5_6: process (Reg3,Reg4,adder5_out,adder6_out,mux5sel,mux6sel)
begin -- process mux5
    if mux5sel = '1' then
        mux5_out <= adder5_out;
    else
        mux5_out <= Reg3;
    end if;

    if mux6sel = '1' then
        mux6_out <= adder6_out;
    else
        mux6_out <= Reg4;
    end if;
end process mux5_6;

logalpha_out <= mux5_out;
logbeta_out <= mux6_out;

end rtl;

```

File name: leunit.vhd

Purpose: extrinsic information calculation unit

library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_signed.all;

--No.0,2,4 corresponds to branch lable xk=-1; No.1,3,5 to xk=1;

entity leunit is

```

port (
    logalpha0 : in std_logic_vector(16 downto 0);
    logalpha1 : in std_logic_vector(16 downto 0);
    logbeta0  : in std_logic_vector(16 downto 0);
    logbeta1  : in std_logic_vector(16 downto 0);
    clk       : in std_logic;
    reset     : in std_logic;
    mux0_1sel : in std_logic;

```

```

mux4_5sel : in std_logic;
leunit_out : out std_logic_vector(16 downto 0));

end leunit;

architecture rtl of leunit is

    signal corvalue0, corvalue1 : std_logic_vector(4 downto 0);

    signal adder0_out, adder1_out, adder2_out, adder3_out : std_logic_vector(17 downto 0);
    signal mux0_out, mux1_out, mux2_out, mux3_out, mux4_out, mux5_out : std_logic_vector(17 downto
0);
    signal Reg0, Reg1, Dif0, Dif1 : std_logic_vector(17 downto 0);
    signal Reg_pipe0, Reg_pipe1, Reg_pipe2, Reg_pipe3 : std_logic_vector(17 downto 0);--pipeline register
group;

begin -- trl

    add0_1: process (logalpha0, logalpha1, logbeta0, logbeta1)
    begin -- process add

        adder0_out <= logalpha0(16)&logalpha0 + logbeta0;
        adder1_out <= logalpha1(16)&logalpha1 + logbeta1;

    end process add0_1;

    mux0_1: process (adder0_out, adder1_out, adder2_out, adder3_out, mux0_1sel)
    begin -- process mux1
        if mux0_1sel = '1' then
            mux0_out <= adder0_out;
            mux1_out <= adder1_out;
        else
            mux0_out <= adder2_out;
            mux1_out <= adder3_out;
        end if;
    end process mux0_1;

    Reg0_1: process (clk, reset)      -- actually it's pipeline 1:
    begin -- process Reg0_1
        if reset = '0' then          -- asynchronous reset (active low)
            Reg0 <= (others => '0');
            Reg1 <= (others => '0');
        elsif clk'event and clk = '1' then -- rising clock edge
            Reg0 <= mux0_out;
            Reg1 <= mux1_out;
        end if;
    end process Reg0_1;

    comp_max_sub: process (Reg0, Reg1, adder0_out, adder1_out)
    begin -- process comp_max
        if Reg0 >= adder0_out then
            mux2_out <= Reg0;
            Dif0 <= Reg0 - adder0_out;
        else

```

```

    mux2_out <= adder0_out;
    Dif0 <= adder0_out - Reg0;
end if;

if Reg1 >= adder1_out then
    mux3_out <= Reg1;
    Dif1 <= Reg1 - adder1_out;
else
    mux3_out <= adder1_out;
    Dif1 <= adder1_out - Reg1;
end if;
end process comp_max_sub;

comp2_3: process (Dif0, Dif1)    -- 18 bits width
begin -- process comp2_3
    if Dif0 >= "0000000000000000" and Dif0 <= "000000000000000100" then -- 0<=Dif0<=0.5;
        corvalue0 <= "00101";        --0.625
    elsif Dif0 > "000000000000000100" and Dif0 <= "0000000000000100000" then --0.5<Dif0<=4.0;
        corvalue0 <= "00100";        -- 0.5
    elsif Dif0 > "0000000000000100000" and Dif0 <= "0000000000001000000" then --4.0<Dif0<=8.0;
        corvalue0 <= "00011";        -- 0.375
    elsif Dif0 > "0000000000001000000" and Dif0 <= "0000000000001100000" then --8.0<Dif0<=12;
        corvalue0 <= "00010";        -- 0.25
    elsif Dif0 > "0000000000001100000" and Dif0 <= "0000000000010000000" then --12<Dif0<=16;
        corvalue0 <= "00001";        -- 0.125
    else
        -- 16 <=Dif0;
        corvalue0 <= "00000";        -- 0.0
    end if;

    if Dif1 >= "0000000000000000" and Dif1 <= "000000000000000100" then -- 0<=Dif1<=0.5;
        corvalue1 <= "00101";        --0.625
    elsif Dif1 > "000000000000000100" and Dif1 <= "0000000000000100000" then --0.5<Dif1<=4.0;
        corvalue1 <= "00100";        -- 0.5
    elsif Dif1 > "0000000000000100000" and Dif1 <= "0000000000001000000" then --4.0<Dif1<=8.0;
        corvalue1 <= "00011";        -- 0.375
    elsif Dif1 > "0000000000001000000" and Dif1 <= "0000000000001100000" then --8.0<Dif1<=12;
        corvalue1 <= "00010";        -- 0.25
    elsif Dif1 > "0000000000001100000" and Dif1 <= "0000000000010000000" then --12<Dif1<=16;
        corvalue1 <= "00001";        -- 0.125
    else
        -- 16 <=Dif1;
        corvalue1 <= "00000";        - 0.0
    end if;
end process comp2_3;

add2_3: process (mux2_out, mux3_out, corvalue0, corvalue1)
begin -- process add2_3
    adder2_out <= mux2_out + corvalue0;
    adder3_out <= mux3_out + corvalue1;
end process add2_3;

pipeline: process (clk, reset)
begin -- process pipeline
    if reset = '0' then                -- asynchronous reset (active low)

```

```

    Reg_pipe0 <= (others => '0');
    Reg_pipe1 <= (others => '0');
    Reg_pipe2 <= (others => '0');
    Reg_pipe3 <= (others => '0');
    elsif clk'event and clk = '1' then -- rising clock edge
        Reg_pipe0 <= Reg0;
        Reg_pipe1 <= Reg1;
        Reg_pipe2 <= adder2_out;
        Reg_pipe3 <= adder3_out;
    end if;
end process pipeline;

mux4_5: process (Reg_pipe0, Reg_pipe1, Reg_pipe2, Reg_pipe3, mux4_5sel)
begin -- process mux4_5
    if mux4_5sel = '1' then
        mux4_out <= Reg_pipe0;
        mux5_out <= Reg_pipe1;
    else
        mux4_out <= Reg_pipe2;
        mux5_out <= Reg_pipe3;
    end if;
end process mux4_5;

sub: process (mux4_out, mux5_out)
    variable diff : std_logic_vector(17 downto 0);
begin -- process sub

    diff := mux5_out - mux4_out;
    leunit_out <= diff(17)&diff(15 downto 0);

end process sub;

end rtl;

```

File name: LLR_unit.vhd

Purpose: log-likelihood soft-out calculation and hard decision unit

library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_signed.all;

--le_h is the sum of $L_c \cdot y_k + L_e$ for the horizontal decoding in the last iteration;

--le_v is the extrinsic value of vertical decoding in the last iteration;

entity LLR_unit is

port (

 le_h : in std_logic_vector(12 downto 0); -- from memlcycle_out

 le_v : in std_logic_vector(11 downto 0);

 infobit : out std_logic);

end LLR_unit;

architecture rtl of LLR_unit is

 signal sum : std_logic_vector(13 downto 0);

begin -- rtl

 sum <= le_h(12)&le_h + le_v;

 harddecision: process (sum)

 begin -- process harddecision

 if sum(13) = '0' then

 infobit <= '1';

 else

 infobit <= '0';

 end if;

 end process harddecision;

end rtl;

File name: FSM_buf.vhd

Purpose: finite state machine of turbo decoder, and address generator for intermediate memory

library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_unsigned.all;

entity FSM_buf is

 port (

 start : in std_logic;

 reset : in std_logic;--='0';

 clk0 : in std_logic; -- transmitted rate clock

 clk1 : in std_logic; -- decoding clock

 iter_number : in std_logic_vector(3 downto 0);--="0001"; -- iteration=5;

 SNR : in std_logic_vector(3 downto 0);--="0001"; -- presetting SNR value;

 deco_done : out std_logic; -- one block decoding finishing signal;

 buffer_sel : out std_logic; -- buffer1 and 2 selective signal

 mux3sel : out std_logic;

 mux4sel : out std_logic;

 mux5sel : out std_logic;

 mux6sel : out std_logic;

 mux0_1sel : out std_logic;

 mux4_5sel : out std_logic;

 we_memyk1 : out std_logic;

 we_memyk2 : out std_logic;

 we_memlcyk : out std_logic;

 web_memle : out std_logic;

 we_memlcykle : out std_logic;

 wea_memgamma : out std_logic;

 web_memalphabet: out std_logic;

 Le_in_sel : out std_logic; -- Le input value (Le value and Zero)selection signal

 addr_memyk1 : out std_logic_vector(9 downto 0); -- buffer1 addr.

 addr_memyk2 : out std_logic_vector(9 downto 0); -- buffer2 addr.

 addr_memlc : out std_logic_vector(3 downto 0);

```
addra_memle      : out std_logic_vector(9 downto 0); -- memle reading_ out addr.
addrb_memle      : out std_logic_vector(9 downto 0); -- memle writing_ in addr.
addr_memlcyk     : out std_logic_vector(9 downto 0);
addr_memlcykle   : out std_logic_vector(9 downto 0);
addra_memgamma   : out std_logic_vector(9 downto 0); -- memgamma reading out addr. A port for
alpha;
addrb_memgamma   : out std_logic_vector(9 downto 0); -- memgamma reading out addr. B port for
beta ;
addrb_memalphabet : out std_logic_vector(9 downto 0); -- memalpha and membeta write back addr.
addra_memalpha0  : out std_logic_vector(9 downto 0); -- memalpha0 read out addr.
addra_memalpha1  : out std_logic_vector(9 downto 0); -- memalpha1 read out addr.
addra_membeta0   : out std_logic_vector(9 downto 0); -- membeta0 read out addr.
addra_membeta1   : out std_logic_vector(9 downto 0)); -- membeta0 read out addr.
```

end FSM_buf;

architecture rtl of FSM_buf is

-- interleave accessing address table of memle and memlcyk in Vertical decoding in Gamma state:
component addrmemletop -- addr. table for memle and memlcyk access in vertical decoding:

```
port (
  addr : in  std_logic_vector(9 downto 0);
  clk  : in  std_logic;
  dout : out std_logic_vector(9 downto 0));
```

end component;

-- Following 4 address memories for memalpha~membeta access in ALPHABETA state-----

component addrmemalpha0top -- addr. table for accessing memalpha0 in alphabet state:

```
port (
  addr : in  std_logic_vector(9 downto 0);
  clk  : in  std_logic;
  dout : out std_logic_vector(9 downto 0));
```

end component;

component addrmemalpha1top -- addr. table for accessing memalpha1 in alphabet state:

```
port (
  addr : in  std_logic_vector(9 downto 0);
  clk  : in  std_logic;
  dout : out std_logic_vector(9 downto 0));
```

end component;

component addrmembeta0top -- addr. table for accessing membeta0 in alphabet state:

```
port (
  addr : in  std_logic_vector(9 downto 0);
  clk  : in  std_logic;
  dout : out std_logic_vector(9 downto 0));
```

end component;

component addrmembeta1top -- addr. table for accessing membeta1 in alphabet state:

```
port (
  addr : in  std_logic_vector(9 downto 0);
  clk  : in  std_logic;
  dout : out std_logic_vector(9 downto 0));
```

end component;

-----following 3 memories for memalpha~membeta access in Le-unit state-----

```
component addralpha01top      --addr. table for accessing memalpha01 in leunit state;
port (
  addr: IN  std_logic_VECTOR(9 downto 0);
  clk : IN  std_logic;
  dout: OUT std_logic_VECTOR(9 downto 0));
end component;
```

```
component addrbeta0top        --addr. table for accessing membeta0 in leunit state;
port (
  addr: IN  std_logic_VECTOR(9 downto 0);
  clk : IN  std_logic;
  dout: OUT std_logic_VECTOR(9 downto 0));
end component;
```

```
component addrbeta1top        --addr. table for accessing membeta1 in leunit state;
port (
  addr: IN  std_logic_VECTOR(9 downto 0);
  clk : IN  std_logic;
  dout: OUT std_logic_VECTOR(9 downto 0));
end component;
```

```
-----
component multiplier1top
port (
  a : in  std_logic_vector(4 downto 0);
  o : out std_logic_vector(9 downto 0));
end component;
```

```
component multiplier2top
port (
  a : in  std_logic_vector(4 downto 0);
  o : out std_logic_vector(7 downto 0));
end component;
```

```
type statetype is (idle, gamma, alphabeta, leunit, LLR); --define state type and signal;
signal state : statetype;
```

```
signal iter_count : std_logic_vector(3 downto 0);
signal we,we0,we1,we_memlcycle0,we_memlcycle1 : std_logic;
signal we2,we2_1,we1_pipe, we2_pipe,we_memle_leunit,we_memle_LLRL : std_logic;
signal upcount0, upcount1, upcount2, upcount3, upcount4 : std_logic;
signal count0,count1, count2, count3, count4 : std_logic_vector(9 downto 0);
signal addrmem, addr,addr0,addr2, addr2_pipe, addr2_pipe1 : std_logic_vector(9 downto 0);
signal addr1, addr1_pipe,addr1_pipe0,addr1_pipe1 : std_logic_vector(9 downto 0);
signal addr0_delay, addr1_delay : std_logic_vector(9 downto 0);
signal X, addrmemle_V : std_logic_vector(9 downto 0);
signal addr_gamma_in, addra_gamma_out, addrb_gamma_out : std_logic_vector(9 downto 0);
signal premux3sel,premux4sel,premux5sel,premux6sel : std_logic;
signal premux0_1sel,premux4_5sel, pipe0,pipe1, pipe2 : std_logic;
signal code26 : std_logic_vector(4 downto 0);
signal muxpipe01 : std_logic;
signal we3, we3_pipe0,we3_pipe1, we3_pipe2 : std_logic;
```

```

signal preaddr_memle_leunit,addr3, addr3_pipe0,addr3_pipe1,addr3_pipe2,addr3_pipe3 :
std_logic_vector(9 downto 0);
signal addr_memle_g, addr_memle_leunit,addr_memle_clear,addr4 : std_logic_vector(9 downto 0);
signal alpha0addr, alpha1addr, beta0addr,beta1addr : std_logic_vector(9 downto 0);
signal mux5sel_pipe, mux6sel_pipe : std_logic;
signal addressmem : std_logic_vector(9 downto 0);
signal alpha0laddr_leunit,beta0addr_leunit,beta1addr_leunit : std_logic_vector(9 downto 0);
signal p1 : std_logic_vector(9 downto 0);
signal p2 : std_logic_vector(7 downto 0);
signal buffer1_on : std_logic;
signal shift, buf_full : std_logic;

```

```

begin -- rtl

```

```

    clock <= start and clk1;          -- avoiding initial 'X' value of memory;

```

```

--process of buffer1 and buffer2 switch control-----

```

```

process (clk0, reset)
begin -- process
    if reset = '0' then                -- asynchronous reset (active low)
        shift<='0';
    elsif clk0'event and clk0 = '1' then -- rising clock edge
        if start='1' then
            shift<='1';
        else
            shift<='0';
        end if;
    end if;
end process;

```

```

process (clk0, reset)                -- receive channel signal 988bit(one package)
begin -- process
    if reset = '0' then                -- asynchronous reset (active low)
        count0 <= (others=>'0');
        buf_full <= '0';
    elsif clk0'event and clk0 = '1' then -- rising clock edge
        if shift='1' then
            if count0<"1111011011" then -- count0=987
                count0<=count0+1;
                buf_full <= '0';
            else
                count0<=(others=>'0');
                buf_full <= '1';        -- one buffer fill full;
            end if;
        end if;
    end if;
end process;

```

```

process (clk0, reset)                -- buffer1 and buffer2 switch over control;
begin -- process
    if reset = '0' then                -- asynchronous reset (active low)
        buffer1_on<='1';
    elsif clk0'event and clk0 = '1' then -- rising clock edge
        if count0="1111011011" then -- count0=987

```

```

    buffer1_on<=not buffer1_on; -- buffer1_on='1'=>active.buffer2 unactive;
  end if;
end if;
end process;

buffer_sel <= buffer1_on;

process(buffer1_on,count0,addr1,start)
begin -- process
  if buffer1_on='1' then      -- buffer1 work;
    addr_memyk1 <= count0;
    we_memyk1 <= start;
  else
    addr_memyk1 <= addr1;
    we_memyk1 <= '0';
  end if;
end process;

process(buffer1_on,count0,addr1,start)
begin
  if buffer1_on='1' then      -- buffer2 work;
    addr_memyk2 <= addr1;
    we_memyk2 <= '0';
  else
    addr_memyk2 <= count0;
    we_memyk2 <= start;
  end if;
end process;
--buffer switch control process finish-----

--finite state machine process-----
--considering about the influence of pipeline, count extra cycles (lency) for state change:
FSM: process (clk1, reset, start)
begin -- process FSM
  if reset = '0' or start = '0' then      -- asynchronous reset (active low)
    state <= idle;
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    case state is

      when idle => if buf_full='1' then -- buffer1_on='1' then
        state <= gamma;
      else
        state <= idle;
      end if;

      when gamma => if iter_count = "0000" then -- from Dec#2 on(iter_count=
        -- 1),alphabet unit just
        -- recalculate info. bit
        -- part(lc*yk + Le)
        if count1 < "1111011101" then -- addr =989="1111011101";
          state <= gamma;
        else
          state <= alphabeta;
        end if;
    end case;
  end if;
end process;

```

```

else      -- iter_count>=1 and up on;
  if count1 < "1010100101" then -- addr=677="1010100101"
    state <= gamma;
  else
    state <= alphabeta;
  end if;
end if;

when alphabeta => if count2 < "1001111110" then -- 638="1001111110"( 637 to 638 due
-- to memory access response delay;
  state <= alphabeta;
else
  state <= leunit;
end if;

when leunit => if count3 < "1000100010" and code26 < "11001" then -- 546="1000100010";
  state <= leunit;
elsif count3 = "1000100010" and code26 < "11001" then --0~25bit
  state <= alphabeta;
elsif count3 = "1000100010" and code26 = "11001" and iter_count < iter_number then --
0~25code
  state <= gamma;
elsif count3 = "1000100010" and code26 = "11001" and iter_count = iter_number then
  state <= LLR;
end if;

when LLR => if count4 < "1010100100" then -- 676 info bits="1010100100"
  state <= LLR;
else
  state <= idle; -- receive new info block;
end if;

when others => null;
end case;
end if;
end process FSM;

deco_done <= '1' when state = idle else '0'; -- one block decoding finish;
upcount1 <= '1' when state = gamma else '0'; -- state counter starting signal;
upcount2 <= '1' when state = alphabeta else '0';
upcount3 <= '1' when state = leunit else '0';
upcount4 <= '1' when state = LLR else '0';
Le_in_sel <= '1' when state = LLR else '0';

-- purpose: choose different Lc value corresponding to different SNR value;
-- type : combinational
-- inputs : SNR
-- outputs: addr_memlc
SNR_ratio: process (SNR)
begin -- process SNR_ratio
  case SNR is
    -- SNR format "XXX.X" dB
    when "1010" => addr_memlc <= "0000"; -- 5dB
    when "1001" => addr_memlc <= "0001"; -- 4.5dB
    when "1000" => addr_memlc <= "0010"; -- 4dB
    when "0111" => addr_memlc <= "0011";
    when "0110" => addr_memlc <= "0100";
    when "0101" => addr_memlc <= "0101";
  end case;
end process;

```

```

when "0100" => addr_memlc <= "0110";
when "0011" => addr_memlc <= "0111";
when "0010" => addr_memlc <= "1000";
when "0001" => addr_memlc <= "1001"; -- 0.5dB
when others => null;
end case;
end process SNR_ratio;

-- iter_count counter-----
-- purpose: count iter_count number for decoding(2 iter_count = one horizontal and vertical decoding)
-- type : sequential
-- inputs : clk1, reset,
-- outputs: iter_count
iter: process (clk1, reset)
begin -- process iter
    if reset = '0' then          -- asynchronous reset (active low)
        iter_count <= (others => '0');
    elsif clk1'event and clk1 = '1' then -- rising clock edge
        if count3 = "1000100010" and code26 = "11001" then -- one time decoding finished, change to next
            interleave
                -- decoding phase, 546="1000100010";
                if iter_count < iter_number then
                    iter_count <= iter_count + 1;
                else
                    iter_count <= (others => '0');
                end if;
            end if;
        end if;
    end process iter;

--counter for state "gamma"-----
c1: process (clk1, reset)
begin -- process c1
    if reset = '0' then          -- asynchronous reset (active low)
        count1 <= (others => '0');
    elsif clk1'event and clk1 = '1' then -- rising clock edge
        if iter_count = "0000" then
            if upcount1 = '1' then --increase address of mem_yk for state "gamma";
                if count1 < "1111011101" then --used uplimit address is 987; 989="1111011101"
                    count1 <= count1 + 1;
                else
                    count1 <= (others => '0'); --"0000000000";
                end if;
            end if;
        else
            if upcount1 = '1' then -- 2nd iter_count and up on
                if count1 < "1010100101" then -- 677="1010100101";
                    count1 <= count1 + 1;
                else
                    count1 <= (others => '0'); --"0000000000";
                end if;
            end if;
        end if;
    end process c1;
end if;

```

end process c1;

--address generator and write control for gamma part-----

addr0 <= count1 when count1 < "1010100100" else (others => '0'); -- 676="1010100100";

addr1 <= count1 when count1 < "1111011100" else (others => '0'); -- 988="1111011100";

-- 676="1010100100";-- 988="1111011100";

we0 <= '1' when count1 < "1010100100" and upcount1 = '1' and upcount2 = '0' and upcount3 = '0' else '0';

we1 <= '1' when count1 < "1111011100" and upcount1 = '1' and upcount2 = '0' and upcount3 = '0' else '0';

process (clk1, reset) -- addr0 and addr1 delay one cycle;

begin -- process

if reset = '0' then -- asynchronous reset (active low)

addr0_delay <= (others => '0');

addr1_delay <= (others => '0');

elsif clk1'event and clk1 = '1' then -- rising clock edge

addr0_delay <= addr0;

addr1_delay <= addr1;

end if;

end process;

addrmemle_lcyk : addrmemletop port map (

addr => addr0,

clk => clock,

dout => addrmemle_V); -- memle, memlcyk addr. in Vertical INTERLEAVE decoding;

process (iter_count, addr0_delay, addr1_delay, addrmemle_V)

begin -- process

if iter_count = "0000" then

addr_memle_g <= addr0_delay;

addr_memlcyk <= addr1_delay; --

elsif iter_count="0010" or iter_count="0100" or iter_count="0110" or iter_count="1000" or
iter_count="1010" or iter_count="1100" or iter_count="1110" then

addr_memle_g <= addr0_delay;

addr_memlcyk <= addr0_delay; -- from iteration=2 on, just calculate

-- 676 info bits's gamma value;

else

addr_memle_g <= addrmemle_V;

addr_memlcyk <= addrmemle_V;

end if;

end process;

process (iter_count, we0, we1)

begin -- process

if iter_count = "0000" then

we <= we1;

else

we <= we0;

end if;

end process;

delay1: process (clk1, reset) --1st stage of pipeline

```

begin -- process delay1
  if reset = '0' then          -- asynchronous reset (active low)
    addr1_pipe <= (others => '0'); -- addr1(0 to 987)

    we_memlcyk <= '0';
    we1_pipe <= '0';
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    if iter_count = "0000" then
      addr1_pipe <= addr1;      -- transfer to next stage of pipeline;

      we_memlcyk <= we1;        -- just read memlcyk for lc*yk value
                                -- after 1st half iter_count;
    else
      addr1_pipe <= addr0;

      we_memlcyk <= '0';
    end if;

    we1_pipe <= we;            -- transfer to next stage of pipeline;

  end if;
end process delay1;

delay2: process (clk1, reset)    -- 2nd stage of pipeline
begin -- process delay2
  if reset = '0' then          -- asynchronous reset (active low)
    addr_gamma_in <= (others => '0');
    wea_memgamma <= '0';
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    addr_gamma_in <= addr1_pipe;    -- mem-loggamma write back address;
    wea_memgamma <= we1_pipe;
  end if;
end process delay2;

--for mem_lcykle write in at gamma state and read out at LLR state-----
process (clk1, reset)
begin -- process
  if reset = '0' then          -- asynchronous reset (active low)
    we_memlcycle0 <= '0';
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    we_memlcycle0 <= we0;
  end if;
end process;

process (clk1, reset)
begin -- process
  if reset = '0' then          -- asynchronous reset (active low)
    addr1_pipe1 <= (others => '0');
    we_memlcycle1 <= '0';
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    addr1_pipe1 <= addrmemle_V;
    we_memlcycle1 <= we_memlcycle0;
  end if;
end process;

```

```

process (state,addr1_pipe1,addr4,iter_number,iter_count)
begin -- process
  if state = gamma then
    if iter_count = iter_number then
      addr_memlcycle <= addr1_pipe1; -- mem-lcycle write in address(GAMMA state);
    else
      addr_memlcycle <= (others => '0');
    end if;
  elsif state = LLR then
    addr_memlcycle <= addr4; -- mem-lcycle read out address(LLR state);
  else
    addr_memlcycle <= (others => '0');
  end if;
end process;

```

```

process (iter_count, we_memlcycle1,iter_number)
begin -- process
  if iter_count = iter_number then --iter_count = "0000" then --
    we_memlcycle <= we_memlcycle1;
  else
    we_memlcycle <= '0';
  end if;
end process;

```

-----alphabet unit-----

```

c2: process (clk1, reset)
begin -- process count2
  if reset = '0' then -- asynchronous reset (active low)
    count2 <= (others => '0');
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    if upcount2 = '1' then
      if count2 < "100111110" then -- 638="100111110";
        count2 <= count2 + 1;
      else
        count2 <= (others => '0');
      end if;
    end if;
  end if;
end process c2;

```

```

we2 <= '1' when count2 < "1001111100" and upcount1 = '0' and upcount2 = '1' and upcount3 = '0' else '0';
addr2 <= count2 when count2 < "1001111101" else (others => '0'); -- 636="1001111100";
addrmem <= count2 when count2 < "1001111100" else (others => '0'); --addrmem(0 to 635) means 636
nodes;

```

```

addralpha0 : addrmemalpha0top port map (
  addr => addrmem,
  clk => clock,--clk1,
  dout => alpha0addr); -- memalpha0 read-out address;

```

```

addralpha1 : addrmemalpha1top port map (

```

```

addr => addrmem,
clk => clock,--clk1,
dout => alpha1addr);      -- memalpha1 read-out address;

addrbeta0 : addrmembeta0top port map (
  addr => addrmem,
  clk => clock,--clk1,
  dout => beta0addr);      -- membeta0 read-out address;

addrbeta1 : addrmembeta1top port map (
  addr => addrmem,
  clk => clock,--clk1,
  dout => beta1addr);      -- membeta1 read-out address;

multiplier1 : multiplier1top port map (
  a => code26,
  o => p1);                -- multiplier1: 26*code26

multiplier2 : multiplier2top port map (
  a => code26,
  o => p2);                -- multiplier2: 6*code26

addrgamma: process (addr2, p1, p2, iter_count)
  variable B : std_logic_vector(9 downto 0);
begin -- process addrgamma

  if iter_count="0000" or iter_count="0010" or iter_count="0100" or iter_count="0110" or
  iter_count="1000" or iter_count="1010" or iter_count="1100" or iter_count="1110" then -- horizontal
  decoding;
    B:="1010100100" + p2;    -- "1010100100"=676;
  else
    B:="1101000000" + p2;    -- "1101000000"=832;
  end if;

  -- for each time k, only one gamma value for all node accordingly;
  -- addr2 means counting clock cycle number;
  if addr2 <= "0000000010" then -- k=0
    addra_gamma_out <= p1;
    addrb_gamma_out <= B + "101";
  elsif "0000000010" < addr2 and addr2 <= "0000000110" then -- k=1
    addra_gamma_out <= p1 + "00001";
    addrb_gamma_out <= B + "100";
  elsif "0000000110" < addr2 and addr2 <= "0000001110" then -- k=2
    addra_gamma_out <= p1 + "00010";
    addrb_gamma_out <= B + "011";
  elsif "0000001110" < addr2 and addr2 <= "0000010110" then -- k=3
    addra_gamma_out <= p1 + "00011";
    addrb_gamma_out <= p1 + "11001";
  elsif "0000010110" < addr2 and addr2 <= "0000100110" then -- k=4
    addra_gamma_out <= p1 + "00100";
    addrb_gamma_out <= B + "010";
  elsif "0000100110" < addr2 and addr2 <= "0000110110" then -- k=5
    addra_gamma_out <= p1 + "00101";
    addrb_gamma_out <= p1 + "11000";
  elsif "0000110110" < addr2 and addr2 <= "0001000110" then -- k=6

```

```

    addra_gamma_out <= p1 + "00110";
    addrb_gamma_out <= p1 + "10111";
    elsif "0001000110" < addr2 and addr2 <= "0001010110" then -- k=7
        addra_gamma_out <= p1 + "00111";
        addrb_gamma_out <= p1 + "10110";
    elsif "0001010110" < addr2 and addr2 <= "0001110110" then -- k=8
        addra_gamma_out <= p1 + "01000";
        addrb_gamma_out <= B + "001";
    elsif "0001110110" < addr2 and addr2 <= "0010010110" then -- k=9
        addra_gamma_out <= p1 + "01001";
        addrb_gamma_out <= p1 + "10101";
    elsif "0010010110" < addr2 and addr2 <= "0010110110" then -- k=10
        addra_gamma_out <= p1 + "01010";
        addrb_gamma_out <= p1 + "10100";
    elsif "0010110110" < addr2 and addr2 <= "0011010110" then -- k=11
        addra_gamma_out <= p1 + "01011";
        addrb_gamma_out <= p1 + "10011";
    elsif "0011010110" < addr2 and addr2 <= "0011110110" then -- k=12
        addra_gamma_out <= p1 + "01100";
        addrb_gamma_out <= p1 + "10010";
    elsif "0011110110" < addr2 and addr2 <= "0100010110" then -- k=13
        addra_gamma_out <= p1 + "01101";
        addrb_gamma_out <= p1 + "10001";
    elsif "0100010110" < addr2 and addr2 <= "0100110110" then -- k=14
        addra_gamma_out <= p1 + "01110";
        addrb_gamma_out <= p1 + "10000";
    elsif "0100110110" < addr2 and addr2 <= "0101000110" then -- k=15
        addra_gamma_out <= B;
        addrb_gamma_out <= p1 + "01111";
    elsif "0101000110" < addr2 and addr2 <= "0101100110" then -- k=16
        addra_gamma_out <= p1 + "01111";
        addrb_gamma_out <= B;
    elsif "0101100110" < addr2 and addr2 <= "0110000110" then -- k=17
        addra_gamma_out <= p1 + "10000";
        addrb_gamma_out <= p1 + "01110";
    elsif "0110000110" < addr2 and addr2 <= "0110100110" then -- k=18
        addra_gamma_out <= p1 + "10001";
        addrb_gamma_out <= p1 + "01101";
    elsif "0110100110" < addr2 and addr2 <= "0111000110" then -- k=19
        addra_gamma_out <= p1 + "10010";
        addrb_gamma_out <= p1 + "01100";
    elsif "0111000110" < addr2 and addr2 <= "0111100110" then -- k=20
        addra_gamma_out <= p1 + "10011";
        addrb_gamma_out <= p1 + "01011";
    elsif "0111100110" < addr2 and addr2 <= "1000000110" then -- k=21
        addra_gamma_out <= p1 + "10100";
        addrb_gamma_out <= p1 + "01010";
    elsif "1000000110" < addr2 and addr2 <= "1000100110" then -- k=22
        addra_gamma_out <= p1 + "10101";
        addrb_gamma_out <= p1 + "01001";
    elsif "1000100110" < addr2 and addr2 <= "1000110110" then -- k=23
        addra_gamma_out <= B + "001";
        addrb_gamma_out <= p1 + "01000";
    elsif "1000110110" < addr2 and addr2 <= "1001000110" then -- k=24

```

```

    addra_gamma_out <= p1 + "10110";
    addrb_gamma_out <= p1 + "00111";
    elsif "1001000110" < addr2 and addr2 <= "1001010110" then -- k=25
        addra_gamma_out <= p1 + "10111";
        addrb_gamma_out <= p1 + "00110";
    elsif "1001010110" < addr2 and addr2 <= "1001100110" then -- k=26
        addra_gamma_out <= p1 + "11000";
        addrb_gamma_out <= p1 + "00101";
    elsif "1001100110" < addr2 and addr2 <= "1001101110" then -- k=27
        addra_gamma_out <= B + "010";
        addrb_gamma_out <= p1 + "00100";
    elsif "1001101110" < addr2 and addr2 <= "1001110110" then -- k=28
        addra_gamma_out <= p1 + "11001";
        addrb_gamma_out <= p1 + "00011";
    elsif "1001110110" < addr2 and addr2 <= "1001111010" then -- k=29
        addra_gamma_out <= B + "011";
        addrb_gamma_out <= p1 + "00010";
    elsif "1001111010" < addr2 and addr2 <= "1001111100" then -- k=30
        addra_gamma_out <= B + "100";
        addrb_gamma_out <= p1 + "00001";
    else
        addra_gamma_out <= (others => '0');--p1;
        addrb_gamma_out <= (others => '0');--p1;
    end if;
end process addrgamma;

-- purpose: determine branch number and label according to address value:
process (alpha0addr,alpha1addr,beta0addr,beta1addr)
begin -- process
    if alpha0addr /= "1001111101" then -- /=637="1001111101";means one branch:
        premux3sel <= '0';
    else
        premux3sel <= '1';
    end if;
    if alpha0addr /= "1001111101" and alpha1addr /= "1001111101" then
        premux5sel <= '1';
    else
        premux5sel <= '0';
    end if;

    if beta0addr /= "1001111101" then
        premux4sel <= '0';
    else
        premux4sel <= '1';
    end if;
    if beta0addr /= "1001111101" and beta1addr /= "1001111101" then
        premux6sel <= '1';
    else
        premux6sel <= '0';
    end if;
end process;

process (clk1, reset)
    -- one more delay due to slow memory
    -- access response;

```

```
begin -- process
  if reset = '0' then          -- asynchronous reset (active low)
    we2_1 <= '0';

    mux3sel <= '0';
    mux4sel <= '0';

    mux5sel_pipe <= '0';
    mux6sel_pipe <= '0';
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    we2_1 <= we2;

    mux3sel <= premux3sel;
    mux4sel <= premux4sel;

    mux5sel_pipe <= premux5sel;
    mux6sel_pipe <= premux6sel;
  end if;
end process;

delay3: process (clk1, reset)
begin -- process delay3
  if reset = '0' then          -- asynchronous reset (active low)
    we2_pipe <= '0';
    mux5sel <= '0';
    mux6sel <= '0';
    addr2_pipe <= (others => '0');
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    we2_pipe <= we2_1;
    mux5sel <= mux5sel_pipe;
    mux6sel <= mux6sel_pipe;
    addr2_pipe <= addr2;
  end if;
end process delay3;

delay4: process (clk1, reset)
begin -- process delay4
  if reset = '0' then          -- asynchronous reset (active low)
    web_memalphabeta <= '0';
    addrb_memalphabeta <= (others => '0');
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    web_memalphabeta <= we2_pipe;
    addrb_memalphabeta <= addr2_pipe; -- for test of address;
  end if;
end process delay4;

process (state, addr_gamma_in, addra_gamma_out, addrb_gamma_out)
begin
  -- memgamma addr. selection
  if state=gamma then
    addra_memgamma <= addr_gamma_in;
  elsif state=alphabeta then
    addra_memgamma <= addra_gamma_out;
    addrb_memgamma <= addrb_gamma_out;
  else
```

```
    addra_memgamma <= (others=>'0');
    addrb_memgamma <= (others=>'0');
end if;
end process;
```

-----leunit unit-----

```
c3: process (clk1, reset)
begin -- process count
    if reset = '0' then          -- asynchronous reset (active low)
        count3 <= (others => '0');
    elsif clk1'event and clk1 = '1' then -- rising clock edge
        if upcount3 = '1' and code26 < "11010" then
            if count3 < "1000100010" then -- count3<546=(total branch 543 + 3 letency);
                count3 <= count3 + 1;      -- 546="1000100010";
            else
                count3 <= (others => '0');
            end if;
        end if;
    end if;
end process c3;
```

```
codenum: process (clk1, reset) -- codeword counter
begin -- process count
    if reset = '0' then          -- asynchronous reset (active low)
        code26 <= (others => '0');
    elsif clk1'event and clk1 = '1' then -- rising clock edge
        if count3 = "1000100010" then
            if code26 < "11001" then -- codenumber<26; sign+value
                code26 <= code26 + 1;
            else
                code26 <= "00000";
            end if;
        end if;
    end if;
end process codenum;
```

```
we3 <= '1' when count3 < "1000100000" and upcount3 = '1' else '0';
addressmem <= count3 when count3 < "1000011111" else (others => '0'); --543="1000011111";
```

```
alpha01 : addralpha01top port map (
    addr => addressmem,
    clk => clock,--clk1,
    dout => alpha01addr_leunit); -- memalpha0 and memalpha1 read-out address;
```

```
leunitbeta0 : addrbeta0top port map (
    addr => addressmem,
    clk => clock,--clk1,
    dout => beta0addr_leunit); -- membeta0 read-out address;
```

```
leunitbeta1 : addrbeta1top port map (
    addr => addressmem,
    clk => clock,--clk1,
    dout => beta1addr_leunit); -- membeta1 read-out address;
-- above 3 memalpha and membeta storing address table
```

```

process (count3,p1,code26,iter_count)      -- generate mem_le write back address addr3
begin -- process

    if iter_count="0000" or iter_count="0010" or iter_count="0100" or iter_count="0110" or
    iter_count="1000" or iter_count="1010" or iter_count="1100" or iter_count="1110" then -- horizontal
    decoding

        if count3 = "0000000000" then -- =0
            addr3 <= p1;
        elsif count3 > "0000000000" and count3 <= "0000000010" then -- <=2;
            addr3 <= p1+ "00001";
        elsif count3 > "0000000010" and count3 <= "0000000110" then -- <=6;
            addr3 <= p1+ "00010";
        elsif count3 > "0000000110" and count3 <= "0000001110" then -- <=14;
            addr3 <= p1+ "00011";
        elsif count3 > "0000001110" and count3 <= "0000010110" then -- <=22;
            addr3 <= p1+ "00100";
        elsif count3 > "0000010110" and count3 <= "0000100110" then -- <=38;
            addr3 <= p1+ "00101";
        elsif count3 > "0000100110" and count3 <= "0000110110" then -- <=54;
            addr3 <= p1+ "00110";
        elsif count3 > "0000110110" and count3 <= "0001000110" then -- <=70;
            addr3 <= p1+ "00111";
        elsif count3 > "0001000110" and count3 <= "0001010110" then -- <=86;
            addr3 <= p1+ "01000";
        elsif count3 > "0001010110" and count3 <= "0001110110" then -- <=118;
            addr3 <= p1+ "01001";
        elsif count3 > "0001110110" and count3 <= "0010010110" then -- <=150;
            addr3 <= p1+ "01010";
        elsif count3 > "0010010110" and count3 <= "0010110110" then -- <=182;
            addr3 <= p1+ "01011";
        elsif count3 > "0010110110" and count3 <= "0011010110" then -- <=214;
            addr3 <= p1+ "01100";
        elsif count3 > "0011010110" and count3 <= "0011110110" then -- <=246;
            addr3 <= p1+ "01101";
        elsif count3 > "0011110110" and count3 <= "0100010110" then -- <=278;
            addr3 <= p1+ "01110";
        elsif count3 > "0100010110" and count3 <= "0100100110" then -- <=294;
            addr3 <= p1+ "01111";
        elsif count3 > "0100100110" and count3 <= "0101000110" then -- <=326;
            addr3 <= p1+ "10000";
        elsif count3 > "0101000110" and count3 <= "0101100110" then -- <=358;
            addr3 <= p1+ "10001";
        elsif count3 > "0101100110" and count3 <= "0110000110" then -- <=390;
            addr3 <= p1+ "10010";
        elsif count3 > "0110000110" and count3 <= "0110100110" then -- <=422;
            addr3 <= p1+ "10011";
        elsif count3 > "0110100110" and count3 <= "0111000110" then -- <=454;
            addr3 <= p1+ "10100";
        elsif count3 > "0111000110" and count3 <= "0111100110" then -- <=486;
            addr3 <= p1+ "10101";
        elsif count3 > "0111100110" and count3 <= "0111110110" then -- <=502;
            addr3 <= p1 + "10110";
    end if;
end process;

```

```

elsif count3 > "0111110110" and count3 <= "1000000110" then -- <=518;
  addr3 <= p1+ "10111";
elsif count3 > "1000000110" and count3 <= "1000010110" then -- <=534;
  addr3 <= p1 + "11000";
else
  -- >534;
  addr3 <= p1+ "11001";
end if;
else
  if count3 = "0000000000" then -- =0
    addr3 <= "00000"&code26;
  elsif count3 > "0000000000" and count3 <= "0000000010" then -- <=2;
    addr3 <= "0000011010" + code26;
  elsif count3 > "0000000010" and count3 <= "0000000110" then -- <=6;
    addr3 <= "0000110100" + code26; --"11010" * "00010" + code26;
  elsif count3 > "0000000110" and count3 <= "0000001110" then -- <=14;
    addr3 <= "0001001110" + code26; --"11010" * "00011" + code26;
  elsif count3 > "0000001110" and count3 <= "0000010110" then -- <=22;
    addr3 <= "0001101000" + code26; --"11010" * "00100" + code26;
  elsif count3 > "0000010110" and count3 <= "0000100110" then -- <=38;
    addr3 <= "0010000010" + code26; --"11010" * "00101" + code26;
  elsif count3 > "0000100110" and count3 <= "0000110110" then -- <=54;
    addr3 <= "0010011100" + code26; --"11010" * "00110" + code26;
  elsif count3 > "0000110110" and count3 <= "0001000110" then -- <=70;
    addr3 <= "0010110110" + code26; --"11010" * "00111" + code26;
  elsif count3 > "0001000110" and count3 <= "0001010110" then -- <=86;
    addr3 <= "0011010000" + code26; --"11010" * "01000" + code26;
  elsif count3 > "0001010110" and count3 <= "0001110110" then -- <=118;
    addr3 <= "0011101010" + code26; --"11010" * "01001" + code26;
  elsif count3 > "0001110110" and count3 <= "0010010110" then -- <=150;
    addr3 <= "0100000100" + code26; --"11010" * "01010" + code26;
  elsif count3 > "0010010110" and count3 <= "0010110110" then -- <=182;
    addr3 <= "0100011110" + code26; --"11010" * "01011" + code26;
  elsif count3 > "0010110110" and count3 <= "0011010110" then -- <=214;
    addr3 <= "0100111000" + code26; --"11010" * "01100" + code26;
  elsif count3 > "0011010110" and count3 <= "0011110110" then -- <=246;
    addr3 <= "0101010010" + code26; --"11010" * "01101" + code26;
  elsif count3 > "0011110110" and count3 <= "0100010110" then -- <=278;
    addr3 <= "0101101100" + code26; --"11010" * "01110" + code26;
  elsif count3 > "0100010110" and count3 <= "0100100110" then -- <=294;
    addr3 <= "0110000110" + code26; --"11010" * "01111" + code26;
  elsif count3 > "0100100110" and count3 <= "0101000110" then -- <=326;
    addr3 <= "0110100000" + code26; --"11010" * "10000" + code26;
  elsif count3 > "0101000110" and count3 <= "0101100110" then -- <=358;
    addr3 <= "0110111010" + code26; --"11010" * "10001" + code26;
  elsif count3 > "0101100110" and count3 <= "0110000110" then -- <=390;
    addr3 <= "0111010100" + code26; --"11010" * "10010" + code26;
  elsif count3 > "0110000110" and count3 <= "0110100110" then -- <=422;
    addr3 <= "0111110110" + code26; --"11010" * "10011" + code26;
  elsif count3 > "0110100110" and count3 <= "0111000110" then -- <=454;
    addr3 <= "1000001000" + code26; --"11010" * "10100" + code26;
  elsif count3 > "0111000110" and count3 <= "0111100110" then -- <=486;
    addr3 <= "1000100010" + code26; --"11010" * "10101" + code26;
  elsif count3 > "0111100110" and count3 <= "0111110110" then -- <=502;
    addr3 <= "1000111100" + code26; --"11010" * "10110" + code26;

```

```

elsif count3 > "0111110110" and count3 <= "1000000110" then -- <=518;
    addr3 <= "1001010110" + code26; --"11010" * "10111" + code26;
elsif count3 > "1000000110" and count3 <= "1000010110" then -- <=534;
    addr3 <= "1001110000" + code26; --"11010" * "11000" + code26;
else
    -- >534;
    addr3 <= "1010001010" + code26; --"11010" * "11001" + code26;
end if;
end if;
end process;

process (count3)
begin -- process
    if count3 /= "0000000000" then -- 2 and over 2 branches;
        premux4_5sel <= '0';
    else
        premux4_5sel <= '1'; -- only one branch;
    end if;

    if (count3="0000000000" or count3="0000000001" or count3="0000000011" or count3="0000000111")
    or (count3="0000001111" or count3="0000010111" or count3="0000100111" or
count3="0000110111")
    or (count3="0001000111" or count3="0001010111" or count3="0001110111" or
count3="0010010111")
    or (count3="0010110111" or count3="0011010111" or count3="0011110111" or
count3="0100010111")
    or (count3="0100100111" or count3="0101000111" or count3="0101100111" or count3="0110000111"
or count3="0110100111")
    or (count3="0111000111" or count3="0111100111" or count3="0111110111" or count3="1000000111"
or count3="1000010111") then
        premux0_1sel <= '1'; -- beginning cycle of each bit,select
        -- adder0 and adder1 output;
    else
        premux0_1sel <= '0';
    end if;
end process;

process (clk1, reset) -- one more delay due to memory access delay;
begin -- process
    if reset = '0' then -- asynchronous reset (active low)
        muxpipe01 <= '0';
        pipe0 <= '0';
        addr3_pipe0 <= (others => '0');
        we3_pipe0 <= '0';

        elsif clk1'event and clk1 = '1' then -- rising clock edge
            muxpipe01 <= premux0_1sel;
            pipe0 <= premux4_5sel;
            addr3_pipe0 <= addr3;
            we3_pipe0 <= we3;

        end if;
    end process;

    delay5: process (clk1, reset)

```

```
begin -- process delay5
  if reset = '0' then          -- asynchronous reset (active low)
    pipe1 <= '0';
    we3_pipe1 <= '0';
    addr3_pipe2 <= (others => '0');
    mux0_lsel <= '0';
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    pipe1 <= pipe0;
    we3_pipe1 <= we3_pipe0;
    addr3_pipe2 <= addr3_pipe0;
    mux0_lsel <= muxpipe01;
  end if;
end process delay5;

delay6: process (clk1, reset)
begin -- process delay2
  if reset = '0' then          -- asynchronous reset (active low)
    pipe2 <= '0';
    we3_pipe2 <= '0';
    addr3_pipe3 <= (others => '0');
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    pipe2 <= pipe1;
    we3_pipe2 <= we3_pipe1;
    addr3_pipe3 <= addr3_pipe2;
  end if;
end process delay6;

delay7: process (clk1, reset)
begin -- process delay7
  if reset = '0' then          -- asynchronous reset (active low)
    mux4_5sel <= '0';
    we_memle_leunit <= '0';
    addr_memle_leunit <= (others => '0');
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    mux4_5sel <= pipe2;
    we_memle_leunit <= we3_pipe2;
    addr_memle_leunit <= addr3_pipe3;
  end if;
end process delay7;

readalphabet: process (state,alpha0addr,alpha1addr,beta0addr,beta1addr,
                      alpha0laddr_leunit,beta0addr_leunit,beta1addr_leunit)
begin -- read mem_alphabet address selection according to different state;
  if state = alphabeta then
    addra_memalpha0 <= alpha0addr;
    addra_memalpha1 <= alpha1addr;
    addra_membeta0 <= beta0addr;
    addra_membeta1 <= beta1addr;
  elsif state = leunit then
    addra_memalpha0 <= alpha0laddr_leunit;
    addra_memalpha1 <= alpha0laddr_leunit;
    addra_membeta0 <= beta0addr_leunit;
    addra_membeta1 <= beta1addr_leunit;
  else

```

```

    addra_memalpha0 <= (others => '0');
    addra_memalpha1 <= (others => '0');
    addra_membeta0 <= (others => '0');
    addra_membeta1 <= (others => '0');
end if;
end process readalphabet;

-----
writememle: process (state,we_memle_LLR,we_memle_leunit)
begin -- process writememle
    if state = leunit then
        web_memle <= we_memle_leunit;    -- write back the new Le values;
    elsif state = LLR then
        web_memle <= we_memle_LLR;    -- clear the old block's Le value;
    else
        web_memle <= '0';
    end if;
end process writememle;

addrmemle: process (state, addr_memle_g, addr_memle_leunit, addr4,addr_memle_clear)
begin -- process addrmemle
    if state=gamma then
        addra_memle <= addr_memle_g;
    elsif state=LLR then
        addra_memle <= addr4;
    else
        addra_memle <= (others => '0');
    end if;

    if state= leunit then
        addrb_memle <= addr_memle_leunit;
    elsif state=LLR then
        addrb_memle <= addr_memle_clear;
    else
        addrb_memle <= (others => '0');
    end if;
end process addrmemle;

--for LLR_hard decision unit -----
c4: process (clk1, reset)
begin -- process c4
    if reset = '0' then                -- asynchronous reset (active low)
        count4 <= (others => '0');
        we_memle_LLR <= '0';
    elsif clk1'event and clk1 = '1' then -- rising clock edge
        if upcount4 = '1' then
            if count4 < "1010100100" then -- 676="1010100100"
                count4 <= count4 + 1;
            else
                count4 <= (others => '0');
            end if;

            we_memle_LLR <= '1';
        else
            we_memle_LLR <= '0';
        end if;
    end if;
end process c4;

```

```

    end if;
  end if;
end process c4;

addr4 <= count4 when count4 < "1010100100" else (others=>'0');

process (clk1, reset)      -- address delay one cycle for clearing
begin -- process
  if reset = '0' then      -- asynchronous reset (active low)
    addr_memle_clear <= (others=>'0');
  elsif clk1'event and clk1 = '1' then -- rising clock edge
    addr_memle_clear <= addr4;
  end if;
end process;

end rtl;

```

File name: FSM_buf_cfg.vhd
 Purpose: configure file of FSM_buf.vhd

```

library ieee;
use ieee.std_logic_1164.all;

configuration cfg_FSM_buf of FSM_buf is

  for rtl
    for all : addrmemletop
      use configuration work.cfg_addrmemletop;
    end for;

    for all : addrmemalpha0top
      use configuration work.cfg_addrmemalpha0top;
    end for;

    for all : addrmemalpha1top
      use configuration work.cfg_addrmemalpha1top;
    end for;

    for all : addrmembeta0top
      use configuration work.cfg_addrmembeta0top;
    end for;

    for all : addrmembeta1top
      use configuration work.cfg_addrmembeta1top;
    end for;

    for all : addralpha01top
      use configuration work.cfg_addralpha01top;
    end for;

    for all : addrbeta0top
      use configuration work.cfg_addrbeta0top;
    end for;
  end configuration;

```

```
for all : addrbeta1top
  use configuration work.cfg_addrbeta1top;
end for;

for all : multiplier1top
  use configuration work.cfg_multiplier1top;
end for;

for all : multiplier2top
  use configuration work.cfg_multiplier2top;
end for;

end for;
end cfg_FSM_buf;
```

File name: memyktop.vhd and memyktop_cfg.vhd
Purpose: receiver buffer generated by memory core and configure file

```
library ieee;
use ieee.std_logic_1164.all;
```

--As the buffer of received sequence yk(have been quantized), size is 4 bits width, 988 depth for one block;

```
entity memyktop is
  port (
    addr : IN std_logic_VECTOR(9 downto 0);
    clk  : IN std_logic;
    yk_in : IN std_logic_VECTOR(3 downto 0);
    yk_out: OUT std_logic_VECTOR(3 downto 0);
    we   : IN std_logic);
end memyktop;
```

architecture rtl of memyktop is

```
  component mem_yk
    port (
      addr: IN std_logic_VECTOR(9 downto 0);
      clk: IN std_logic;
      din: IN std_logic_VECTOR(3 downto 0);
      dout: OUT std_logic_VECTOR(3 downto 0);
      we: IN std_logic);
  end component;
```

begin -- rtl

```
  myk: mem_yk
    port map (
      addr => addr,
      clk  => clk,
      din  => yk_in,
      dout => yk_out,
      we   => we);
```

```
end rtl;
-----
library XilinxCoreLib;
use XilinxCoreLib.all;

configuration cfg_memyktop of memyktop is

    for rtl

        -- synopsys translate_off

            for all : mem_yk use entity XilinxCoreLib.blkmemsp_v3_2(behavioral)
generic map(
c_has_en => 0,
c_has_din => 1,
c_has_limit_data_pitch => 0,
c_has_sinit => 0,
c_limit_data_pitch => 8,
c_width => 4,
c_sinit_value => "0",
c_addr_width => 10,
c_has_rfd => 0,
c_has_we => 1,
c_depth => 988,
c_write_mode => 0,
c_pipe_stages => 0,
c_has_nd => 0,
c_default_data => "0",
c_has_default_data => 1,
c_mem_init_file => "mif_file_16_1",
c_reg_inputs => 0,
c_enable_rlocs => 0,
c_has_rdy => 0);
            end for;

        -- synopsys translate_on

        end for;
    end cfg_memyktop;

-----
File name: memlctop.vhd and memlctop_cfg.vhd
Purpose: storage memory of channel reliability value, and configure file
-----
library ieee;
use ieee.std_logic_1164.all;

--mem_lc store 10 Lc values corresponding to SNR 5.0db downto 0.5db steps of 0.5
--db; size is 8 bit wodth(3 bits fraction part), 10 deepth;

entity memlctop is

    port (
        clk : in std_logic;
```

```
    addr : in std_logic_vector(3 downto 0);
    Lc   : out std_logic_vector(7 downto 0));

end memlctop;

architecture rtl of memlctop is

    component mem_lc
        port (
            addr: IN std_logic_VECTOR(3 downto 0);
            clk: IN std_logic;
            dout: OUT std_logic_VECTOR(7 downto 0));
    end component;

begin -- rtl

    memlc: mem_lc port map (
        addr => addr,
        clk  => clk,
        dout => Lc);

end rtl;
-----
Library XilinxCoreLib;
use XilinxCoreLib.all;

configuration cfg_memlctop of memlctop is
    for rtl
        -- synopsys translate_off

        for all : mem_lc use entity XilinxCoreLib.blkmemsp_v3_2(behavioral)
            generic map(
                c_has_en => 0,
                c_has_din => 0,
                c_has_limit_data_pitch => 0,
                c_has_sinit => 0,
                c_limit_data_pitch => 8,
                c_width => 8,
                c_sinit_value => "0",
                c_addr_width => 4,
                c_has_rfd => 0,
                c_has_we => 0,
                c_depth => 10,
                c_write_mode => 0,
                c_pipe_stages => 0,
                c_has_nd => 0,
                c_default_data => "0",
                c_has_default_data => 0,
                c_mem_init_file => "mem_lc.mif",
                c_reg_inputs => 0,
                c_enable_rlocs => 0,
                c_has_rdy => 0);
        end for;
    end for;
end configuration;
```

```
-- synopsys translate_on
```

```
end for;
end cfg_memletop;
```

File name: memletop.vhd and memletop_cfg.vhd
 Purpose: storage memory of extrinsic information

```
library ieee;
use ieee.std_logic_1164.all;
```

```
--only info. bits need Le extrinsic value to calculate the log-gamma.so only
--need 26 mem-cell to save the Le for each codeword(32,26); totally for 26
--codewords of one block size, need 26x26 mem-cells which are 12 bit width
--with 3-bit fraction part;
```

```
entity memletop is
```

```
port (
  addra : IN std_logic_vector(9 downto 0);
  addrb : IN std_logic_vector(9 downto 0);
  clka : IN std_logic;
  clkb : IN std_logic;
  Le_in : IN std_logic_vector(11 downto 0);
  web : IN std_logic;
  Le_out : OUT std_logic_vector(11 downto 0));
```

```
end memletop;
```

```
architecture rtl of memletop is
```

```
  component mem_le
    port (
      addra: IN std_logic_VECTOR(9 downto 0);
      addrb: IN std_logic_VECTOR(9 downto 0);
      clka: IN std_logic;
      clkb: IN std_logic;
      dinb: IN std_logic_VECTOR(11 downto 0);
      douta: OUT std_logic_VECTOR(11 downto 0);
      web: IN std_logic);
  end component;
```

```
begin -- rtl
  memle: mem_le
    port map (
      addra => addra,
      addrb => addrb,
      clka => clka,
      clkb => clkb,
      dinb => Le_in,
      douta => Le_out,
      web => web);
end rtl;
```

```
Library XilinxCoreLib;
use XilinxCoreLib.all;

configuration cfg_memletop of memletop is

    for rtl
        -- synopsys translate_off

            for all : mem_le use entity XilinxCoreLib.blkmemdp_v3_2(behavioral)
generic map(
c_depth_b => 676,
c_depth_a => 676,
c_has_rdyb => 0,
c_has_rdyb => 0,
c_has_web => 1,
c_has_wea => 0,
c_sinitb_value => "000",
c_has_doutb => 0,
c_has_douta => 1,
c_has_limit_data_pitch => 0,
c_sinita_value => "000",
c_limit_data_pitch => 18,
c_width_b => 12,
c_width_a => 12,
c_write_modeb => 0,
c_write_modea => 0,
c_addrb_width => 10,
c_has_ndb => 0,
c_has_nda => 0,
c_has_dinb => 1,
c_has_dina => 0,
c_pipe_stages_b => 0,
c_pipe_stages_a => 0,
c_has_rfdb => 0,
c_has_rfda => 0,
c_has_enb => 0,
c_has_ena => 0,
c_reg_inputsb => 0,
c_reg_inputsa => 0,
c_default_data => "000",
c_has_default_data => 1,
c_mem_init_file => "mif_file_16_1",
c_has_sinitb => 0,
c_has_sinita => 0,
c_enable_rlocs => 0,
c_addrb_width => 10);
end for;

        -- synopsys translate_on

    end for;
end cfg_memletop;
```

File name: memlcyktop.vhd and memlcyktop_cfg.vhd

Purpose: internal memory of $L_c \cdot Y_k$

library ieee;

use ieee.std_logic_1164.all;

--Memory for multiplier result. $-21.11 < lcyk < 21.11$, need 8 bits width, for one block, need 988 depth cells;

-- y_k is quantitive result with 4-bits from (1000 to 0111) (-8 to 7), so $lcyk$'s width expands to 12bit with 3 bit fraction part.

entity memlcyktop is

port (

addr: IN std_logic_VECTOR(9 downto 0);

clk: IN std_logic;

lcyk_in: IN std_logic_VECTOR(11 downto 0);

lcyk_out: OUT std_logic_VECTOR(11 downto 0);

we: IN std_logic);

end memlcyktop;

architecture rtl of memlcyktop is

component mem_lcyk

port (

addr: IN std_logic_VECTOR(9 downto 0);

clk: IN std_logic;

din: IN std_logic_VECTOR(11 downto 0);

dout: OUT std_logic_VECTOR(11 downto 0);

we: IN std_logic);

end component;

begin -- rtl

memlcyk : mem_lcyk

port map (

addr => addr,

clk => clk,

din => lcyk_in,

dout => lcyk_out,

we => we);

end rtl;

Library XilinxCoreLib;

use XilinxCoreLib.all;

configuration cfg_memlcyktop of memlcyktop is

for rtl

-- synopsys translate_off

for all : mem_lcyk use entity XilinxCoreLib.blkmemsp_v3_2(behavioral)

generic map(

```
        c_has_en => 0,
        c_has_din => 1,
        c_has_limit_data_pitch => 0,
        c_has_sinit => 0,
        c_limit_data_pitch => 8,
        c_width => 12,
        c_sinit_value => "0",
        c_addr_width => 10,
        c_has_rfd => 0,
        c_has_we => 1,
        c_depth => 988,
        c_write_mode => 0,
        c_pipe_stages => 0,
        c_has_nd => 0,
        c_default_data => "0",
        c_has_default_data => 1,
        c_mem_init_file => "mif_file_16_1",
        c_reg_inputs => 0,
        c_enable_rlocs => 0,
        c_has_rdy => 0);
    end for;

-- synopsys translate_on

    end for;
end cfg_memlcyktop;
```

File name: memlcykle.vhd and memlcykle_cfg.vhd
Purpose: Internal memory of Lc*Yk+Le

```
library ieee;
use ieee.std_logic_1164.all;
```

--store the adder output in the 1st half decoding(horizontal) of last iteration;

entity memlcykletop is

```
port (
    addr : in  std_logic_vector(9 downto 0);
    clk  : in  std_logic;
    din  : in  std_logic_vector(12 downto 0);
    dout : out std_logic_vector(12 downto 0);
    we   : in  std_logic);
```

end memlcykletop;

architecture rtl of memlcykletop is

```
component mem_lcykle
    port (
        addr: IN  std_logic_VECTOR(9 downto 0);
        clk : IN  std_logic;
        din : IN  std_logic_VECTOR(12 downto 0);
```

```
dout: OUT std_logic_VECTOR(12 downto 0);
we : IN std_logic);
end component;

begin -- rtl

memlcykle : mem_lcykle port map (
  addr => addr,
  clk  => clk,
  din  => din,
  dout => dout,
  we   => we);

end rtl;
-----
library xilinxcorelib;
use xilinxcorelib.all;

configuration cfg_memlcykletop of memlcykletop is

  for rtl
    -- synopsys translate_off

    for all : mem_lcykle use entity XilinxCoreLib.blkmemsp_v3_2(behavioral)
      generic map(
        c_has_en => 0,
        c_has_din => 1,
        c_has_limit_data_pitch => 0,
        c_has_sinit => 0,
        c_limit_data_pitch => 8,
        c_width => 13,
        c_sinit_value => "0",
        c_addr_width => 10,
        c_has_rfd => 0,
        c_has_we => 1,
        c_depth => 676,
        c_write_mode => 0,
        c_pipe_stages => 0,
        c_has_nd => 0,
        c_default_data => "0",
        c_has_default_data => 1,
        c_mem_init_file => "mif_file_16_1",
        c_reg_inputs => 0,
        c_enable_rlocs => 0,
        c_has_rdy => 0);
      end for;

    end for;

  -- synopsys translate_on

  end for;
end cfg_memlcykletop;
```

File name: memloggammatop.vhd and memloggammatop_cfg.vhd

Prpose: storage memory of branch transition values

```
library ieee;
use ieee.std_logic_1164.all;
```

--Memory for saving log-gamma value corresponding to Xk=1, 12 bits width, 988depth for one info. block;

entity memloggamma_top is

```
port (
  addra: IN std_logic_VECTOR(9 downto 0);
  addrb: IN std_logic_VECTOR(9 downto 0);
  clka : IN std_logic;
  clkb : IN std_logic;
  dina : IN std_logic_VECTOR(11 downto 0);
  douta: OUT std_logic_VECTOR(11 downto 0);
  doutb: OUT std_logic_VECTOR(11 downto 0);
  wea  : IN std_logic);
```

end memloggamma_top;

architecture rtl of memloggamma_top is

```
component mem_loggamma
  port (
    addra: IN std_logic_VECTOR(9 downto 0);
    addrb: IN std_logic_VECTOR(9 downto 0);
    clka: IN std_logic;
    clkb: IN std_logic;
    dina: IN std_logic_VECTOR(11 downto 0);
    douta: OUT std_logic_VECTOR(11 downto 0);
    doutb: OUT std_logic_VECTOR(11 downto 0);
    wea: IN std_logic);
end component;
```

begin -- rtl

```
memloga: mem_loggamma
  port map (
    addra => addra,
    addrb => addrb,
    clka  => clka,
    clkb  => clkb,
    dina  => dina,
    douta => douta,
    doutb => doutb,
    wea   => wea);
```

end rtl;

```
library XilinxCoreLib;
use XilinxCoreLib.all;
```

configuration cfg_memloggamma_top of memloggamma_top is

```

for rtl
-- synopsys translate_off

    for all : mem_loggamma use entity XilinxCoreLib.blkmemdp_v3_2(behavioral)
        generic map(
            c_depth_b => 988,
            c_depth_a => 988,
            c_has_rdyb => 0,
            c_has_rdyb_a => 0,
            c_has_web => 0,
            c_has_wea => 1,
            c_sinitb_value => "000",
            c_has_doutb => 1,
            c_has_douta => 1,
            c_has_limit_data_pitch => 0,
            c_sinita_value => "000",
            c_limit_data_pitch => 18,
            c_width_b => 12,
            c_width_a => 12,
            c_write_modeb => 0,
            c_write_modea => 0,
            c_addrb_width => 10,
            c_has_ndb => 0,
            c_has_nda => 0,
            c_has_dinb => 0,
            c_has_dina => 1,
            c_pipe_stages_b => 0,
            c_pipe_stages_a => 0,
            c_has_rfdb => 0,
            c_has_rfda => 0,
            c_has_enb => 0,
            c_has_ena => 0,
            c_reg_inputsb => 0,
            c_reg_inputsa => 0,
            c_default_data => "000",
            c_has_default_data => 1,
            c_mem_init_file => "mif_file_16_1",
            c_has_sinitb => 0,
            c_has_sinita => 0,
            c_enable_rlocs => 0,
            c_addrb_width => 10);
        end for;

-- synopsys translate_on
    end for;
end cfg_memloggamma;

```

File name: memalphabetatop.vhd and memalphabetatop_cfg.vhd
Purpose: storage memory of forward/backward recursion

```

library ieee;
use ieee.std_logic_1164.all;

```

--according to state complexity profile. total node=638 for all time instant.we never use log_alpha value at --
-k=32, so memory deepth is designed 637 cells for one codeword decoding. in order to read and write -----
-within one clock cycle,use Dual-port memory; in order to judge one or two branches. add one cell addressd
---637 with zero content.So the total deepth is 638(0 to 637)

entity memalphabetatop is

```
port (  
  addra : in  std_logic_vector(9 downto 0);  
  addrb : in  std_logic_vector(9 downto 0);  
  clka  : in  std_logic;  
  clk b : in  std_logic;  
  dinb  : in  std_logic_vector(16 downto 0);  
  douta : out std_logic_vector(16 downto 0);  
  doutb : out std_logic_vector(16 downto 0);  
  web   : in  std_logic);
```

end memalphabetatop;

architecture rtl of memalphabetatop is

```
component mem_alphabeteta  
  port (  
    addra : IN std_logic_VECTOR(9 downto 0);  
    addrb : IN std_logic_VECTOR(9 downto 0);  
    clka  : IN std_logic;  
    clk b : IN std_logic;  
    dinb  : IN std_logic_VECTOR(16 downto 0);  
    douta : OUT std_logic_VECTOR(16 downto 0);  
    doutb : OUT std_logic_vector(16 downto 0);  
    web   : IN std_logic);  
end component;
```

begin -- rtl

```
memalphabeteta : mem_alphabeteta port map (  
  addra => addra,  
  addrb => addrb,  
  clka  => clka,  
  clk b => clk b,  
  dinb  => dinb,  
  douta => douta,  
  doutb => doutb,  
  web   => web);
```

end rtl;

```
-----  
library xilinxcorelib;  
use xilinxcorelib.all;
```

configuration cfg_memalphabetatop of memalphabetatop is

```
  for rtl  
    -- synopsys translate_off
```

```
for all : mem_alphabeta use entity XilinxCoreLib.blkmemdp_v3_2(behavioral)
  generic map(
    c_depth_b => 638,
    c_depth_a => 638,
    c_has_rdyb => 0,
    c_has_rdyb => 0,
    c_has_web => 1,
    c_has_wea => 0,
    c_sinitb_value => "00000",
    c_has_doutb => 1,
    c_has_douta => 1,
    c_has_limit_data_pitch => 0,
    c_sinita_value => "00000",
    c_limit_data_pitch => 18,
    c_width_b => 17,
    c_width_a => 17,
    c_write_modeb => 0,
    c_write_modea => 0,
    c_addrb_width => 10,
    c_has_ndb => 0,
    c_has_nda => 0,
    c_has_dinb => 1,
    c_has_dina => 0,
    c_pipe_stages_b => 0,
    c_pipe_stages_a => 0,
    c_has_rfdb => 0,
    c_has_rfda => 0,
    c_has_enb => 0,
    c_has_ena => 0,
    c_reg_inputsb => 0,
    c_reg_inputsa => 0,
    c_default_data => "00000",
    c_has_default_data => 1,
    c_mem_init_file => "mif_file_16_1",
    c_has_sinitb => 0,
    c_has_sinita => 0,
    c_enable_rlocs => 0,
    c_addrb_width => 10);
  end for;

-- synopsys translate_on
end for;
end cfg_memalphabetatop;
```
