2010

A New Simplified Algorithm Suitable for Implementation on FPGA for Turbo Codes

Krishnamohan Thangarajah
University of Windsor

A New Simplified Algorithm Suitable for Implementation on FPGA for Turbo Codes

By

Krishnamohan Thangarajah

A Thesis

Submitted to the Faculty of Graduate Studies

Through Electrical and Computer Engineering

In Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science

at the University of Windsor

Windsor, Ontario, Canada

2010

Your file Votre référence
ISBN: 978-0-494-70590-2
Our file Notre référence
ISBN: 978-0-494-70590-2

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

In this thesis, a new algorithm for Turbo codes and a novel implementation of turbo decoder employed with this algorithm is developed. The decoder has an optimal performance in terms of Bit Error Rate(BER) in all Signal to Noise Ratio(SNR) for all frame sizes and any states of Turbo codes. In hardware implementation, we combine the normalization and matrices modules in a single module in order to minimize the internal connection delay which is the bottleneck in hardware implementation, so that the result can be obtained in one single clock signal. Having implemented in this fashion, data rate of 28Mbps for16 state decoder has been achieved. This can be further improved by changing the algorithm for the normalization modules and LLR modules with MAX operator. The matrices modules with the proposed algorithm and the normalization and LLR modules with MAX-LOG-MAP algorithm have been implemented to achieve a data rate of 60Mbps.

A Sincere Dedication

To my ever loving family

Mom, dad, sisters Mala,and Kala,my wife Shammy ,my kids vithu and Jathu,

my nephew Keeran, and my niece Nakeeta.

Your ever lasting support can not be forgettable!

Ohm Sivaya Nama!

Acknowledgement

With utmost sincerity I express my gratitude and respect to my advisors Dr. Behnam Shahrrava and Dr. Mohammed Khalid, who have always inspired me to work with honesty, and discipline. Their timely guidance has been indispensible boons contributing to the completion of this thesis. I am also thankful to my committee members Dr.Robert Kent and Dr. Mitra Mirhassani for their remarkable comments.

I am also thankful to Sundeep Lal and Matt Murawski for their helpful comments, and also extend my note of thanks to Dr. Rashid Rashidzadeh for letting me to use the RCIM lab for the simulations.

This note would be incomplete without thanking graduate departmental secretary Andria Ballo for always being there for valuable guidance.

Table of Contents

Author's Declaration of Originality iii

Abstract iv

A Sincere Dedication v

Acknowledgement vi

List of Figures x

List of Tables xii

List of Abbreviations xiii

CHAPTER 1 INTRODUCTION 1

1.1 Problem Statement 1

1.2 Motivation 2

1.3 Research Methodology 3

1.4 Principal Results 3

1.5 Thesis Organization 4

CHAPTER 2 Turbo Codes and Its Implementation Issues 5

2.1 Fundamentals of Turbo Codes 5

2.1.1 Mathematical Background 8

2.1.2 Literature Survey 10

2.1.2.1 LOG-MAP 12

2.1.2.2 MAX-LOG-MAP 13

2.1.2.3 SIMPLIFIED LOG-MAP 13

2.1.2.4 IMPROVED LOG-MAP 14

2.2 Fixed Point Representation 14

2.3 Implementation Issues of Turbo decoder 14

CHAPTER 3 New Simplified Algorithm Suitable for Implementation on FPGA for Turbo Decoding 16

3.1 New Algorithm 16

3.1.1 Comparison of Complexity of All Algorithms 18
3.2 Software Implementation and Simulation 19
3.2.1 Software Implementation of Turbo Decoder with New Algorithm 19
3.2.2 Floating Point Simulation 21
3.2.2.1 State-4 Turbo Decoder 21
3.2.2.2 State-16 Turbo Decoder 27
3.2.3 Fixed Point Simulation 29
3.3 Observations From the Simulation 32
CHAPTER 4 HARDWARE IMPLEMENTATION AND VALIDATION 34
4.1 Hardware Implementation of Turbo Decoder 34
4.1.1 16-State Turbo Decoder on FPGA 35
4.1.1.1 Trellis Diagram Of the Turbo Decoder 39
4.1.1.2 Gamma Module 40
4.1.1.3 FinalAlpha Module 41
4.1.1.4 FinalBeta Module 49
4.1.1.5 LLRCore of Turbo Decoder 55
4.1.1.6 LECore of Turbo Decoder 58
4.1.1.7 Control Module of Turbo Decoder 59
4.1.2 MyLog104 Module for MAX* 63
4.2 Optimization of Data Rate 65
4.3 Hardware Synthesis And Simulation 66
4.3.1 Hardware Synthesis of Turbo Decoder 68
4.4 Novelties of Implementation 69
4.5 Observation from the Hardware Simulation 70
CHAPTER 5 CONCLUSIONS 71
REFERENCES 73
APPENDIX 76
A1. Matlab codes used in this thesis 76
A2. VHDL Codes for the implementation 93

VITA AUCTORIS.....151

List of Figures

Figure 2-1 Turbo Encoder with code rate 1/3 5
Figure 2-2 RSC encoder with 4 memory elements 6
Figure 2-3 Serial MAP Turbo decoder 7
Figure 3-1 Correction values to be added with MAX 17
Figure3-2 Flowchart for MATLAB simulation of Turbo decoder 20
Figure 3-3 Simulation results for all algorithms 22
Figure 3-4: Simulation result for Low SNR. 23
Figure 3.5 Effect of forward normalization and backward normalization 24
Figure 3.6: Different frame sizes with 0dB SNR. 25
Figure3.7: Different frame sizes for 0.5dB 25
Figure 3.8: Different Frame size for SNR 1 26
Figure 3.9: Different Iterations with frame size 1024 26
Figure 3. 10: From very low to moderate SNR with different Iterations 27
Figure 3. 11 Simulation results for 16-state decoder with 5 iterations for all algorithms. 28
Figure 3. 12 Simulation results of 16 states decoder for MAP and new algorithm with 4 iterations and Improved MAX-LOG-MAP and Simplified MAX-LOG-MAP with 5 iterations. 29
Figure 3. 13 Bit Error Rate for all algorithm with low SNR 30
Figure3. 14 Simulation results for Two different fixed point representations 31
Figure 3. 15 Comparison of fixed point of (10,4) and (8,4) representation 32
Figure 4.1: HDL blocks for the Turbo decoder. 36
Figure 4.2 Trellis Diagram for the generator polynomial [1 0 0 0 1; 1 1 1 1 1] 39
Figure 4.3: Gamma module used as GAMMAF and GAMMAB in the DECODER 40
Figure 4.4: Final alpha for the Turbo Decoder 42
Figure 4.5: Total Alpha module without normalization 44
Figure 4.6 : Basic Block for Alpha module 45
Figure 4.7 : Normalization module for Alpha. 48
Figure 4.8: Final Beta with normalization. 49
Figure 4.9: Final Beta without normalization 52
Figure 4.10: Basic block for backward metric calculation. 53
Figure 4.11: Normalization module for Beta. 54
Figure 4.12: Architecture for finding LLRone. 56
Figure 4.13: Architecture for computing LLRzero. 57

Figure 4.14: LLR module 58
Figure 4.15: Block diagram for extrinsic computation 58
Figure 4.16 : Flow chart for the control module. 61
Figure 4.17: Control module 62
Figure 4.18: Signal flow diagram for the new algorithm. 64
Figure 4.19: Comparison of single algorithm with double algorithm. 66
Figure 4.20: Hardware simulation for both realizations to compare with MatLab simulation 68

List of Tables

Table3.1 Slope and the intersection for the selected regions 18
Table3.2 : Complexity of the algorithms. 18
Table 4.1: Stratix II : EP2S180F150814 features. 35
Table 4.2: Memory and its usage 38
Table 4.3 Description of the modules used in the DECODER 38
Table 4.4 Signal description for GAMMACORE 41
Table 4.5: Signal description for Alpha module 42
Table 4.6: Signal description of FINALBETA. 50
Table 4.7: State and Signal assignment. 59
Table 4.8: Resources and Frequency comparison 67
Table 4.9: Resource comparison for the Turbo Decoder 69
Table 4.10 Comparison of throughput with recent implementation 70

List of Abbreviations

3GPP	Third Generation Partnership Program
4G	Forth Generations
ABS	Absolute
AWGN	Additive White Gaussian Noise
ALUT	Adaptive Look Up Table
BCJR	Bahl Cocke Jeinek Raviv
BER	Bit Error Rate
BPSK	Binary Phase Shift Key
CE	Cross Entropy
CRC	Cyclic Redundancy Check
DSP	Digital Signal Processors
FPGA	Field Programmable Gate Arrays
HDA	Hard Decision Aided
HDL	Hardware Description Language
LLR	Log Likelihood Ratio
MAP	Maximum A posteriori Probability
MAX	Maximum
MULT	Multiplier
NRE	Non Returnable Engineering
RSC	Recursive Systematic Convolutional
SCR	Sign Change Ratio
SISO	Soft Input Soft Output
SNR	Signal to Noise Ratio
SOVA	Soft Output Veterbi Algorithm
VHDL	Very high speed integrated circuit HDL
VLSI	Very Large Scale Integration
UMTS	Universal Mobile Telecommunication System

CHAPTER 1
INTRODUCTION

Channel codes can be classified into two major classes; block codes and convolutional codes. In block codes, one of the information sequence of length k is mapped into a binary sequence of length n, called codeword, and the code rate is defined as k/n. Block codes are memory less, i.e. the codeword depends only on the current k information bits, whereas the convolutional codes have finite-state machines which makes the current codeword depend not only on the current data bit but also state of the finite machine. Turbo code underlies within the convolutional codes.

In this Chapter, the importance of Turbo codes and its usage are addressed, and the motivation for this thesis and principal results are briefly explained.

1.1 Problem Statement

Due to the presence of distortion, noise, and interference, achieving error-free digital communication is not possible without channel coding which basically adds redundant information called parity bits to the data bits to detect and correct the errors. After the invention of Turbo Code [1] which has a BER performance very close to Shannon's theoretical limit, most researchers had been trying to find a practical algorithm that can be implemented in real world applications. These algorithms are optimal at medium to high SNR for small constraint length of the encoder.

The decoding algorithm used in turbo codes can be MAP or SOVA, but MAP has better BER performance than SOVA. Due to the complexity inherited with MAP algorithm makes it impossible to implement in hardware; as a result LOG-MAP algorithm was a feasible solution to MAP without incurring any performance loss. Since the on-going research on improving the data rate while minimizing the BER performance loss, the VLSI implementation of turbo code is not a practical solution due to its NRE cost. The feasible solution is implementing on FPGA which can be reconfigurable when the

modifications need to be done or the standards should be changed according to the research outcomes.

To improve the data rate, researchers came up with windowing technique [22], where the frame size is divided into small frame sizes and each window is parallel processed using the data acquisition in order to find the backward matrices to be initialized. This type of implementation has some drawbacks such as it cannot be used in power limited applications, has some BER performance degradation, not a feasible solution for small frame sizes.

Because of its near Shannon limit performance, turbo codes have been incorporated into many standards such as the consulate committee for space data systems (CCSDS), 3GPP/UMTS, and cdma2000[10], and an standard for IEEE 802.16(WiMax)[11]. For 4G systems, turbo codes should support different frame sizes ranging from 100 to 10,000 and the data rate of 100 Mbps to 1 Gbps. The existing algorithms have BER degradation for large frame size.

The objective of this thesis is to come up with an optimal algorithm for MAX* in order to avoid the errors as much as possible so that the quantization errors due to the fixed point implementation can be minimized, and throughput of the decoder is increased so that it can support the 4G applications.

1.2 **Motivation**

Near Shannon's limit of Turbo codes BER performance instigated to find a feasible algorithm which can achieve a performance similar to LOG-MAP algorithm. High data rate needed for 4G applications motivated this research in order to find a solution without affecting the BER performance for any frame size at all SNR ranging from low to high, and independent of the encoder's constraint length.

1.3 Research Methodology

Research papers related to Turbo decoding and implementation provided by my advisors were analyzed thoroughly to understand the principle of Turbo codes and its implementation issues, and recent publications were also subject to analyze the role of the interleavers and windowing technique in Turbo Codes. Once the principle of Turbo codes was analyzed a new algorithm was developed and its performance was simulated using Matlab to verify its validity for different frame sizes and constraint lengths. Fixed point simulation of the algorithm was done to finalize the fixed point representation that best validates the findings for the hardware implementation.

Once the validity of the new algorithm is assured, the FPGA was chosen to fit the memory and DSP blocks needed for the Turbo decoders. The first step in hardware implementation was to develop the MAX* function, which is the major module that dominates the performance of the Turbo decoder, and checked the output with the floating point calculation to see how it differs from it. After the validation of MAX*, which is named as "mylog104", each individual modules was realized seperately in order to utilize the modular design so that if any changes to be made, the particular module can be modified without affecting the whole design.

Finally all the modules were integrated and tested with the data from the MatLab using a test bench to evaluate the BER performance of the designed Turbo Decoder.

1.4 Principal Results

- A new simplified and implementable algorithm, which has similar BER performance to LOG-MAP for any SNR, constraint length, and frame size, was developed for MAX* function.

- Complete decoder was implemented in FPGA with the assumption of availability of received data and parity information.
- High data rate of 60 Mbps without penalizing the BER performance was achieved
- Number of adders and MAX* functions needed for the normalization module used in Alpha and Beta modules were reduced significantly.

1.5 Thesis Organization

Chapter 2 discusses the Turbo codes and the mathematical background needed for the turbo decoder. The research work related to Turbo Codes and the existing algorithm used in MAX* function and the implementation issues with each algorithm are also presented in this chapter.

Chapter 3 introduces a new algorithm and validity of this algorithm is evaluated with MatLab simulation. All existing algorithms discussed in Chapter 2 are subject to comparison to see the benefit of the new algorithm. And also the fixed point simulation was carried out to best define the wordlength suitable for all SNR ranging from low to high for hardware implementation.

Chapter 4 describes the hardware implementation for 16-state Turbo Decoder and optimization for normalization module and the data rate of the turbo decoder. The evaluation of the turbo decoder is also carried out by comparing with the floating point simulation with MatLab.

Finally Chapter 5 brings out the conclusion of this research work and the benchmark of this thesis. Throughout this thesis we assume the modulation scheme is BPSK and the channel is considered as an AWGN.

CHAPTER 2
Turbo Codes and Its Implementation Issues

In this chapter, fundamentals of turbo codes and its mathematical background will be discussed, and also the research work related to turbo codes and its implementation will be reviewed briefly. The fixed point representation is also discussed for signed numbers and the factors that need to be considered are discussed for the hardware implementation.

2.1 Fundamentals of Turbo Codes

Turbo Codes are two RSC codes concatenated with an interleaver. The Figure 2.1 shows the basic block for the turbo codes. The two RSC encoders can be identical or different. For the particular turbo codes, the code rate is 1/3, but this can be improved by puncturing the parity bit from the encoders; the higher the code rate the better the spectral efficiency.

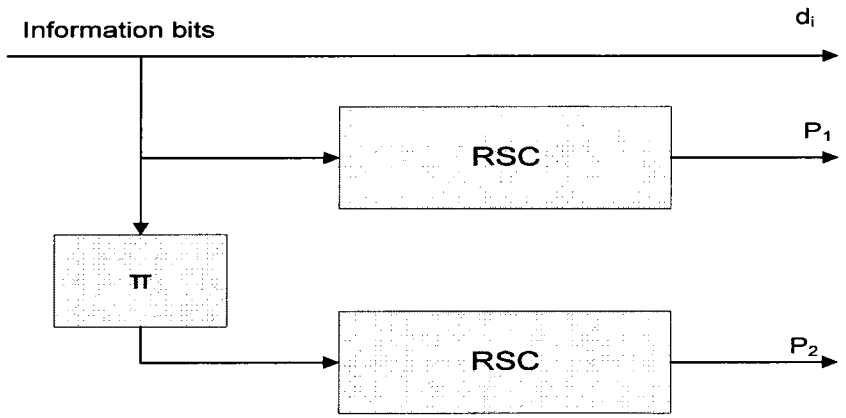

Figure 2-1 Turbo Encoder with code rate 1/3.

The interleaver(Π) is usually selected to be a block pseudorandom interleaver that reorders the bits in the information sequence before feeding them to the second

encoder. However, for high data rate turbo codes, the interleaver must be designed to avoid the memory contention due to the multiple processes trying to access the data from the memory. The purpose of the interleaver is to make the data uncorrelated and produce a code that contains very few code words of low weight, which is called multiplicity that is a factor for the coding gain of the turbo codes.

An important factor in the performance of the turbo code is the length of the interleaver, which is referred to as interleaver gain. With sufficiently large interleaver[1], the performance of the turbo code is very close to the Shannon limit. The data bit along the parity bits are modulated and transmitted to the channel serially. The systematic bit from the second RSC encoder is just ignored.

The RSC codes are given by their generator matrix of the form G(D) = $\left[1 \ \frac{g_2(D)}{g_1(D)}\right]$, where $g_1(D)$ and $g_2(D)$ are the feedback and feed forward polynomial, respectively. The figure 2.2 shows a $(37, 21)_{oct}$ RSC encoder where $g_1(D)$ = [1 1 1 1 1] and $g_2(D)$ = [1 0 0 0 1] corresponding to $g_1(D) = 1 + D + D^2 + D^3 + D^4$ and $g_2(D) = 1 + D^4$.

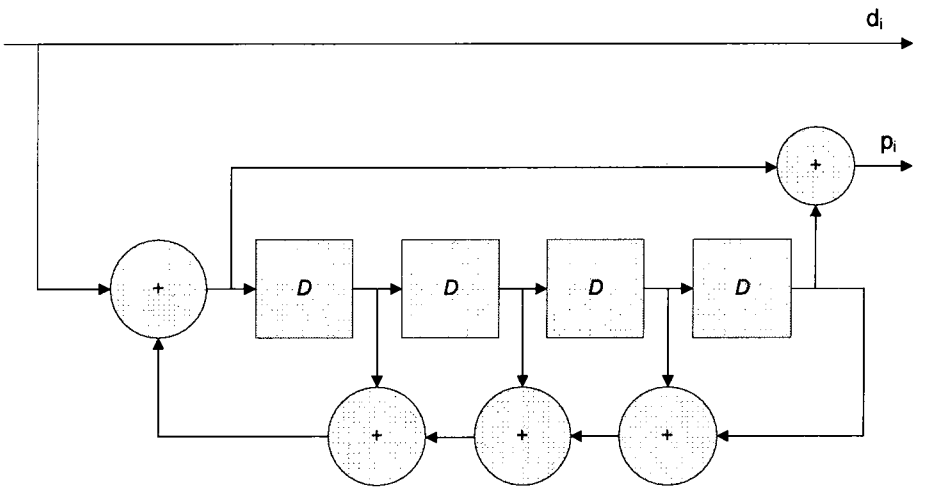

Figure 2-2 RSC encoder with 4 memory elements

In the receiver part, the turbo decoding can start once the data and parity bits are available; in this case the branch matrix and the forward matrix are

calculated and stored for calculating the backward matrix and log likelihood ratios once all the branch matrix and forward matrix are computed. With dual path processing, i.e. backward and forward matrices are calculated simultaneously, the decoder has to wait till all the data and parity bits are completely received. The advantage of the dual path processing is that of doubling the data rate while minimizing the decoding delay [12].

The turbo decoder can be equipped with SOVA or MAP algorithm to decode the data, but MAP algorithm has better BER performance than SOVA [5]. The Figure 2.3 shows a typical turbo decoder with MAP algorithm.

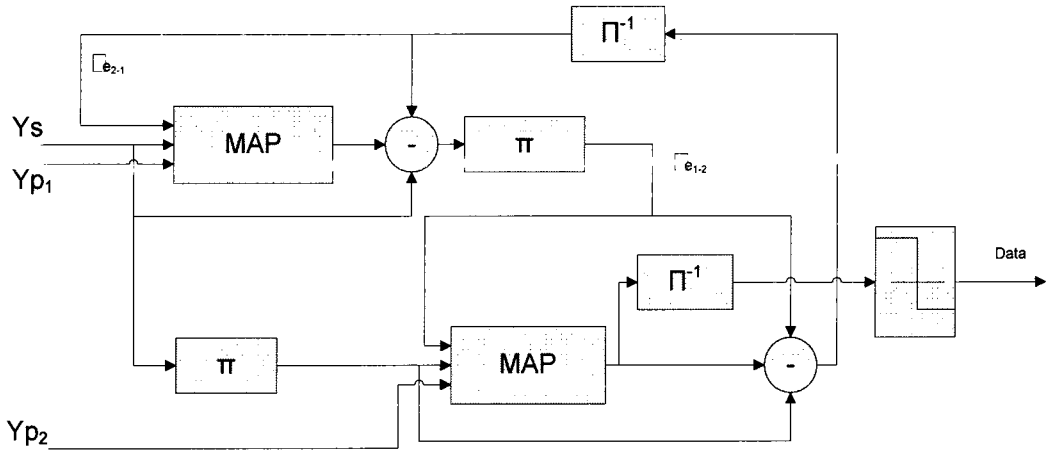

Figure 2-3 Serial MAP Turbo decoder

Initially the extrinsic values for first MAP decoder are set to zero by assuming the equal probability for 1 and 0. The first MAP decoder computes its LLR values from the systematic data, parity data and the extrinsic information. Since the channel information is available and the extrinsic values are from the other MAP decoder, the first MAP decoder must suppress the channel information and the extrinsic values from its LLR value to calculate the extrinsic values to be fed to the next MAP decoder. The extrinsic values and the systematic data are interleaved before second interleaver starts the decoding process.

The second decoder then computes its own LLR values from the channel information which includes interleaved data and the parity bits from the second encoder and computes the extrinsic values to be passed to the first MAP decoder as explained before. The extrinsic information generated by a decoder acts as the a priori probability information for the next stage. This process will be continued iteratively until a stopping criterion is met. At the end of the iteration, the LLR values are used to decode the data by using its sign; positive means 1 and negative means 0. This is also called hard decision making.

Since the extrinsic values, which is referred to as soft values, are the input and the output from each MAP decoder the decoder is sometimes referred to as SISO decoder. More number of iterations is done for getting better BER but once it reached the error floor, which occurs for high SNR, there would not be any significant improvement in BER.

2.1.1 Mathematical Background

The computation of a priority probability is the key factor in turbo decoding algorithm. The MAP decoder, which implements the BCJR algorithm, has to evaluate the LLR defined as:

$$\Lambda(d_k) = log \frac{\Pr(d_k=1|y_1^N)}{\Pr(d_k=0|y_1^N)} \qquad \text{Eq1}$$

Since the encoder has a very short memory, i.e. the output depends only the current state of the encoder and the current input to the encoder, the process can be considered a Markov process [8]. Eq1 can be simplified as:

$$\Lambda(d_k) = log \frac{\sum_{s_k}\sum_{s_{k-1}} \gamma_1(y_k,s_{k-1},s_k).\alpha_{k-1}(s_{k-1}).\beta_k(s_k)}{\sum_{s_k}\sum_{s_{k-1}} \gamma_0(y_k,s_{k-1},s_k).\alpha_{k-1}(s_{k-1}).\beta_k(s_k)} \qquad \text{Eq2}$$

Where,

$$\gamma_i\left[\left(y_k^s, y_k^p\right), s_{k-1}, s_k\right] = p(y_k^s|d_k = i)\,.p\left(y_k^p|d_k = i, s_k, s_{k-1}\right) .q(d_k = i|s_k, s_{k-1}).\Pr\{s_k|s_{k-1}\} \quad \text{Eq3.}$$

$$\alpha_k(s_k) = \sum_{s_k}\sum_{i=0}^{1}\gamma_i(y_k, s_{k-1}, s_k).\alpha_{k-1}(s_{k-1}) \quad \text{Eq4.}$$

$$\beta_k(s_k) = \sum_{s_{k+1}}\sum_{i=0}^{1}\gamma_i(y_{k+1}, s_k, s_{k+1}).\beta_{k+1}(s_{k+1}) \quad \text{Eq5.}$$

If the initial state of the encoder is known, i.e. s_0 = 0 then

$$\alpha_0(m) = \Pr\{s_0 = m\} = \begin{cases}1\ m = 0\\ 0\ m \neq 0\end{cases}$$

And the encoder is terminated with known state, i.e. s_N= 0 then

$$\beta_N(m) = \Pr\{s_N = m\} = \begin{cases}1\ m = 0\\ 0\ m \neq 0\end{cases}$$

If the encoder is not terminated, then

$\beta_N(m) = \Pr\{s_N = m\} = 1/P$, where P is the number of states of the encoder. In our case, the first encoder is terminated to a known state whereas the second encoder is left open, i.e. can have any one of the 16 states.

In Eq3, the value of $q(d_k = i|s_k, s_{k-1})$ is either one or zero depending on whether bit I is associated with the transition from state s_{k-1} to s_k or not [3]. Since there is no parallel transition, i.e. only one transition is possible; the followings are the facts [3].

$\Pr\{s_k|s_{k-1}\} = \Pr\{d_k = 1\}$ when $q(d_k = 1|s_k, s_{k-1})$ = 1and

$\Pr\{s_k|s_{k-1}\} = \Pr\{d_k = 0\}$ when $q(d_k = 0|s_k, s_{k-1}) = 1$

2.1.2 Literature Survey

Since the introduction of turbo codes [1] and its capability of near Shannon limit error correction performance, a lot of interest had been raised to find practical decoding algorithms for implementation in real systems. The original BCJR [6] algorithm used in MAP, cannot be realized in hardware due to its complex probability functions and non-liner functions. The modified BCJR [3], which is a logarithmic version of the original algorithm, was proposed by Patrick, Peter, and Emmanuelle, which in turns, brought other algorithms such as MAX-LOG-MAP, simplified MAX-LOG-MAP and improved MAX-LOG-MAP. Amongst these algorithms the MAX-LOG-MAP is not sensitive to SNR mismatch [9] and requires a max operation only. To improve the performance of the MAX-LOG-MAP, a simple look-up table [3] or a threshold detector [2] was utilized as a correction value. Each of these algorithms is discussed in the following sections.

Due to the iterative nature of the decoding process, a high computational complexity is inevitable; as a result the latency and energy consumption increase linearly with the number of iterations. Particularly, more number of iterations would not improve the BER at high SNR due to the fast convergence to the error floor. And also the effectiveness of the decoding process strongly depends on the channel characteristics, which can change from block to block due to the noise and fading. In some cases even with very large number of iterations, it is impossible to have a successful decoding. Therefore some stopping criterions must be implemented in turbo decoders. Sum-Reliability, and combined minimum LLR and sum reliability stopping criterions were proposed in [23]. In sum-reliability, the sum of absolute value of LLR is computed after each iteration and compared with the previous sum-reliability value. If the sum-reliability for the current iteration is less than the previous one the decoding process is halted. In the second stopping criterion, a threshold value is used to compare with the minimum value of LLR and the sum-reliability is also used. Having met either one of the conditions makes the decoding process to be stopped. Other stopping criterions such as

CE [24], mean-reliability [25], minimum LLR [26], SCR [27], HAD [27], and CRC [28] are also proposed in the literature.

Due to the complexity of the MAP algorithm, it is impossible to design a high throughput turbo decoder unless the windowing technique is employed, wherein several MAP processors operate on smaller sized windows within each received frame [8]. This is very essential for 4G applications where the peak data rate is in the range of 200Mbps [29] with extremely tight delay constraints. In windowing technique, the frame length is divided into smaller sizes and each window is associated with its own MAP processor. For the windowing technique data acquisition is performed in order to initialize the backward matrix to reduce the errors in the initialization. The problem with the windowing technique is the memory contention [8], where multiple processors try to access the same memory simultaneously. Using the buffers [33] or modified memory addressing [34], memory contention can be resolved, and also specifically designed interleaver [30, 31, 32] can be employed to totally avoid the memory contention. If an interleaver is contention free for all window sizes that divide the interleaver length, it is called a maximum contention free interleaver [8]. The advantage of the maximum contention free interleaver is that there is no restrictions on selecting the window size other than it should divide the interleaver length. Quadratic permutation polynomials over integer rings and maximum contention free permutation polynomials interleaver were proposed in [8]. The contention free requirements are discussed in [29].

The implementation of turbo codes can be classified into serial and parallel architectures. In serial architecture one MAP decoder is utilized to decode the whole frame length, whereas in parallel architecture frame length is considered into several sub blocks which can be parallel processed using MAP decoders associated with each sub block in order to reduce the latency and in the mean time increasing the throughput which is directly proportional to the number of sub blocks. The parallel architecture can use the contention free interleavers [8] or the network on chip [35] in order to avoid the memory contention. The bit width that represents the data, parity, and extrinsic

information that stored in the memory has an impact on power consumption since the most of the power consumption comes from power dissipation from the memory [35]. The optimization of bit width for extrinsic information was proposed in [5].

Since the throughput of the decoder is strongly dependent upon the MAX* operator used in MAP decoder, and the recursive computations needed for the LLR calculations, tree architecture[20] is inevitable to reduce the number of clock cycles for the recursions and the critical path delay of the LLR module. Dual path processing [12] and radix-4 [16] are also used to improve the throughput, however the radix-4 implementation has some BER degradation.

2.1.2.1 LOG-MAP

In Eq3, the probability functions can be written as

$$p(y_k^p|d_k = i, s_k, s_{k-1}) = \frac{1}{\sqrt{\pi N_0}} . e^{\frac{-1}{N_0}[y_k^p - x_k^p(i,s_k,s_{k-1})]^2} \quad \text{Eq6.}$$

$$p(y_k^s|d_k = i) = \frac{1}{\sqrt{\pi N_0}} . e^{\frac{-1}{N_0}[y_k^s - x_k^s(i)]^2} \quad \text{Eq7.}$$

Due to the exponential terms in Eq6 and Eq7, the Eq3, 4, and 5 are calculated in logarithmic domain to reduce the complexity in MAP. These equations can be written as

$$ln\gamma_i[(y_k^s, y_k^p), s_{k-1}, s_k] = \frac{2y_k^s x_k^s(i)}{N_0} + \frac{2y_k^p x_k^p(i, s_k, s_{k-1})}{N_0} + ln\Pr\{s_k|s_{k-1}\} + K \quad \text{Eq8.}$$

$$ln\alpha_k(s_k) = ln\left(\sum_{s_{k-1}} \sum_{i=0}^{1} e^{ln\gamma_i[(y_k^s, y_k^p, s_{k-1}, s_k] + ln\alpha_{k-1}(s_{k-1})}\right) \quad \text{Eq9.}$$

$$\ln\beta_k(s_k) = ln\left(\sum_{s_{k-1}} \sum_{i=0}^{1} e^{ln\gamma_i[(y_{k+1}^s, y_k^p, s_{k+1}, s_k] + ln\beta_{k+1}(s_{k+\{\{1\}}}\right) \quad \text{Eq10.}$$

The value K in Eq8 can be ignored [3].

For simplicity we refer to log values as

$$\bar{\alpha}_k(s_k) = \ln(\alpha_k(s_k))$$

$$\bar{\beta}_k(s_k) = \ln(\beta_k(s_k))$$

$$\bar{\gamma}_i[(y_k^s, y_k^p), s_{k-1}, s_k] = ln\gamma_i[(y_k^s, y_k^p), s_{k-1}, s_k]$$

Using the Jacobean expansion, we can write

$$\ln(e^x + e^y) = \max(x, y) + \ln(1 + e^{-|x-y|}) \quad \text{Eq11.}$$

Therefore the LLR can be written as:

$$\Lambda(d_k) = \underset{(S_k, S_{k-1}), d_k=1}{\widehat{\max}} \left(\bar{\alpha}_{k-1}(S_{k-1}) + \bar{\gamma}_k(S_{k-1}, S_k) + \bar{\beta}_k(S_k)\right)$$

$$- \underset{(S_k, S_{k-1}), d_k=0}{\widehat{\max}} \left(\bar{\alpha}_{k-1}(S_{k-1}) + \bar{\gamma}_k(S_{k-1}, S_k) + \bar{\beta}_k(S_k)\right) \quad \text{Eq12.}$$

Where

$$\widehat{max}(x, y) = \text{MAX}^* = \max(x, y) + \ln(1 + e^{-|x-y|})$$

In Eq11, the second term is considered the correction function. Using different methods to find this correction function, MAP algorithm can be referred to MAX-LOG-MAP, simplified MAX-LOG-MAP, and improved MAX-LOG-MAP.

2.1.2.2 MAX-LOG-MAP

In MAX-LOG-MAP, the correction function is ignored. This ignorance will have a significant impact at 0 to 3dB SNR values, but not in SNR values greater than 5dB. Even in the range of 3 to 5dB SNR values, more number of iterations may be needed to achieve the same BER compared to other algorithm. But from the hardware point of view, it is the least complex algorithm amongst the existing algorithms and also it is not sensitive to SNR mismatch [9].

2.1.2.3 SIMPLIFIED LOG-MAP

In this algorithm [2], the correction function has two values. If the difference is less than 2 a value 0.375 is added to the max operation, otherwise zero is added. This

algorithm also has some variation over MAP algorithm, because the errors due to the quantization are not evenly distributed.

2.1.2.4 IMPROVED LOG-MAP

McLaren series is employed in this algorithm [4], to find the correction function as explained below:

$$\log(1+\exp(-|x-y|)) \approx log2 - \frac{1}{2}|x-y|$$

$$\approx \max(0, log2 - \frac{1}{2}|x-y|)$$

This will work great for the absolute value of the difference less than 1.3863, and also shift operation and the one more max operation are the overhead in the hardware implementation, and have poor BER performance for higher order of turbo decoders. Even though this algorithm also has some deviation from the MAP at 0 to 2dB SNR values, it is better than the simplified algorithm when the absolute values are less than 1.3863. In our case, we will compare this algorithm and the MAP algorithm for the fixed point analysis.

2.2 Fixed Point Representation

The fixed point representation is very important in FPGA implementation of any digital circuit since the floating point implementation is costly and power consuming. A signed fixed point is represented as A(a,b), where a is the word length and b is the fraction length. The range of the representation is from -2^{a-b} to $-2^{a-b}- 1/2^{b}$.

2.3 Implementation Issues of Turbo decoder

When the turbo decoder is to be designed, we need to consider some design constraints such as area, power, memory requirements, throughput and latency, and the

acceptable BER. The fixed point representation plays a major role in deciding constraints mentioned above. Since the received data is influenced with the noise, the SNR decides the range of the received data; as a result, the fixed point representation must include all the data from minimum to maximum. If the range of the fixed point representation is too large most of the input will be quantized by a zero value, i.e. an erasure [21]. On the other hand, if the range is too small most of the input values are saturated and leads the hard decision to soft quantization [21].

The data rate and the latency of the turbo decoder is the inverse of the maximum critical path delay from the LLR module and the normalization module. Fortunately the BER performance of the turbo decoder is not sensitive to either of these two computations. In contrast, branch, forward, and backward matrices calculations are the ones determine the BER performance of the turbo decoder in addition to the other factors such as interleaver and generator polynomial. Therefore the selection of the algorithm for these calculations can be a relaxed algorithm in order to increase the data rate while maintaining the BER performance. And also this will have a significant reduction in area and power consumption.

The selection of the algorithm also has an impact on the constraints mentioned above. The simple algorithm which gives the high throughput while giving very poor BER performance is MAX-LOG-MAP. Therefore it is totally dependent upon designer's choice of what algorithm needs to be chosen and what fixed point representation must be considered based on the applications.

CHAPTER 3
New Simplified Algorithm Suitable for Implementation on FPGA for Turbo Decoding

In this chapter, a new algorithm is derived based on the correction function mentioned in the previous chapter. The simulation of the new algorithm with the existing algorithms is analysed. Fixed point simulation is also performed in order to come up with a suitable word length which does not affect the BER performance in any SNR.

3.1 New Algorithm

The objective of developing this new algorithm is to have a BER performance similar to LOG-MAP at low SNR applications. The development of the algorithm stars with the correction function in MAX* operation. The MAX* operation is defined as:

$$\mathrm{MAX}^*(x,y) = \max(x,y) + \ln\left(1 + e^{-|x-y|}\right) \qquad \text{Eq13.}$$

The Figure 3.1 shows the correction values versus the absolute difference of x and y. The function can be modeled with linear functions by considering the whole range into sub ranges covering from 0 to 4, and rest of the range can be ignored. The correction values have significant effect when the absolute difference is between 0 and 1 for low SNR application.

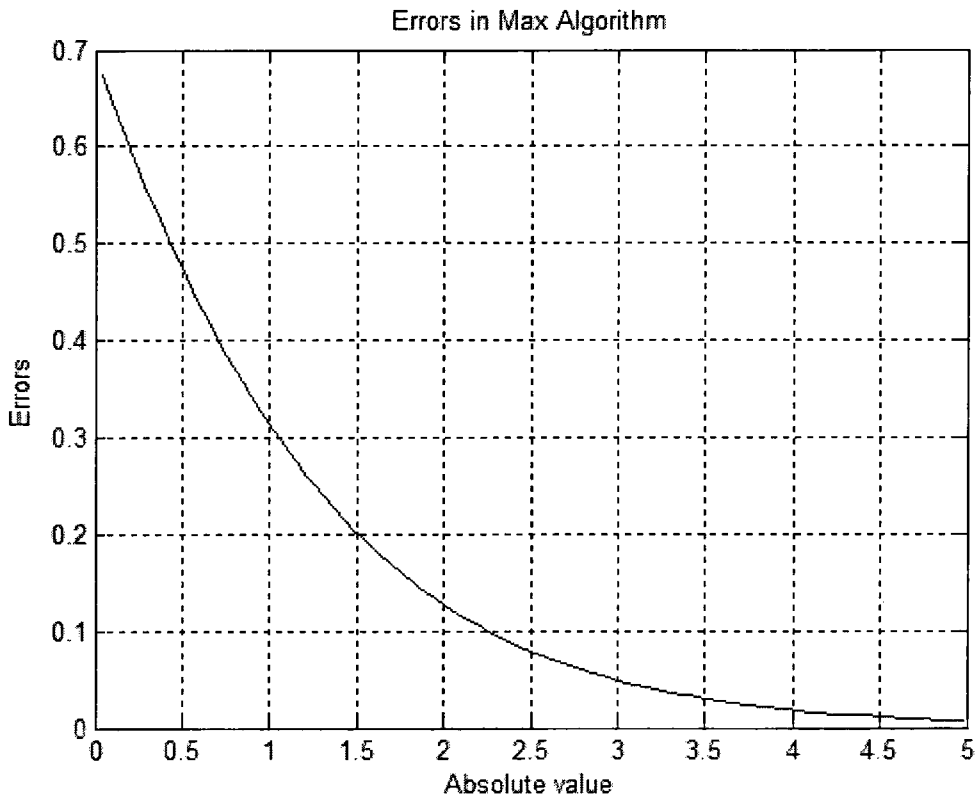

Figure 3-1 Correction values to be added with MAX

Therefore the correction function can be written as:

$$\log\left(1 + e^{-|x-y|}\right) = m * |x - y| + c,$$

where m and c are the slope and the intersection of the linear functions of each region shown in Table 3-1.

Since the region between 1 and 2 in Figure 3-1 has the maximum curvature, it is considered into two regions to reduce the effects of correction errors. It is noteworthy to point out these slopes and constants do not depend on the SNR. Having reduced the correction errors have a better BER performance in all SNR ranging from very low to high and also it has minimal effect on quantization errors in hardware implementation.

Table3.1 Slope and the intersection for the selected regions

|x-y|	Slope(m).	Intersection(c) .
0 to 1	-0.3788	0.6931
1 to 1.5	-0.2238	0.5371
1.5 to 2	-0.1490	0.4249
2 to 3	-0.0783	0.2835
3 to 4	-0.0305	0.1401

3.1.1 Comparison of Complexity of All Algorithms

Table 3-2 describes the complexity of the algorithm with the existing algorithm in algorithmic point of view.

Table3.2 : Complexity of the algorithms.

Operation	Simplified MAX	Improved MAX	Proposed Algorithm
max	1	2	1
multiplication	0	0	1
abs	1	1	1
addition	2	3	3

The proposed algorithm has a multiplier to be implemented, but today's technology allows implementing very fast multipliers in FPGA. To reduce the area and the critical path delay for the multiplier, the slope and intersection can be represented in (4, 4) with unsigned format, and the absolute different can be truncated to (6, 4) with unsigned format as well so that the multiplier inputs have width 4 and 6, since the slope and intersection values are always less than 1, and correction values are neglected after 4.

3.2 Software Implementation and Simulation

MATLAB simulations of the Turbo decoder with new algorithm are carried out in this chapter, and the results are presented. For the simulation, different algorithms discussed in Chapter 2, are considered for the sake of comparison with the new algorithm, and various frame sizes with fixed SNR are also taken into account to validate the performance of the turbo decoder with the frame size. The fixed point simulation is also carried out to determine the best wordlength and fraction length for all SNR in order to implement in the hardware. The results from the MATLAB simulation validate the developed algorithm and form the basis for the HDL implementation of the same.

3.2.1 Software Implementation of Turbo Decoder with New Algorithm

Following the literature survey and the development of the new algorithm, the next step is a detailed MATLAB simulation of the proposed algorithm. MATLAB R2009a Version 7.8 has been used to develop the code to verify the turbo decoder performance in terms of BER.

There are three stages of testing the algorithm in MATLAB:

1. Test the algorithm with fixed frame size and different SNR.
2. Test the algorithm with fixed SNR and various frame sizes.
3. Test the algorithm with fixed frame size and various SNR with fixed point representation

The flowchart in Figure 3.2 shows the sequential of the MATLAB program used to simulate the turbo decoder (see Appendix for complete code listing).

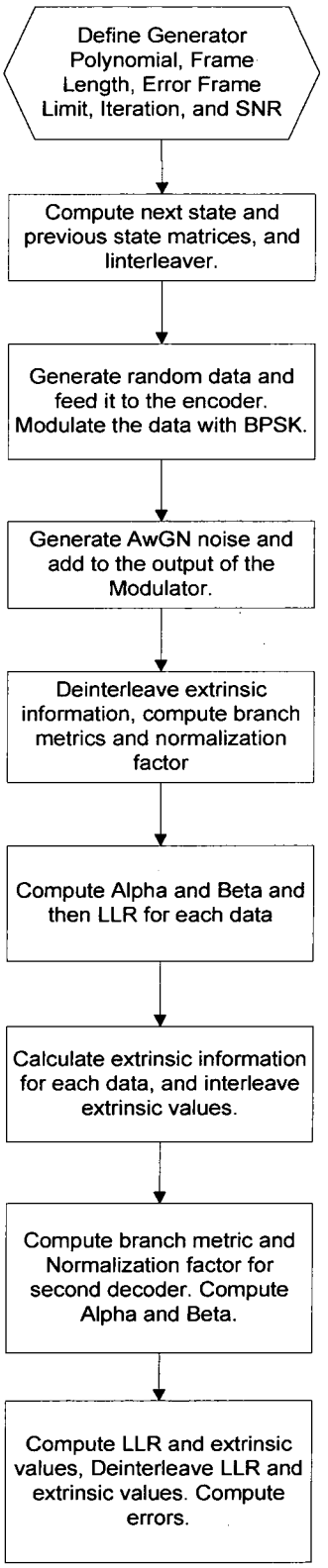

Figure3-2 Flowchart for MATLAB simulation of Turbo decoder

3.2.2 Floating Point Simulation

For floating point simulation, two constraint length decoders were considered in order to evaluate the BER for state-4 and state-16 decoders. The whole purpose of doing both state decoders is to show how the existing algorithms fail to show an optimal BER performance when the algorithms are applied to high order decoders.

3.2.2.1 State-4 Turbo Decoder

- Generator Polynomial [1 0 1; 1 1 1].
- Random Interleaver.
- AWGN channel.
- Frame error limit – 25
- Frame size 1024.

For the simulation, encoder 1 is brought to a known state; state0, by adding 2 tail bits to the frame and encoder 2 is left open; it can have any state at the end of the frame bit plus tail bits. As a result, the following matrices are initialized as:

- Encoder 1 forward and backward matrices are set to [0, infty, infty, infty].
- Encoder 2 forward matrix is set to [0, infty, infty, infty].
- Encoder2 backward matrix is set to [log0.25, log0.25, log0.25, log0.25].
- Extrinsic values are set to 0.

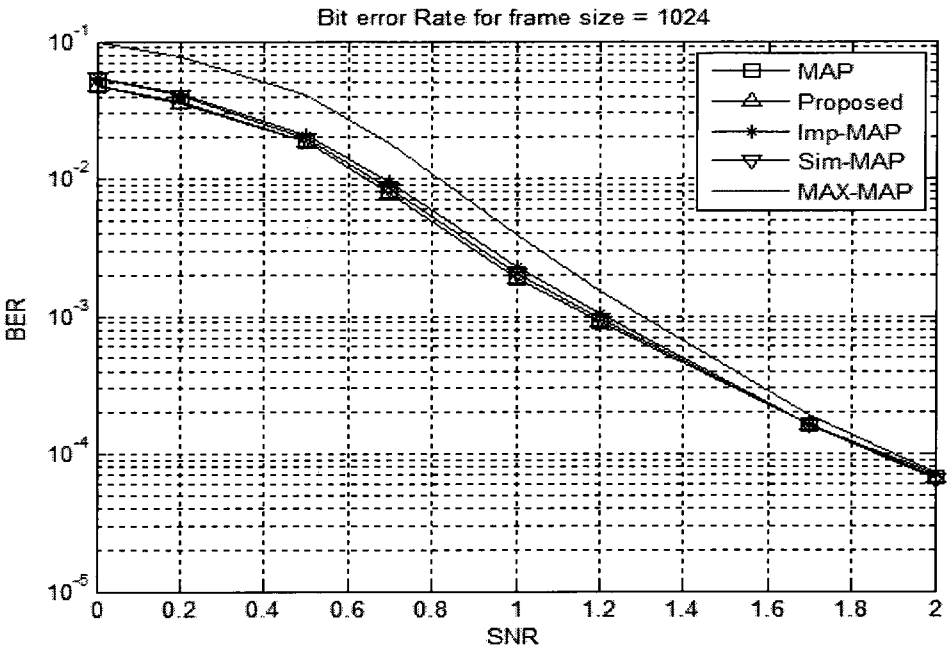

Figure 3-3 Simulation results for all algorithms

From the simulation, it can be seen that all algorithms except the proposed algorithm, have some deviation from the original MAP algorithm, and also they exhibit almost same BER performance after 1.9dB SNR.

The following simulation is done to show the results clearly at low SNR.

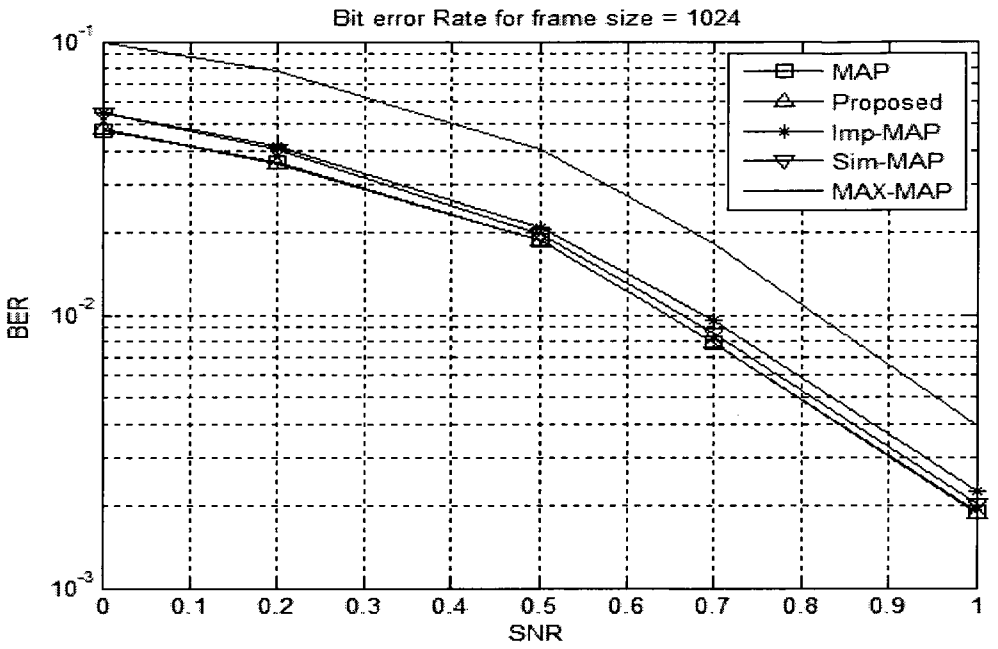

Figure 3-4: Simulation result for Low SNR.

The following simulation was done to show that the normalization of alpha and beta can be done either using total alpha or total beta. This is very important in hardware implementation for the dual processes in order to optimize the data rate. The Figure 3.5 shows that both normalization end with similar BER performance.

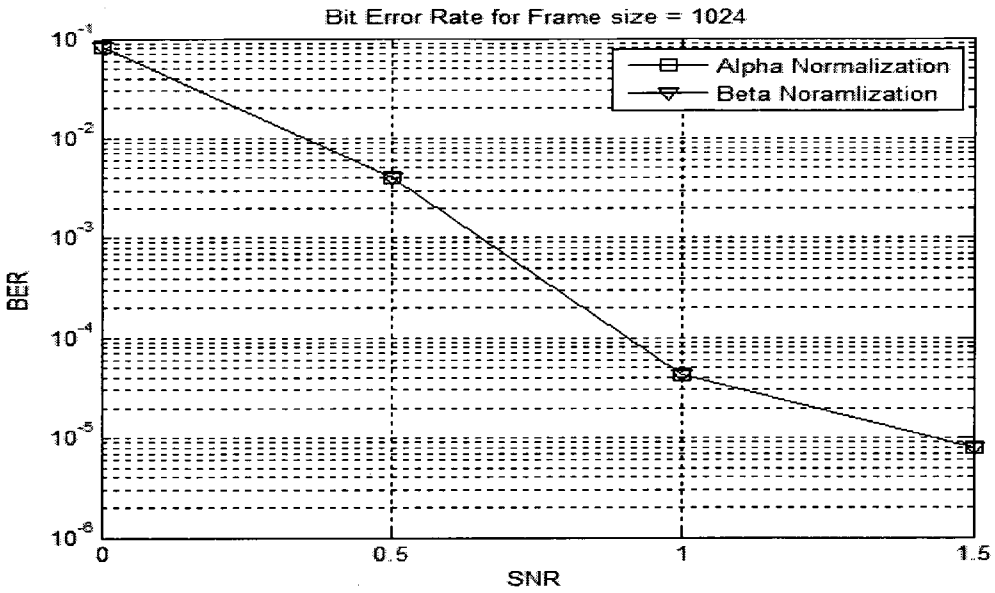

Figure 3.5 Effect of forward normalization and backward normalization

Figure 3.6 to 3.9 show the effect of the frame size for all the algorithm. Frame sizes were considered from 100 to 4000. It can be seen from the simulation results, when the SNR decreases, the performance of the other algorithm deviates from the original MAP algorithm due to the error propagation along the frame size.

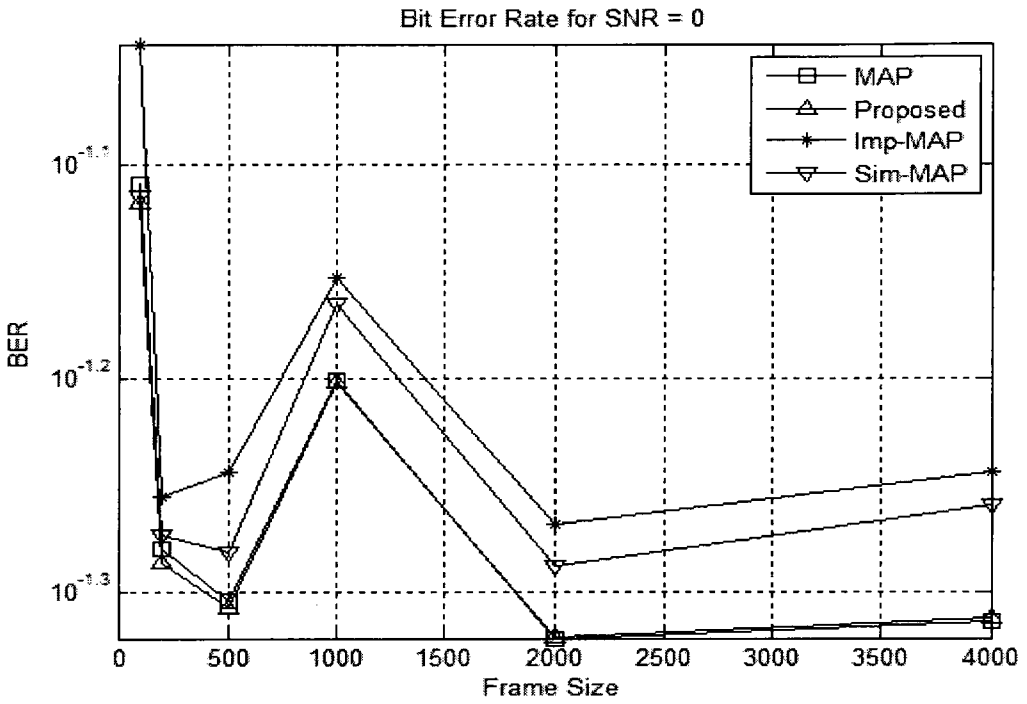

Figure 3.6: Different frame sizes with 0dB SNR.

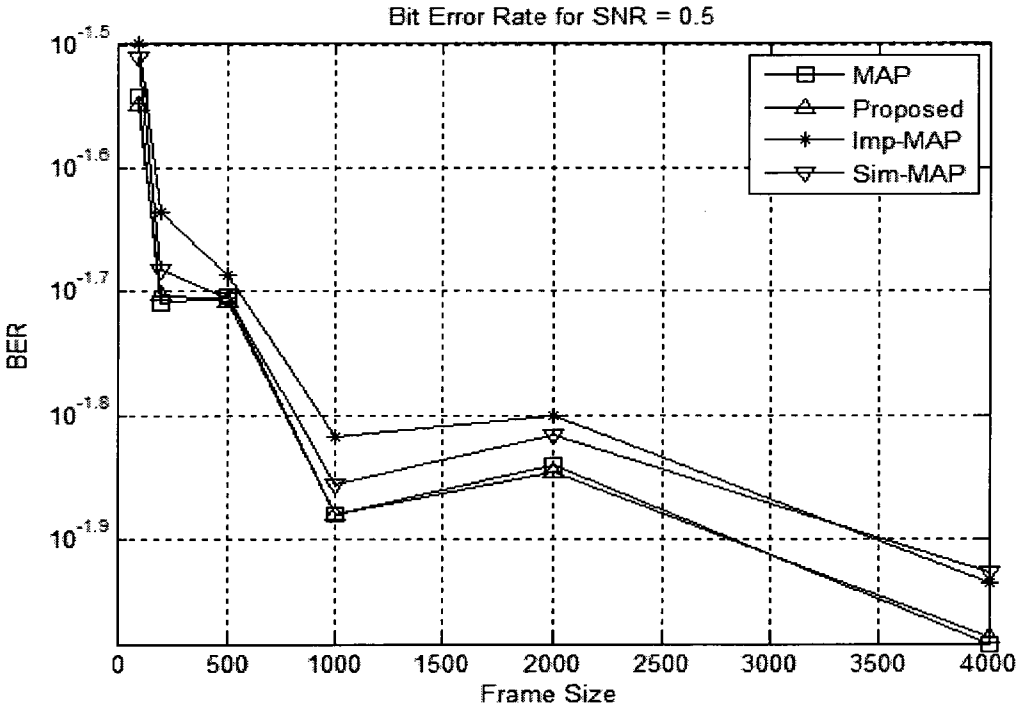

Figure3.7: Different frame sizes for 0.5dB .

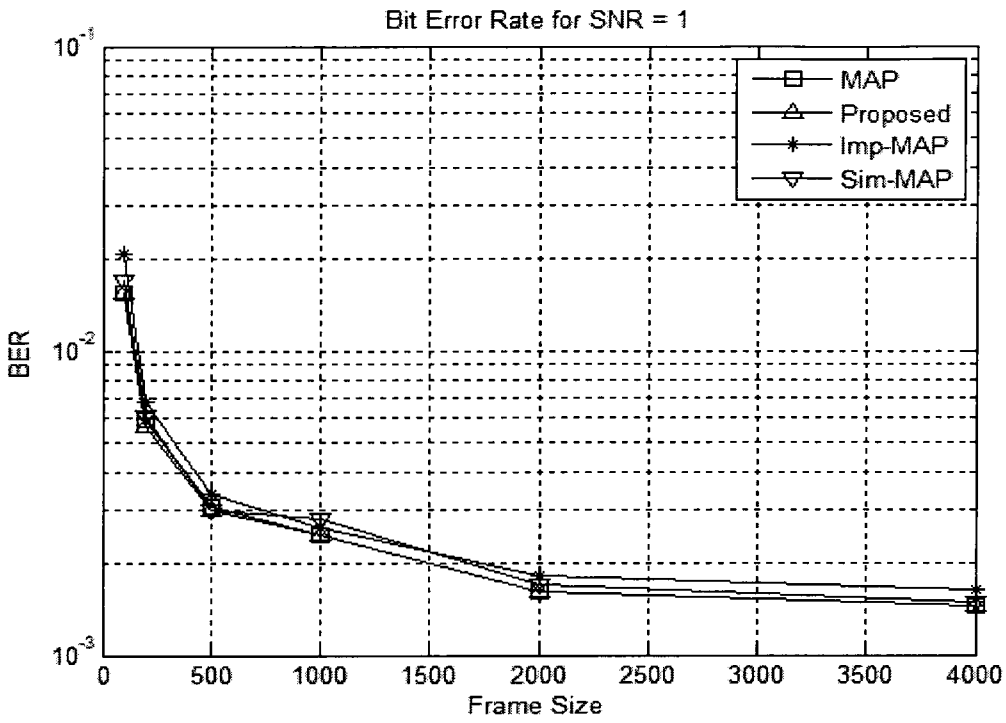

Figure 3.8: Different Frame size for SNR 1.

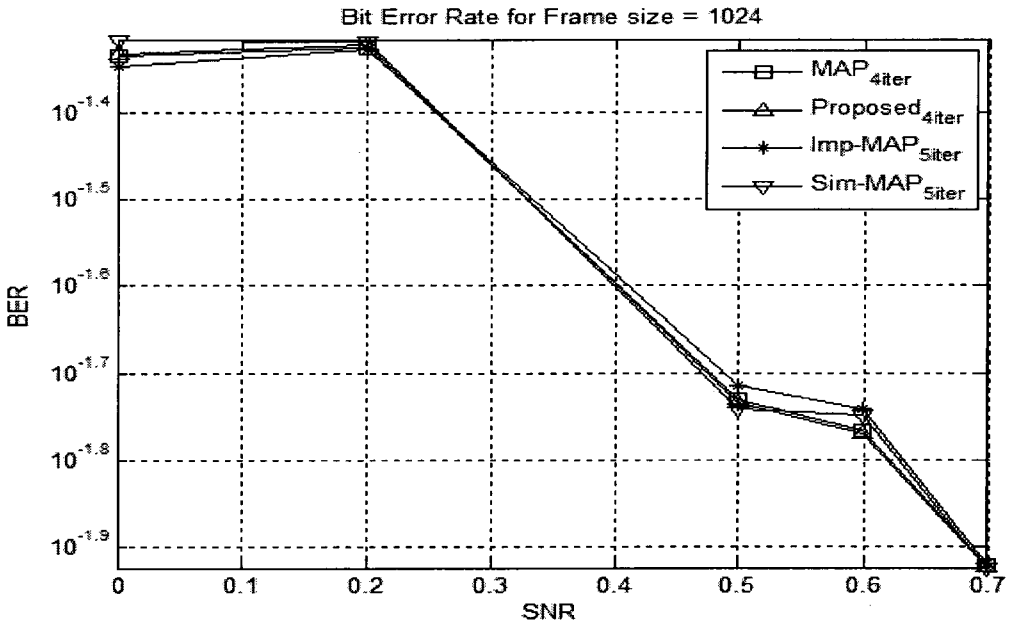

Figure 3.9: Different Iterations with frame size 1024.

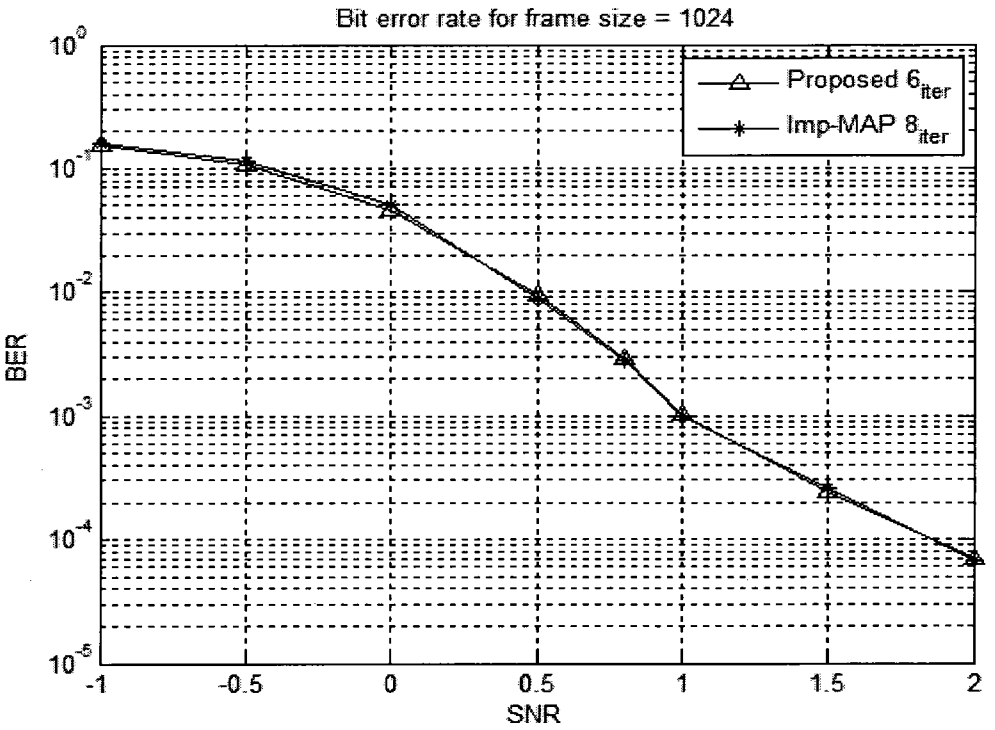

Figure 3. 10: From very low to moderate SNR with different Iterations

From figure 3.9, the number of iterations needed to get the same BER for the proposed algorithm is 4 while the other algorithm needs 5 iterations. Simulation result in figure 3.10 compares the proposed algorithm and improved MAX-LOG-MAP with 6 iterations and 8 iterations respectively, from very low to moderate SNR. Increasing the number of iterations will increase the latency of the decoder which is directly proportional to the frame size.

3.2.2.2 State-16 Turbo Decoder

- Generator Polynomial [1 0 0 0 1; 1 1 1 1 1].
- Random Interleaver.
- AWGN channel.
- Frame error limit 25
- Frame size 1024.

For simulation, encoder 1 is brought to a known state; state0, by adding 4 tail bits to the frame and encoder 2 is left open; it can have any state at the end of the frame. The data and the tail bits along with the parity bits are transmitted through an AWGN channel. As a result, the following matrices are initialized as:

- Encoder 1 forward and backward matrices are set to [0, infty=> 15].
- Encoder 2 forward matrix is set to [0, infty=>15].
- Encoder2 backward matrix is set to [log(1/16) =>16].
- Extrinsic values are set to 0.

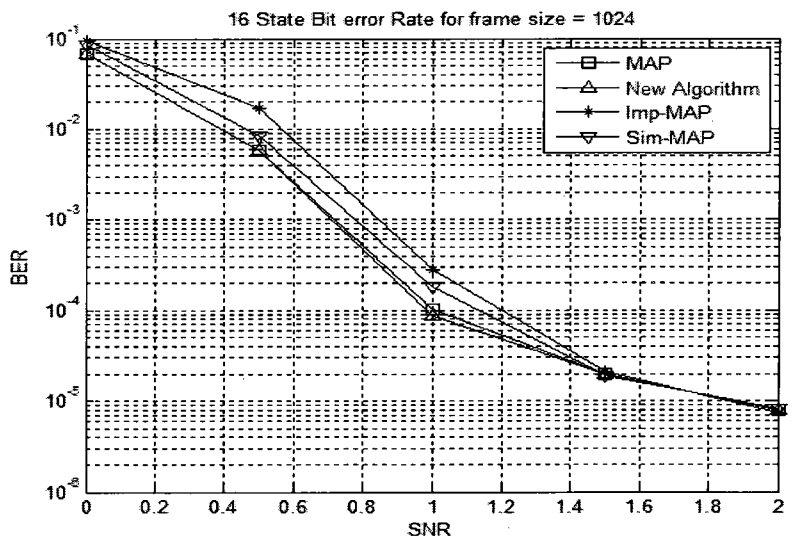

Figure 3. 11 Simulation results for 16-state decoder with 5 iterations for all algorithms.

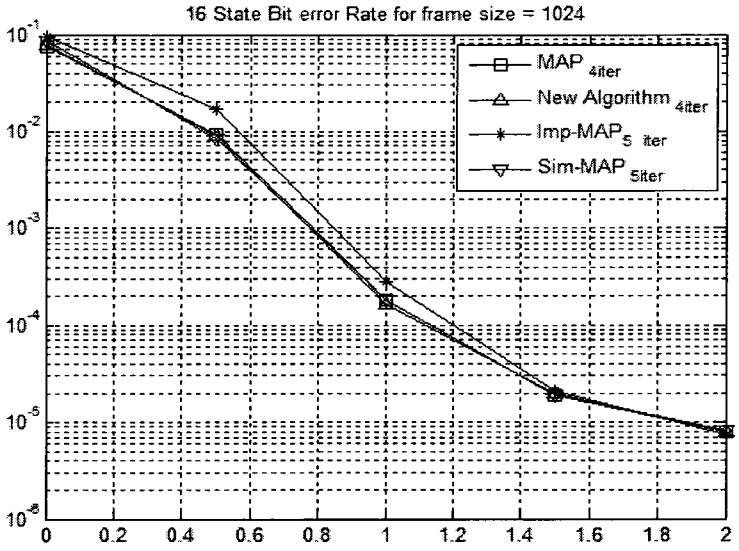

Figure 3. 12 Simulation results of 16 states decoder for MAP and new algorithm with 4 iterations and Improved MAX-LOG-MAP and Simplified MAX-LOG-MAP with 5 iterations.

From Figures 3.11 and 3.12 it can be concluded that these two algorithms depend on the number of states of the decoder, whereas the proposed algorithm shows optimal BER performance. It is to note that most of the satellite communications use 16-states Turbo Codes to have better BER performance.

3.2.3 Fixed Point Simulation

The following are the modes for the fixed point simulation.

- Round mode - Nearest
- Overflow mode – Saturate
- Product Wordlength – 8,10bits
- Product Fraction length – 2,4 bits
- Sum word length - 8,10 bits
- Sum Fraction length – 2,4 bits
- Frame size - 400

- Random Interleaver

Due to the time consuming simulation for the fixed point representation, frame size was considered as 400. The whole purpose of the fixed point simulation is to find the optimal word length and fraction length for all SNR ranging from very low to high without affecting the BER. Increasing the word length will retard the decoder data rate and increase the area needed for the decoder. Therefore it is solely dependent upon the application where the trade off amongst the BER, area, and data rate is paramount.

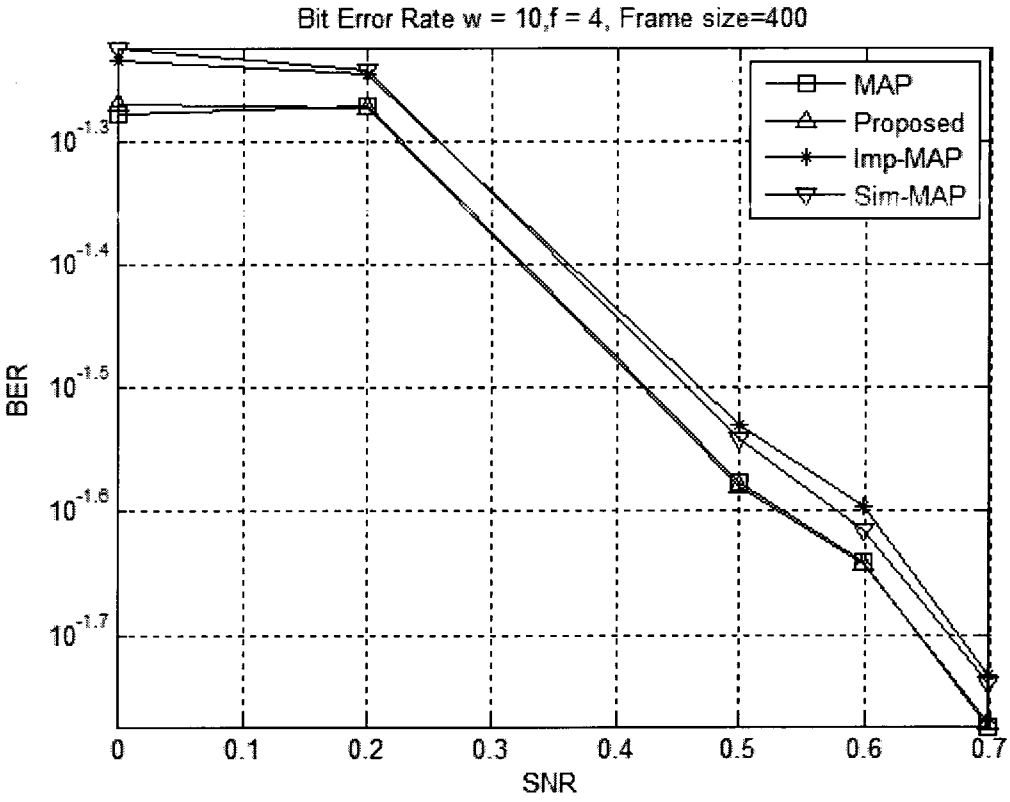

Figure 3. 13 Bit Error Rate for all algorithm with low SNR

The Figure 3.13 shows the simulation results for MAP algorithm with floating point and for other algorithm with fixed point representation. It can be seen from the simulation, that the proposed algorithm has the same BER as that of MAP algorithm

even in the presence of quantization errors since the errors induced by the algorithm are minimized to its thousandth position, which can be represented by the fixed point representation chosen for this simulation without incurring quantization errors. The SNR range was purposely selected in order to reduce the simulation time needed for all algorithms, which was almost 5 min for one frame transmission, meanwhile to show clearly the deviation from the original algorithm.

For the following simulation, only the proposed algorithm and the improved MAX-LOG-MAP were considered.

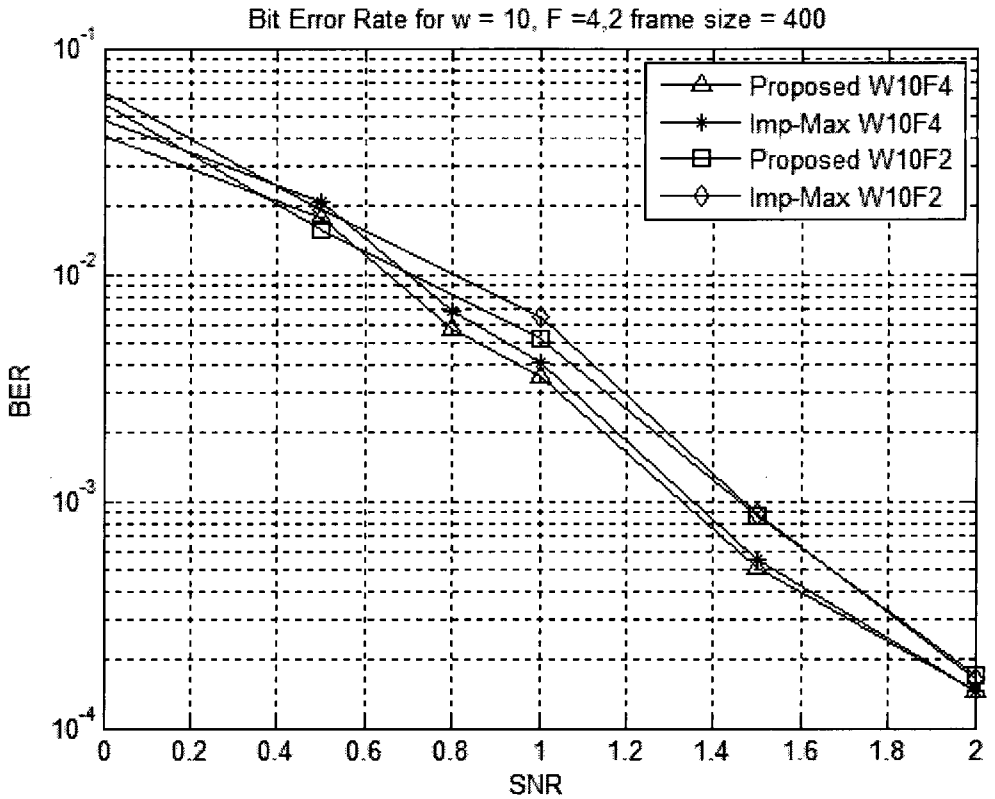

Figure3. 14 Simulation results for Two different fixed point representations

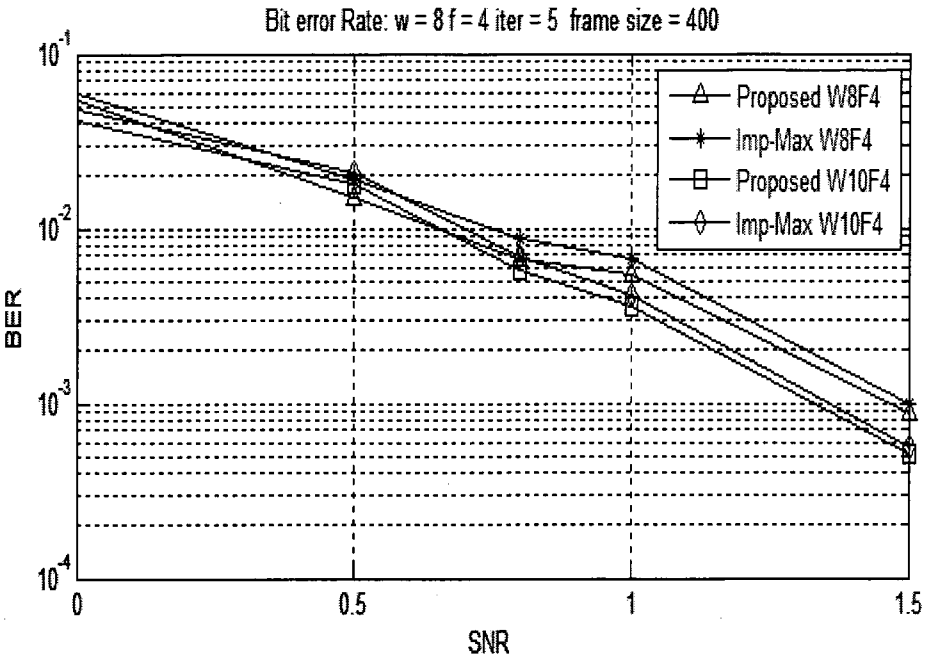

Figure 3. 15 Comparison of fixed point of (10,4) and (8,4) representation

From Figure 3.14 and 3.15, it can be seen that the fixed point representation of (10,4) has better BER performance in all SNR compared to other representation.

3.3 Observations From the Simulation

The simulation results from floating point and fixed point confirm the validity of the proposed algorithm for low SNR applications. In fact, all algorithms exhibit same BER performance at high SNR since the errors introduced by the MAX* operator can be ignored with the high reliability of the data bits. At low SNR applications, increasing the frame size does not improve the performance of the decoder for other algorithm; this is as opposed to the fact that increasing the frame size will improve the BER performance.

The number of iterations to get the BER from other algorithms is less than that of these algorithms. This is due to the errors from the MAX* operation which uses different algorithm. This, in turns will reduce the latency of the decoder and the power consumption as well.

The fixed point representation of (10,4), which has a range of -32 to 31.9375, has better BER performance in all SNR. This representation is chosen to implement the turbo decoder in the hardware.

CHAPTER 4
HARDWARE IMPLEMENTATION AND VALIDATION

The turbo decoder with MAP algorithm using proposed MAX* operation is implemented using VHDL and the modular design has been shown in this chapter. The data flow through individual modules is also described in top-down fashion, and an overview of the entire HDL implementation is produced. To optimize the data rate of the turbo decoder using hybrid algorithms, with slight degradation in BER, is also presented. The system is analyzed and synthesized using Altera Quartus and the test bench was simulated using ModelSim. It is assumed that the received systematic and parity data are available before the decoding process is initiated, and the output from the decoder is stored to compare with the original data to calculate the errors in the decoding process. The performance of the decoder is also presented in this Chapter.

4.1 Hardware Implementation of Turbo Decoder

For the hardware implementation, a 16-state decoder is chosen to incorporate with the 4G applications. The following are the parameters chosen for the turbo decoder:

- Generator Polynomial : [1 0 0 0 1; 1 1 1 1 1].
- Random Interleaver
- Frame size 1024, including the tail bits.
- 5 iterations.

The four tail bits are generated by the encoder 1 at the end of the frame, which has only 1020 bits, and these four bits are used to bring the encoder 1 to a known state; state 0. The same four bits are fed to the second encoder and the final state of the

second encoder is left open, i.e., any state is possible. The generated bits are also transmitted with the parity bits generated by both encoders.

The targeted device for hardware implementation of the turbo decoder is Altera Stratix II : EP2S180F150814. Table 4.1 highlights the main aspects of the selected FPGA.

Table 4.1: Stratix II : EP2S180F150814 features.

Feature	Value
Combinational ALUTs	143,520
Block / Distributed RAM	9,383,040
DSP block 9 bit	768
Maximum Clock Frequency	550 MHz
Gate Technology	90 nm
Core Voltage	1.2 V

4.1.1 16-State Turbo Decoder on FPGA

The block diagram of the HDL representation of a turbo decoder on FPGA is presented in Figure 4.1. The language used for the FPGA implementation is VHDL, and the internal modules are described in the following sections. At the end of this Chapter, the hardware simulation is compared with the MatLab simulation and area and data rate are compared with [20].

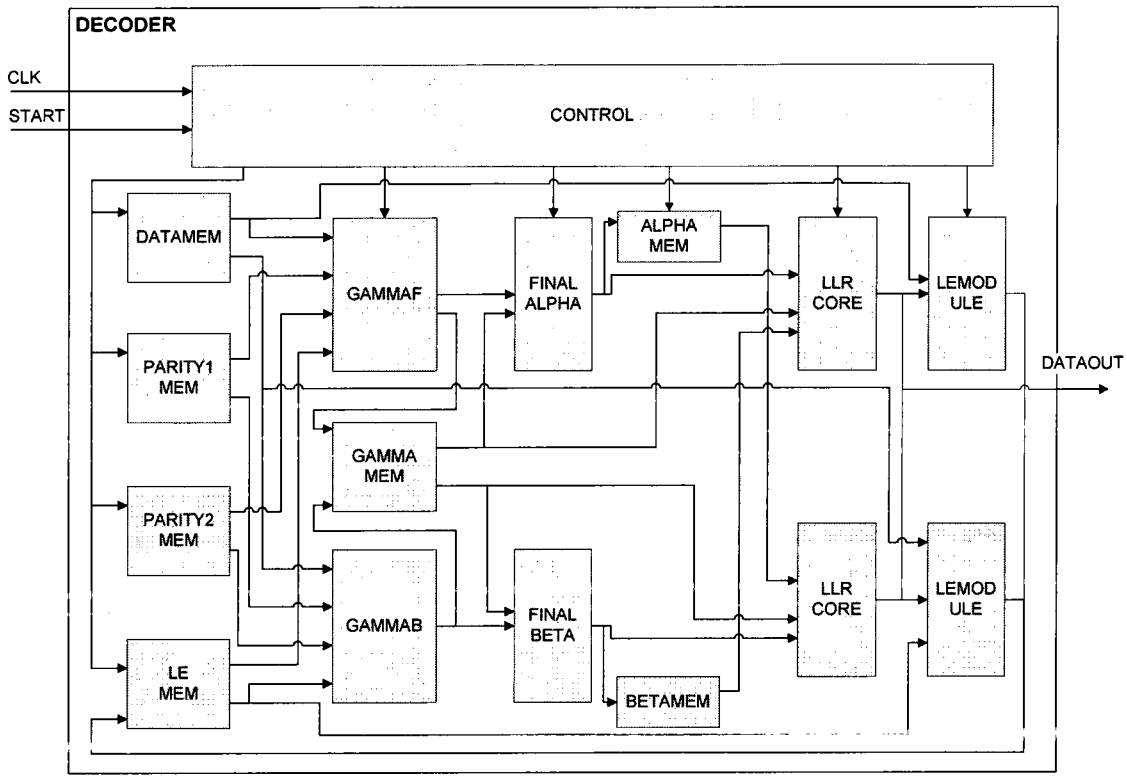

Figure 4.1: HDL blocks for the Turbo decoder.

The DECODER has only two inputs which are single bit and one output of ten bits. The sign of the DATAOUT is used to decode the data. The CONTROL module is the brain of the decoder as it controls all the modules, including the memory modules and temporary registers. Table 4.2 briefly describes each of the module, and Table 4.3 shows the memory modules and its sizes.

As mentioned earlier, the data and parity bits are assumed to be available, and the LEMEM is initialized to zero before the start of decoding process. GAMMAF and GAMMAB read the data, parity, and the extrinsic values from the beginning and end of the frame respectively and calculate the branch matrix, and during this clock period, the initialization values for the alpha and beta are saved in the ALPHAMEM and BETAMEM respectively, and also these values are made available for FINALALPHA and FINALBETA through a temporary register, to calculate the next alpha and beta values.

In the next clock cycle, outputs from the gamma modules and the temporary registers are fed to the FINALALPHA and FINALBETA to calculate the forward and backward matrices, which are loaded back to the temporary registers for the next data. The values in the temporary registers are stored in the ALPHAMEM and BETAMEM during the next clock cycle while the gamma values are computed. This process will continue until the middle of the frame is reached. At this point the control generates the enable signal for LLRCORE and LEMODULE.

Once the middle of the frame is reached the gamma modules are disenabled and the LLRCORE for the forward direction reads the gamma and beta values and evaluate the a posteriori probability using the alpha values from the temporary register that holds the alpha values, whereas the LLRCORE for the backward direction reads the gamma and alpha values from the memory, and beta values from the temporary register. LEMODULE then reads the data value from the data memory and calculates the extrinsic values.

During the next clock cycle, alpha and beta are calculated and the extrinsic values are stored in the LEMEM for the next iteration. Once the end of the frame is reached the next decoder starts decoding as explained above, but data and extrinsic values are read using the interleaver addresses, which are hard wired constants within the control unit.

When the iteration reaches its predefined value and the frame position is at the middle, the decoding process starts with the outputs from both LLRCOREs while the LEMODULE is deactivated. Since we have two LLRCOREs, the decoder decodes two data per one clock cycle.

Table 4.2: Memory and its usage

MEMORY	SIZE(bits)	USAGE
DATAMEM	1024X10	Holds the received data.
PARITY1MEM	1024X10	Holds parity from encoder1.
PARITY2MEM	1024X10	Holds parity from encoder2.
LEMEM	1024X10	Saves the extrinsic values.
ALPHAMEM	512X10	Saves alpha values.
BETAMEM	512X10	Saves the beta values.

Table 4.3 Description of the modules used in the DECODER

MODULE	FUNCTIONALITY.
GAMMAF	Calculates the branch matrix from beginning of the frame.
GAMMAB	Calculates the branch matrix from end of the frame.
FINALALPHA	Calculates the forward matrix.
FINALBETA	Calculates the backward matrix
LLRCORE	Calculates the a posteriori probability for each data.
LEMODULE	Calculates the extrinsic information for each data
CONTROL	Generates all the addresses for the memory and control signals for the modules

4.1.1.1 Trellis Diagram Of the Turbo Decoder

The trellis diagram is derived from the generator polynomial which is used to encode the data. Since the encoder has 4 memory elements, it can have 16 states and the input is binary, the trellis diagram has 32 branches. Figure 4.2 shows the transitions and the output from each state when the input is 0 or 1.

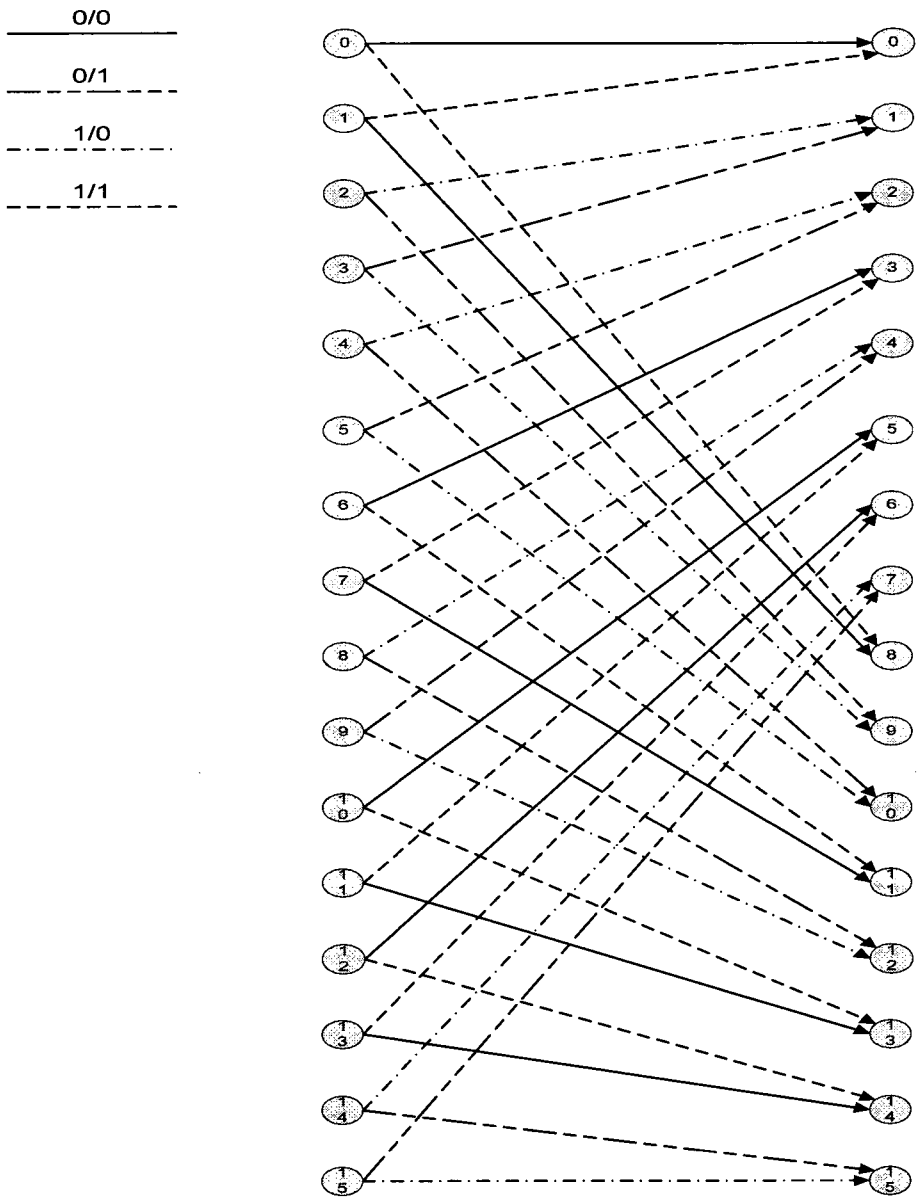

Figure 4.2 Trellis Diagram for the generator polynomial [1 0 0 0 1; 1 1 1 1 1]

4.1.1.2 Gamma Module

Gammacore is used to calculate the branch matrix associated with data, parity, and extrinsic information for a particular frame position k. Since the decoder is binary, we have only four possible transitions such as 0 to 0, 0 to 1, 1 to 0, and 1 to 1 and both 0 to 0 and 0 to 1 need three input adder and the other two transitions need four input adder. Though, the name implies adder, the operation is mixed with adder and subtraction, as a result, all the inputs are two's complemented in order to use the adder to subtract. The output GAMMA[K] is 4x10 bus which is gamma00, gamma01, gamma10, and gamma11 respectively. The equations implemented in this module are:

$$\gamma_0(k) = -datain(k) - parityin(k) - LE(k).$$

$$\gamma_1(k) = -datain(k) + parityin(k) - LE(k).$$

$$\gamma_2(k) = datain(k) - parityin(k) + LEin(k) - LE(k).$$

$$\gamma_3(k) = datain(k) + parityin(k) + LEin(k) - LE(k).$$

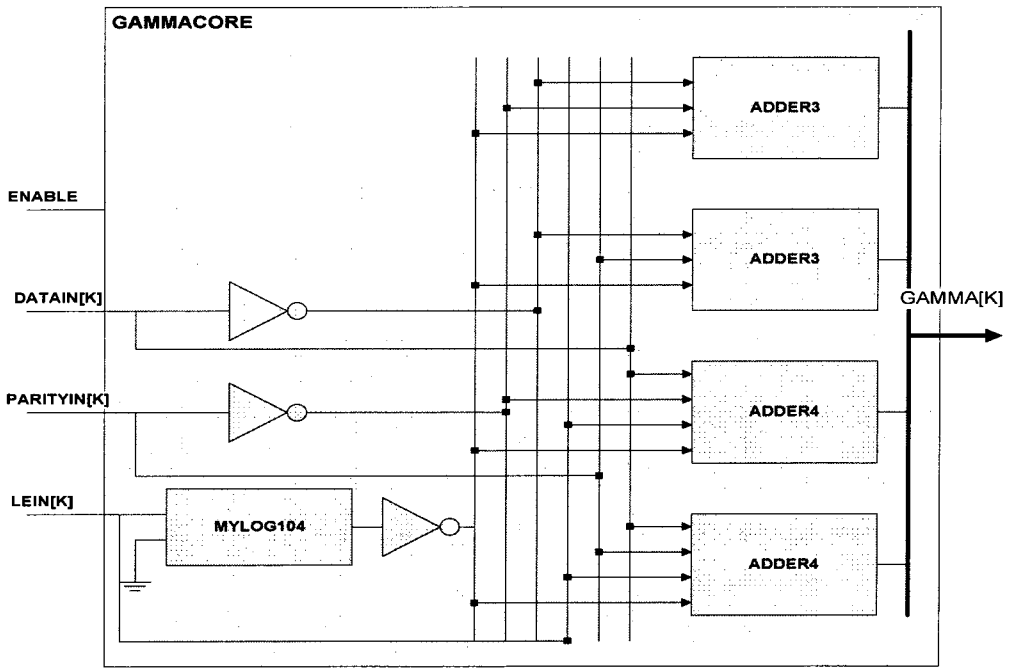

Figure 4.3: Gamma module used as GAMMAF and GAMMAB in the DECODER.

Table 4.4 Signal description for GAMMACORE

SIGNAL	Size	Description
ENABLE	1	Input from the control, when enable is high, the ouput is read.
DATAIN	10	Input from the data memory
PARITYIN	10	Input from the parity memory
LEIN	10	Input from the LE memory
GAMMA	4X10	Output to alpha and beta modules.

The GAMMACORE in Figure 4.3 is instantiated as GAMMAF and GAMMAB in the turbo decoder.

4.1.1.3 FinalAlpha Module

The Alpha module is used to calculate the 16 forward matrices from the branch matrices and the previous forward matrices. From Figure 4.4, all 16 forward matrices are derived and implemented in the alpha core. In the fixed point implementation, we need to normalize the alpha values to avoid the overflow. The normalization factor does not affect the BER performance of the decoder, but it affects the throughput of the decoder since it has the longest critical path delay. But it is indispensable in fixed point implementation.

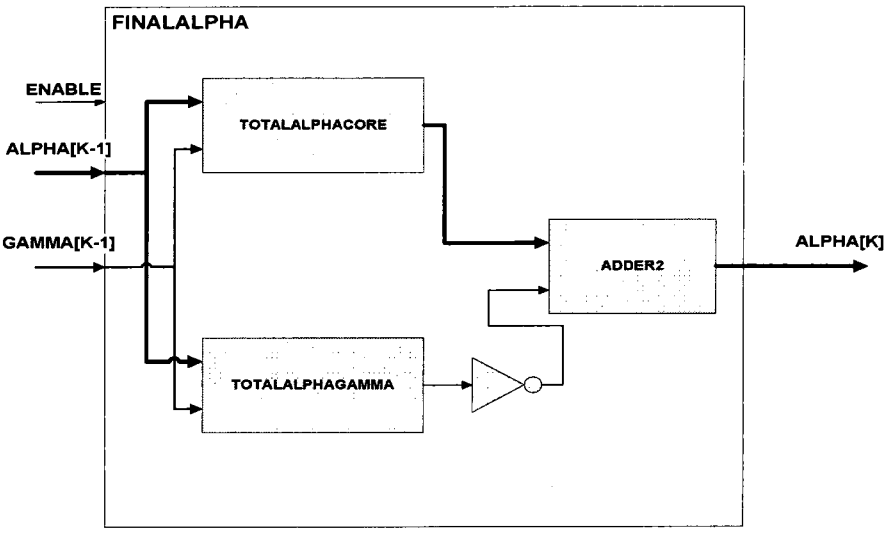

Figure 4.4: Final alpha for the Turbo Decoder

The alpha values and gamma values from the current data are used to calculate the alpha values for the next data. The output of TOTALALPHAGAMMA is subtracted from the output of the TOTALALPHACORE, in order to normalize the alpha values.

Table 4.5: Signal description for Alpha module

Signal	Size	Description
Enable	1	Synchronized with the clock and activate the FINALALPHA.
Alpha[k-1]	16X10	Forward matrices for the current data.
Gamma[k-1]	4X10	Branch matrices for the current data.
Alpha[k]	16X10	Forward matrices for the next data.

4.1.1.3.1 TotalAlphaCore Of FinalAlpha

The following 16 equations are implemented in the TOTALALPHACORE.

$$\propto_0 (k) = \gamma_0(k-1)\alpha_0(k-1) + \gamma_3(k-1)\alpha_1(k-1)$$

$$\propto_1 (k) = \gamma_2(k-1)\alpha_2(k-1) + \gamma_1(k-1)\alpha_3(k-1)$$

$$\propto_2 (k) = \gamma_1(k-1)\alpha_5(k-1) + \gamma_2(k-1)\alpha_4(k-1)$$

$$\propto_3 (k) = \gamma_0(k-1)\alpha_6(k-1) + \gamma_3(k-1)\alpha_7(k-1)$$

$$\propto_4 (k) = \gamma_1(k-1)\alpha_9(k-1) + \gamma_2(k-1)\alpha_8(k-1)$$

$$\propto_5 (k) = \gamma_0(k-1)\alpha_{10}(k-1) + \gamma_3(k-1)\alpha_{11}(k-1)$$

$$\propto_6 (k) = \gamma_0(k-1)\alpha_{12}(k-1) + \gamma_3(k-1)\alpha_{13}(k-1)$$

$$\propto_7 (k) = \gamma_1(k-1)\alpha_{15}(k-1) + \gamma_2(k-1)\alpha_{14}(k-1)$$

$$\propto_8 (k) = \gamma_0(k-1)\alpha_1(k-1) + \gamma_3(k-1)\alpha_0(k-1)$$

$$\propto_9 (k) = \gamma_1(k-1)\alpha_2(k-1) + \gamma_2(k-1)\alpha_3(k-1)$$

$$\propto_{10} (k) = \gamma_1(k-1)\alpha_4(k-1) + \gamma_2(k-1)\alpha_5(k-1)$$

$$\propto_{11} (k) = \gamma_0(k-1)\alpha_7(k-1) + \gamma_3(k-1)\alpha_6(k-1)$$

$$\propto_{12} (k) = \gamma_1(k-1)\alpha_8(k-1) + \gamma_2(k-1)\alpha_9(k-1)$$

$$\propto_{13} (k) = \gamma_0(k-1)\alpha_{11}(k-1) + \gamma_3(k-1)\alpha_{10}(k-1)$$

$$\propto_{14} (k) = \gamma_0(k-1)\alpha_{13}(k-1) + \gamma_3(k-1)\alpha_{12}(k-1)$$

$$\propto_{15} (k) = \gamma_1(k-1)\alpha_{14}(k-1) + \gamma_2(k-1)\alpha_{15}(k-1)$$

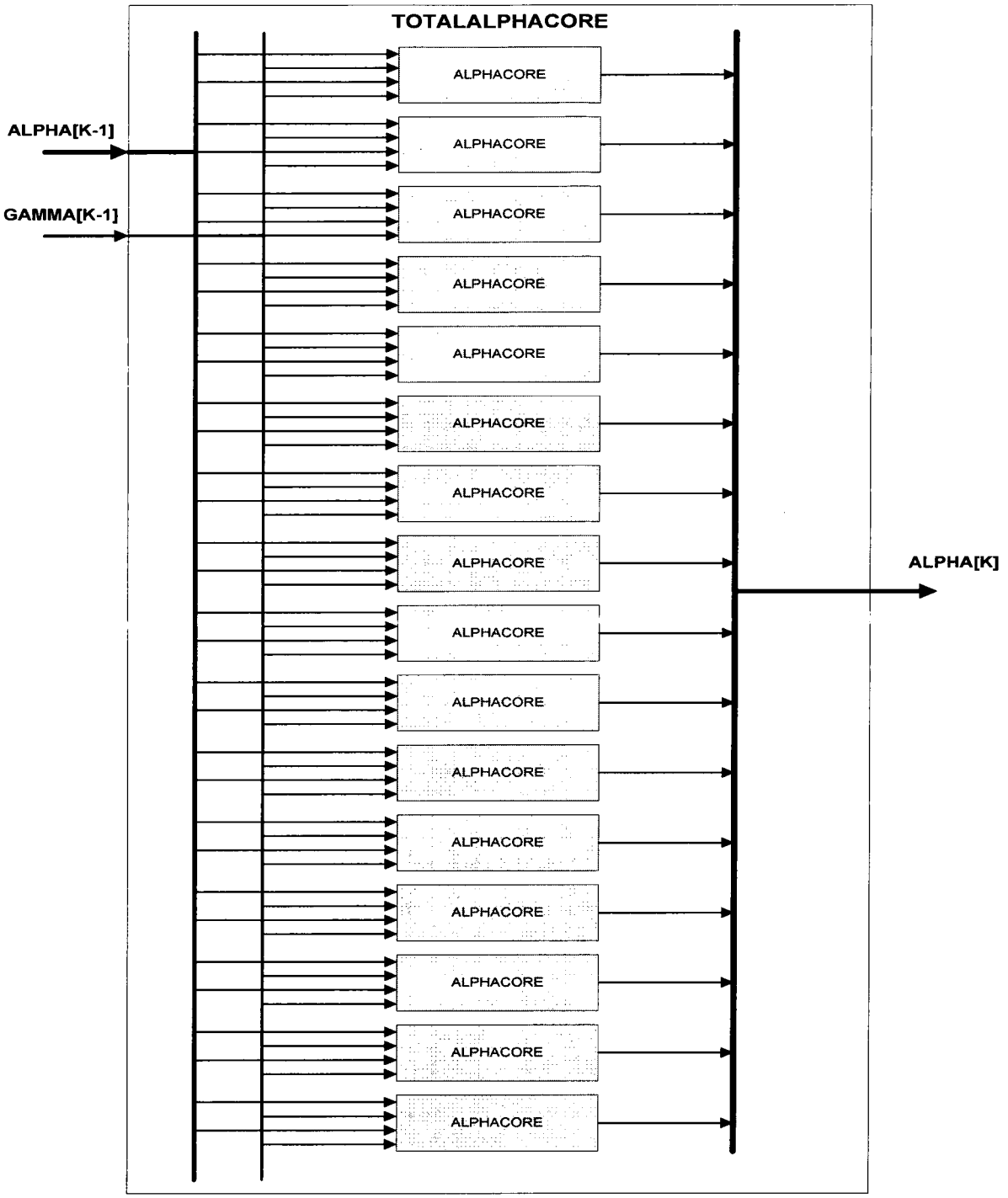

Figure 4.5: Total Alpha module without normalization.

4.1.1.3.1.1 AlphaCore Of TotalAlphaCore module

This is the basic block used in the TOTALALPHACORE. This block instantiates the MYLOG104. It just adds the alpha values with the associated gamma values and these values are captured by the MYLOG104 module to give the final output. The MYLOG104 module is described at the end of this Chapter. Figure 4.6 shows the block diagram of the ALPHACORE. If the decoder is intended to modify for other algorithm we need to simply change the MYLOG104 with the new module that is realized with the particular algorithm.

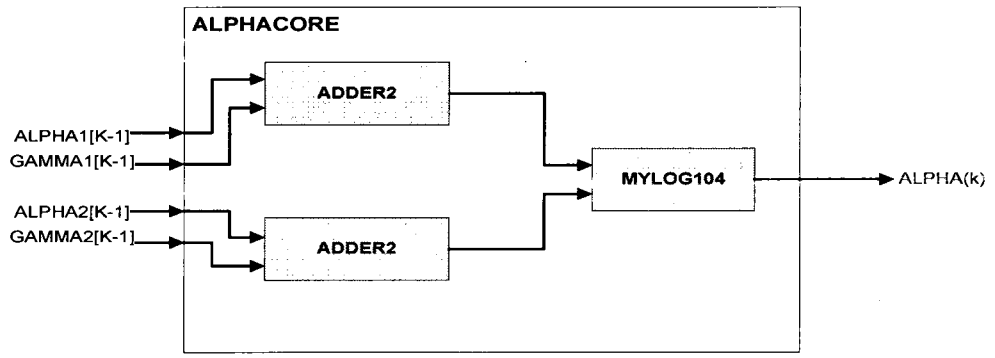

Figure 4.6 : Basic Block for Alpha module

4.1.1.3.2 TotalAlphaGamma of FinalAlpha

From Figure 4.2, we see 32 branches for the alpha and beta calculations which contribute the calculation of normalization values for alpha and beta. Therefore the normalization function can be written as:

$$TOTALGAMMAALPHA(K)$$
$$= LOG(\gamma_{K-1}(0)\alpha_{K-1}(0) + \gamma_{K-1}(3)\alpha_{K-1}(0) + \gamma_{K-1}(0)\alpha_{K-1}(1) + \gamma_{K-1}(3)\alpha_{K-1}(1) + \gamma_{K-1}(2)\alpha_{K-1}(2) + \gamma_{K-1}(1)\alpha_{K-1}(2) + \gamma_{K-1}(2)\alpha_{K-1}(3) + \gamma_{K-1}(1)\alpha_{K-1}(3) + \gamma_{K-1}(2)\alpha_{K-1}(4) + \gamma_{K-1}(1)\alpha_{K-1}(4) + \gamma_{K-1}(2)\alpha_{K-1}(5) + \gamma_{K-1}(1)\alpha_{K-1}(5) + \gamma_{K-1}(0)\alpha_{K-1}(6) + \gamma_{K-1}(3)\alpha_{K-1}(6) + \gamma_{K-1}(0)\alpha_{K-1}(7) + \gamma_{K-1}(3)\alpha_{K-1}(7) + \gamma_{K-1}(2)\alpha_{K-1}(8) + \gamma_{K-1}(1)\alpha_{K-1}(8) + \gamma_{K-1}(2)\alpha_{K-1}(9) + \gamma_{K-1}(1)\alpha_{K-1}(9) + \gamma_{K-1}(0)\alpha_{K-1}(10) + \gamma_{K-1}(3)\alpha_{K-1}(10) + \gamma_{K-1}(0)\alpha_{K-1}(11) + \gamma_{K-1}(3)\alpha_{K-1} + \gamma_{K-1}(0)\alpha_{K-1}(12) + \gamma_{K-1}(3)\alpha_{K-1}(12) + \gamma_{K-1}(0)\alpha_{K-1}(13) + \gamma_{K-1}(3)\alpha_{K-1} + \gamma_{K-1}(2)\alpha_{K-1}(14) + \gamma_{K-1}(1)\alpha_{K-1}(14) + \gamma_{K-1}(2)\alpha_{K-1}(15) + \gamma_{K-1}(1)\alpha_{K-1}(15))$$

For simplicity, we ignore the time index from the equation.

$$= \log(\gamma_0(\alpha_0 + \alpha_1 + \alpha_6 + \alpha_7 + \alpha_{10} + \alpha_{11} + \alpha_{12} + \alpha_{13}) + \gamma_3(\alpha_0 + \alpha_1 + \alpha_6 + \alpha_7 + \alpha_{10} + \alpha_{11} + \alpha_{12} + \alpha_{13}) + \gamma_1(\alpha_2 + \alpha_3 + \alpha_4 + \alpha_5 + \alpha_8 + \alpha_9 + \alpha_{14} + \alpha_{15}) + \gamma_2(\alpha_2 + \alpha_3 + \alpha_4 + \alpha_5 + \alpha_8 + \alpha_9 + \alpha_{14} + \alpha_{15}))$$

$$= \log((\gamma_0 + \gamma_3)(\alpha_0 + \alpha_1 + \alpha_6 + \alpha_7 + \alpha_{10} + \alpha_{11} + \alpha_{12} + \alpha_{13}) + (\gamma_1 + \gamma_2)(\alpha_2 + \alpha_3 + \alpha_4 + \alpha_5 + \alpha_8 + \alpha_9 + \alpha_{14} + \alpha_{15}))$$

Let $x = (\gamma_0 + \gamma_3)(\alpha_0 + \alpha_1 + \alpha_6 + \alpha_7 + \alpha_{10} + \alpha_{11} + \alpha_{12} + \alpha_{13})$ and

$Y = (\gamma_1 + \gamma_2)(\alpha_2 + \alpha_3 + \alpha_4 + \alpha_5 + \alpha_8 + \alpha_9 + \alpha_{14} + \alpha_{15})$

Therefore

TOTALGAMMAALPHA = LOG(x+y)

$$= LOG(e^{\bar{x}} + e^{\bar{y}})$$

Where $\bar{x} = \log((\gamma_0 + \gamma_3)(\alpha_0 + \alpha_1 + \alpha_6 + \alpha_7 + \alpha_{10} + \alpha_{11} + \alpha_{12} + \alpha_{13}))$

$$= \log((\gamma_0 + \gamma_3) + \log(\alpha_0 + \alpha_1 + \alpha_6 + \alpha_7 + \alpha_{10} + \alpha_{11} + \alpha_{12} + \alpha_{13})$$

Similarly

$$\bar{y} = \log(\gamma_1 + \gamma_2) + \log(\alpha_2 + \alpha_3 + \alpha_4 + \alpha_5 + \alpha_8 + \alpha_9 + \alpha_{14} + \alpha_{15})$$

Therefore instead of adding the alpha values with the corresponding gamma values we came up with the following architecture for the normalization module.

The figure 4.7 shows the block diagram of the TOTALALPHAGAMMA. It is noteworthy at this point to reveal the fact that this module has the longest critical path delay and affects the data rate of the decoder. To improve the data rate without affecting BER performance is also given at the end of this Chapter and the BER performance and data rate is compared.

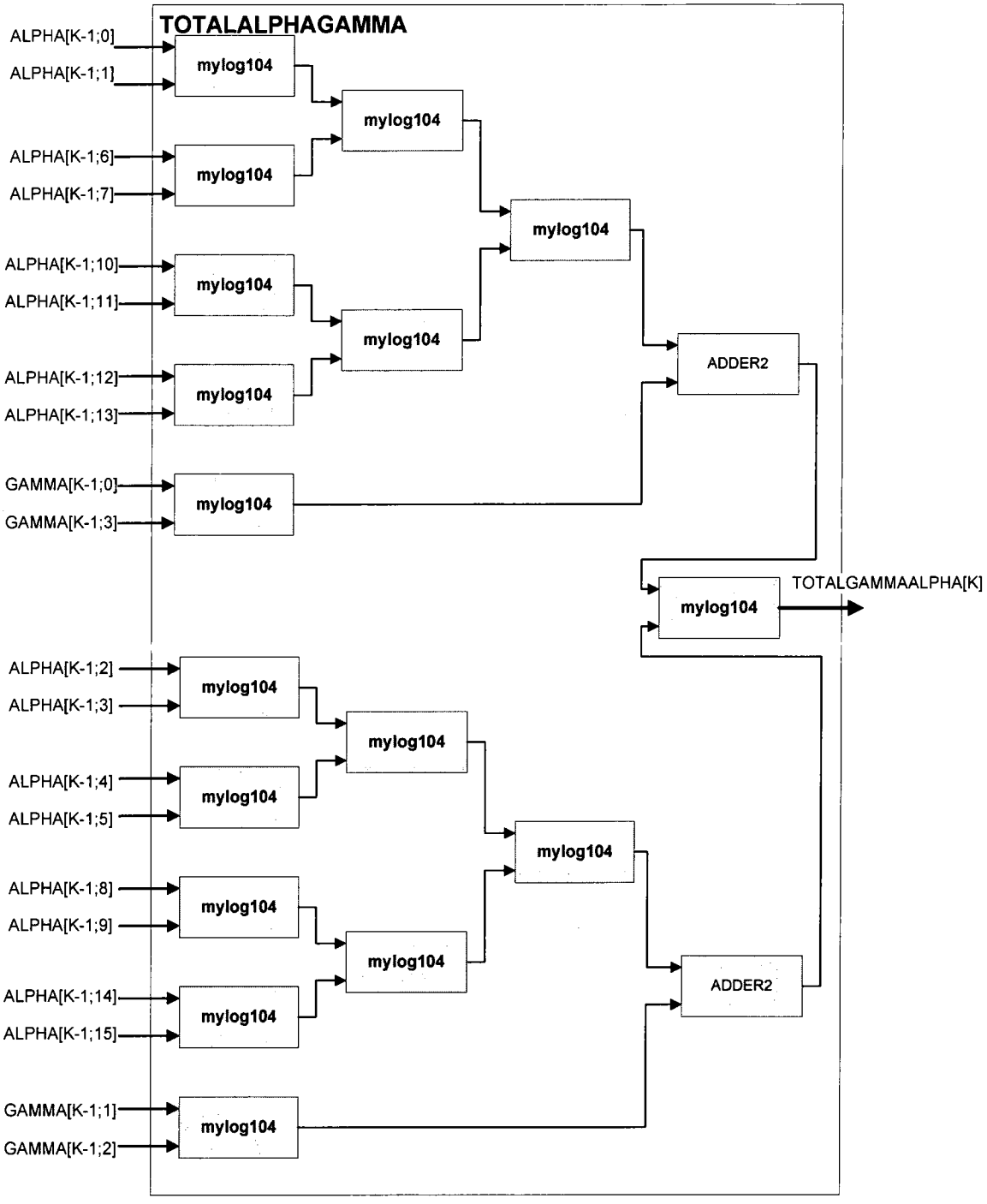

Figure 4.7 : Normalization module for Alpha.

4.1.1.4 FinalBeta Module

The FINALBETA module is similar to the FINALALPHA module in terms of signal processing, but it differs how the inputs and outputs are related. In this module the beta values of the current data are used to compute the beta values for the previous data. And also to normalize the backward matrices, the backward matrices are used instead of old-fashioned method which uses the forward matrices, so that the storing the forward matrices can be avoided. All the backward matrices are derived from the Figure 4.2 and implemented in the FINALBETA module.

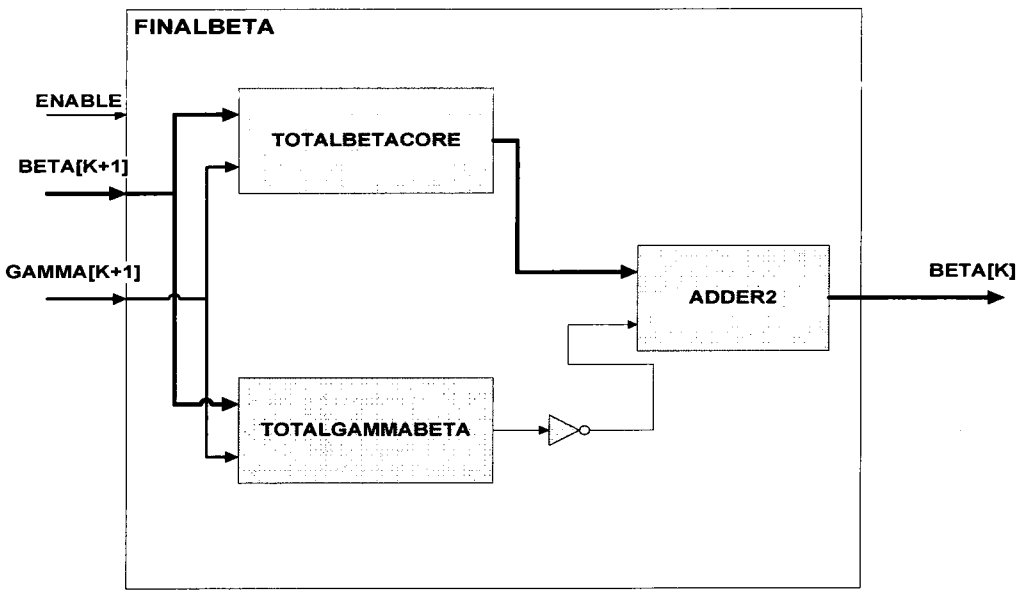

Figure 4.8: Final Beta with normalization.

The output of TOTALGAMMABETA is two's complemented so that the adder can be used to subtract it from the output of TOTALBETACORE.

Table 4.6: Signal description of FINALBETA.

SIGNAL	SIZE	DESCRIBTION
ENABLE	1	Synchronized with the clock, used to activate the module
GAMMA[K+1]	4X10	Input from the GAMMA module
BETA[K+1]	16X10	Feedback from the FINALBETA
BETA[K]	16X10	Output fed back to same module

4.1.1.4.1 TotalBetaCore of FinalAlpha

The following equations are implemented in the TOTALBETACORE.

$$\beta_0(k-1) = \gamma_0(k)\beta_0(k) + \gamma_3(k)\beta_8(k)$$

$$\beta_1(k-1) = \gamma_0(k)\beta_8(k) + \gamma_3(k)\beta_0(k)$$

$$\beta_2(k-1) = \gamma_2(k)\beta_2(k) + \gamma_1(k)\beta_3(k)$$

$$\beta_3(k-1) = \gamma_2(k)\beta_2(k) + \gamma_1(k)\beta_3(k)$$

$$\beta_4(k-1) = \gamma_0(k)\beta_0(k) + \gamma_3(k)\beta_8(k)$$

$$\beta_5(k-1) = \gamma_0(k)\beta_8(k) + \gamma_3(k)\beta_0(k)$$

$$\beta_6(k-1) = \gamma_2(k)\beta_2(k) + \gamma_1(k)\beta_3(k)$$

$$\beta_7(k-1) = \gamma_2(k)\beta_2(k) + \gamma_1(k)\beta_3(k)$$

$$\beta_8(k-1) = \gamma_0(k)\beta_0(k) + \gamma_3(k)\beta_8(k)$$

$$\beta_9(k-1) = \gamma_0(k)\beta_8(k) + \gamma_3(k)\beta_0(k)$$

$$\beta_{10}(k-1) = \gamma_2(k)\beta_2(k) + \gamma_1(k)\beta_3(k)$$

$$\beta_{11}(k-1) = \gamma_2(k)\beta_2(k) + \gamma_1(k)\beta_3(k)$$

$$\beta_{12}(k-1) = \gamma_0(k)\beta_0(k) + \gamma_3(k)\beta_8(k)$$

$$\beta_{13}(k-1) = \gamma_0(k)\beta_8(k) + \gamma_3(k)\beta_0(k)$$

$$\beta_{14}(k-1) = \gamma_2(k)\beta_2(k) + \gamma_1(k)\beta_3(k)$$

$$\beta_{15}(k-1) = \gamma_2(k)\beta_2(k) + \gamma_1(k)\beta_3(k)$$

Each of the BETACORE modules in TOTALBETACORE implements one of the above equation.

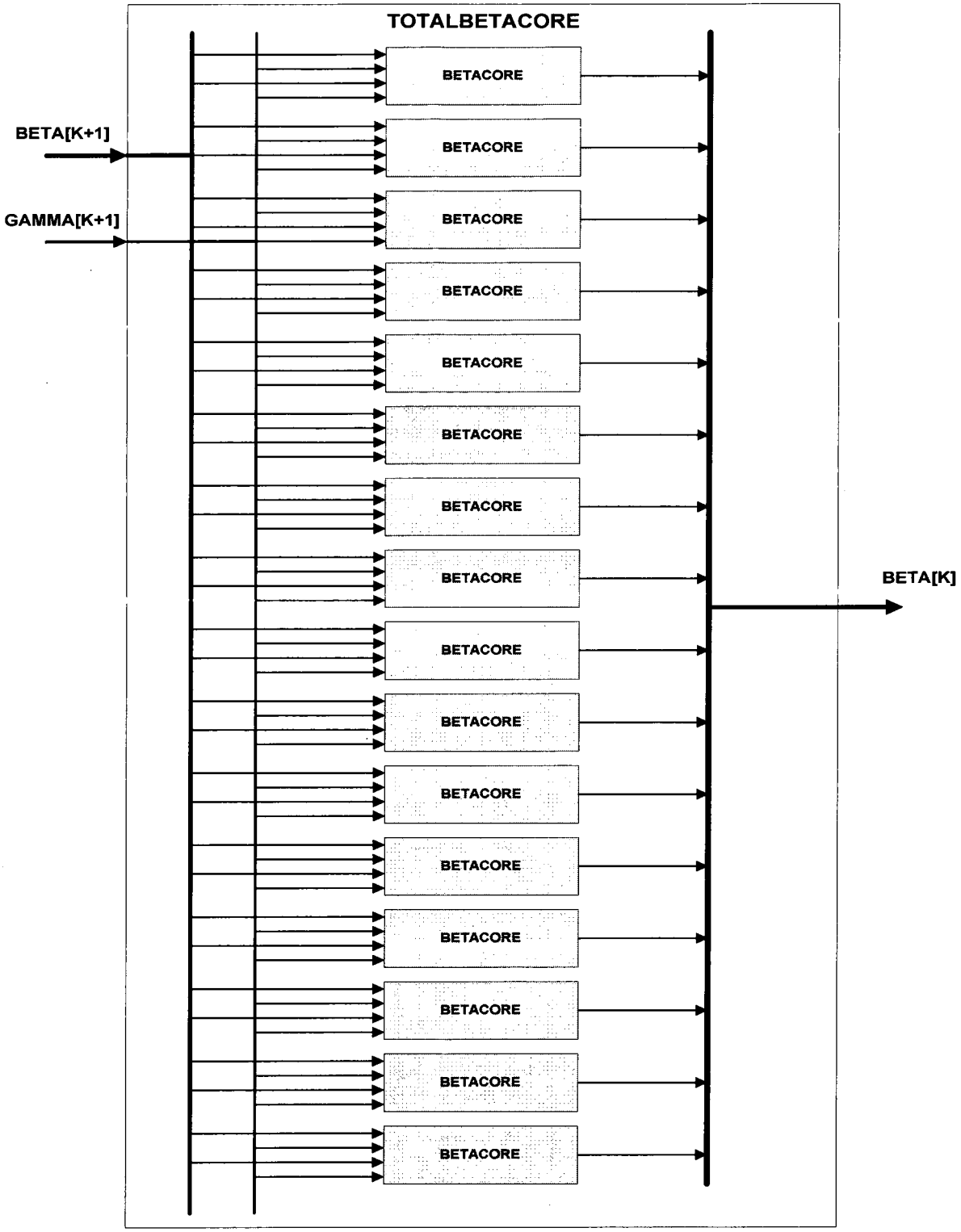

Figure 4.9: Final Beta without normalization

4.1.1.4.1.1 BetaCore of TotalBeta Module

The BETACORE computes the backward matrices for the previous data from the current data. The beta values are added with the associated gamma values and the result is fed to the MYLOG104 module to compute the final output.

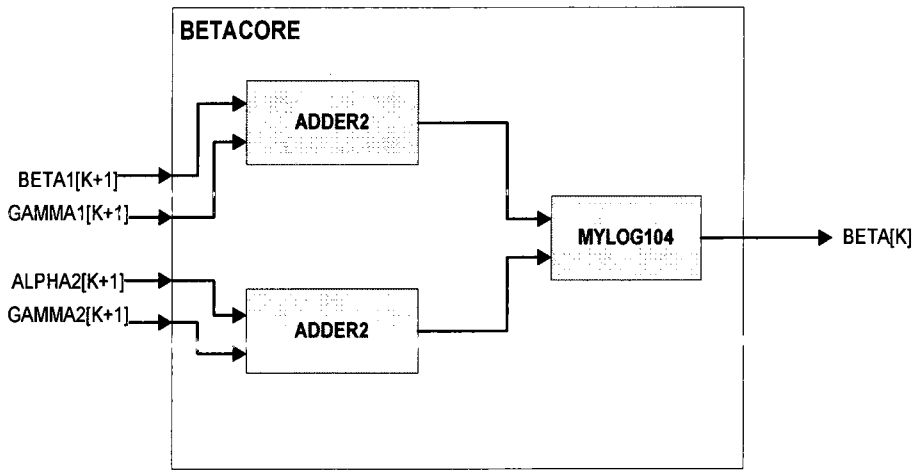

Figure 4.10: Basic block for backward matrix calculation.

4.1.1.4.2 TotalBetaGamma of FinalBeta

This module implements the following equation which is derived from the trellis diagram, in order to normalize the beta values.

$$
\begin{aligned}
&TOTALGAMMABETA(K-1)\\
&\quad = LOG(\gamma_{K-1}(0)\beta_K(0) + \gamma_{K-1}(0)\beta_K(8) + \gamma_{K-1}(0)\beta_K(3)\\
&\quad + \gamma_{K-1}(0)\beta_K(11) + \gamma_{K-1}(0)\beta_K(5) + \gamma_{K-1}(0)\beta_K(13) + \gamma_{K-1}(0)\beta_K(6)\\
&\quad + \gamma_{K-1}(0)\beta_K(14) + \gamma_{K-1}(3)\beta_K(0) + \gamma_{K-1}(3)\beta_K(8) + \gamma_{K-1}(3)\beta_K(3)\\
&\quad + \gamma_{K-1}(3)\beta_K(11) + \gamma_{K-1}(3)\beta_K(5) + \gamma_{K-1}(3)\beta_K(13) + \gamma_{K-1}(3)\beta_K(6)\\
&\quad + \gamma_{K-1}(3)\beta_K(14) + \gamma_{K-1}(1)\beta_K(1) + \gamma_{K-1}(1)\beta_K(9) + \gamma_{K-1}(1)\beta_K(10)\\
&\quad + \gamma_{K-1}(1)\beta_K(2) + \gamma_{K-1}(1)\beta_K(12) + \gamma_{K-1}(1)\beta_K(4) + \gamma_{K-1}(1)\beta_K(15)\\
&\quad + \gamma_{K-1}(1)\beta_K(7) + \gamma_{K-1}(2)\beta_K(1) + \gamma_{K-1}(2)\beta_K(9) + \gamma_{K-1}(2)\beta_K(10)\\
&\quad + \gamma_{K-1}(2)\beta_K(2) + \gamma_{K-1}(2)\beta_K(12) + \gamma_{K-1}(2)\beta_K(4) + \gamma_{K-1}(2)\beta_K(15)\\
&\quad + \gamma_{K-1}(2)\beta_K(7))
\end{aligned}
$$

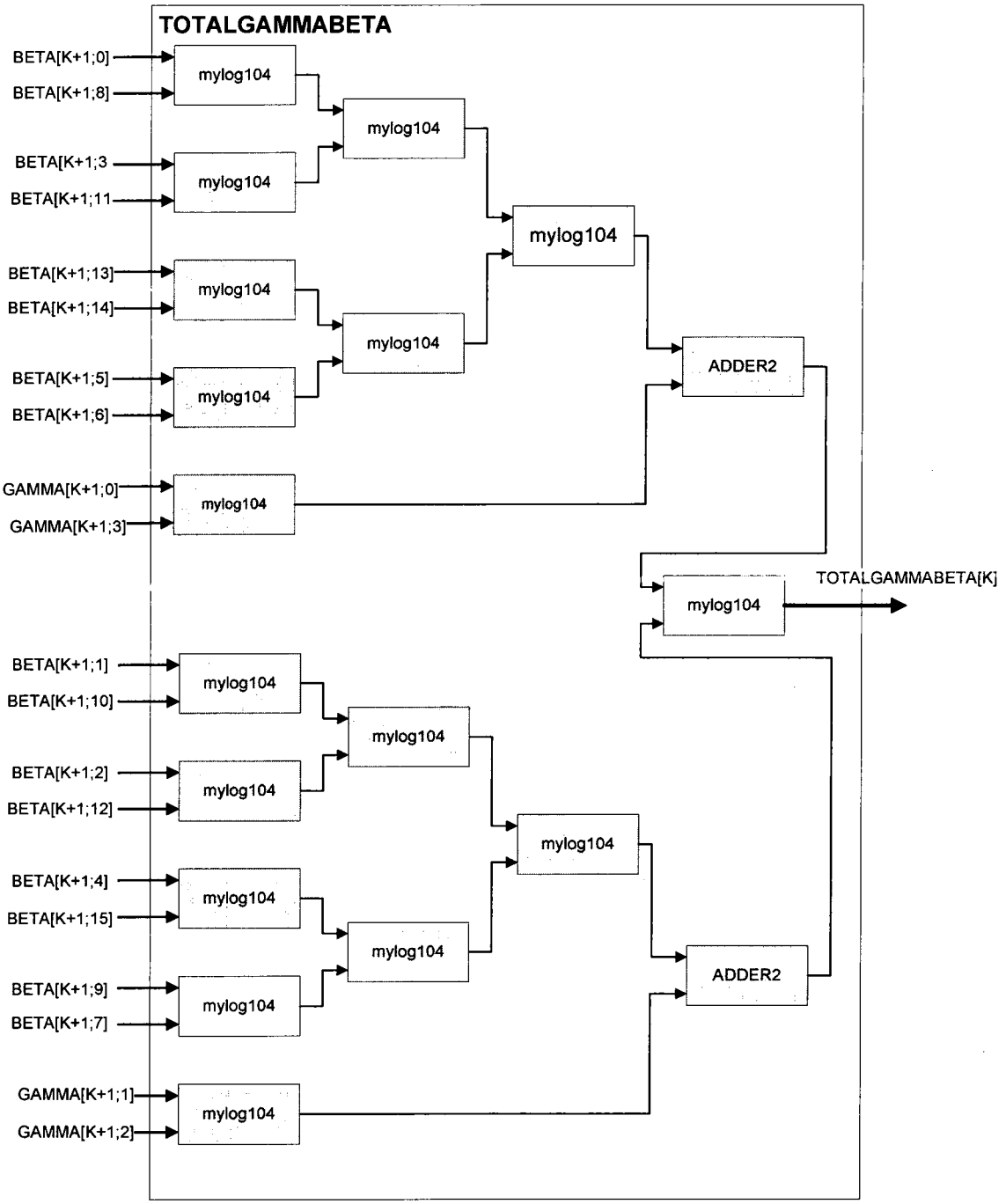

Figure 4.11: Normalization module for Beta.

The novelty of this architecture is that of the reduced number of adders needed by avoiding the addition of gamma and beta at the beginning. Instead beta and gamma values are fed to the MYLOG104 and the corresponding gamma beta values are then added. This will reduce the number of adders to 2 from 64, and it would not change the critical path delay of the module.

4.1.1.5 LLRCore of Turbo Decoder

The LLRCORE consists of two modules that one of the modules computes the a posteriori probability of input 1 and the other module computes for input 0. The implemented equations are derived from the trellis diagram shown in Figure 4.2 , by considering the forward and backward matrices that associated with gamma10 and gamma11 for the case of input 1 and gamma00 and gamma01 for the case of input 0.

The derived equation for input 1 :

$$\begin{aligned} LLRone(k) = Log[&\alpha_k(0)\gamma_k(3)\beta_k(8) + \alpha_k(1)\gamma_k(3)\beta_k(0) + \alpha_k(2)\gamma_k(2)\beta_k(1) \\ &+ \alpha_k(3)\gamma_k(2)\beta_k(9) + \alpha_k(4)\gamma_k(2)\beta_k(2) + \alpha_k(5)\gamma_k(2)\beta_k(10) \\ &+ \alpha_k(6)\gamma_k(3)\beta_k(11) + \alpha_k(7)\gamma_k(3)\beta_k(3) + \alpha_k(8)\gamma_k(2)\beta_k(4) \\ &+ \alpha_k(9)\gamma_k(2)\beta_k(12) + \alpha_k(10)\gamma_k(3)\beta_k(13) + \alpha_k(11)\gamma_k(3)\beta_k(5) \\ &+ \alpha_k(12)\gamma_k(3)\beta_k(14) + \alpha_k(13)\gamma_k(3)\beta_k(6) + \alpha_k(14)\gamma_k(2)\beta_k(7) \\ &+ \alpha_k(15)\gamma_k(2)\beta_k(15)] \end{aligned}$$

The derived equation for input 0:

$$\begin{aligned} LLRzero(k) = Log[&\alpha_k(0)\gamma_k(0)\beta_k(0) + \alpha_k(1)\gamma_k(0)\beta_k(8) + \alpha_k(2)\gamma_k(1)\beta_k(9) \\ &+ \alpha_k(3)\gamma_k(1)\beta_k(1) + \alpha_k(4)\gamma_k(1)\beta_k(10) + \alpha_k(5)\gamma_k(1)\beta_k(2) \\ &+ \alpha_k(6)\gamma_k(0)\beta_k(3) + \alpha_k(7)\gamma_k(0)\beta_k(11) + \alpha_k(8)\gamma_k(1)\beta_k(12) \\ &+ \alpha_k(9)\gamma_k(1)\beta_k(4) + \alpha_k(10)\gamma_k(0)\beta_k(5) + \alpha_k(11)\gamma_k(0)\beta_k(13) \\ &+ \alpha_k(12)\gamma_k(0)\beta_k(6) + \alpha_k(13)\gamma_k(0)\beta_k(14) + \alpha_k(14)\gamma_k(1)\beta_k(15) \\ &+ \alpha_k(15)\gamma_k(1)\beta_k(7)] \end{aligned}$$

The implementation of these equations are shown in Figure 4.12 and Figure 4.13 . Though the implementation employs the parallel architecture the critical path delay of these modules significantly reduces the throughput of the decoder.

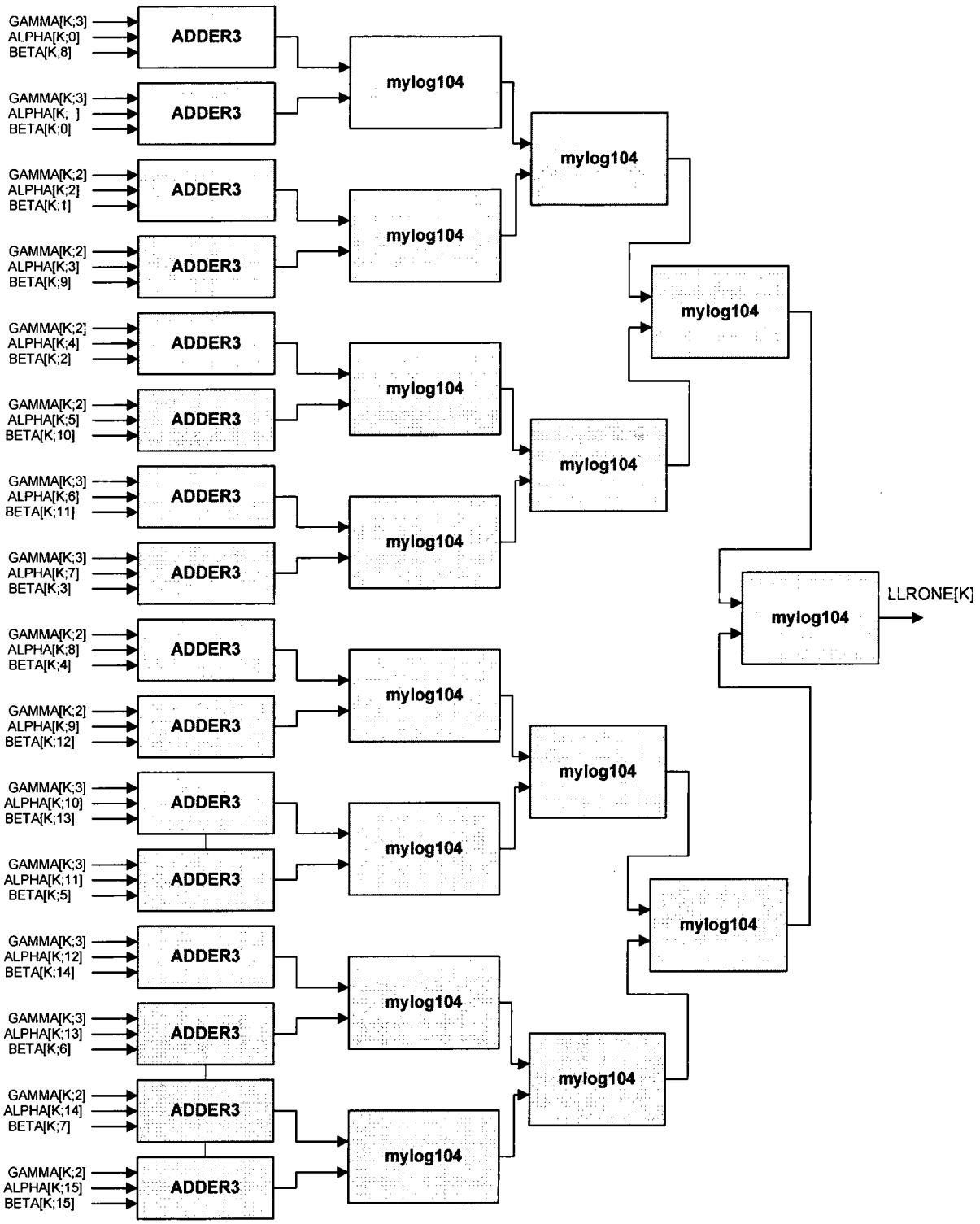

Figure 4.12: Architecture for finding LLRone.

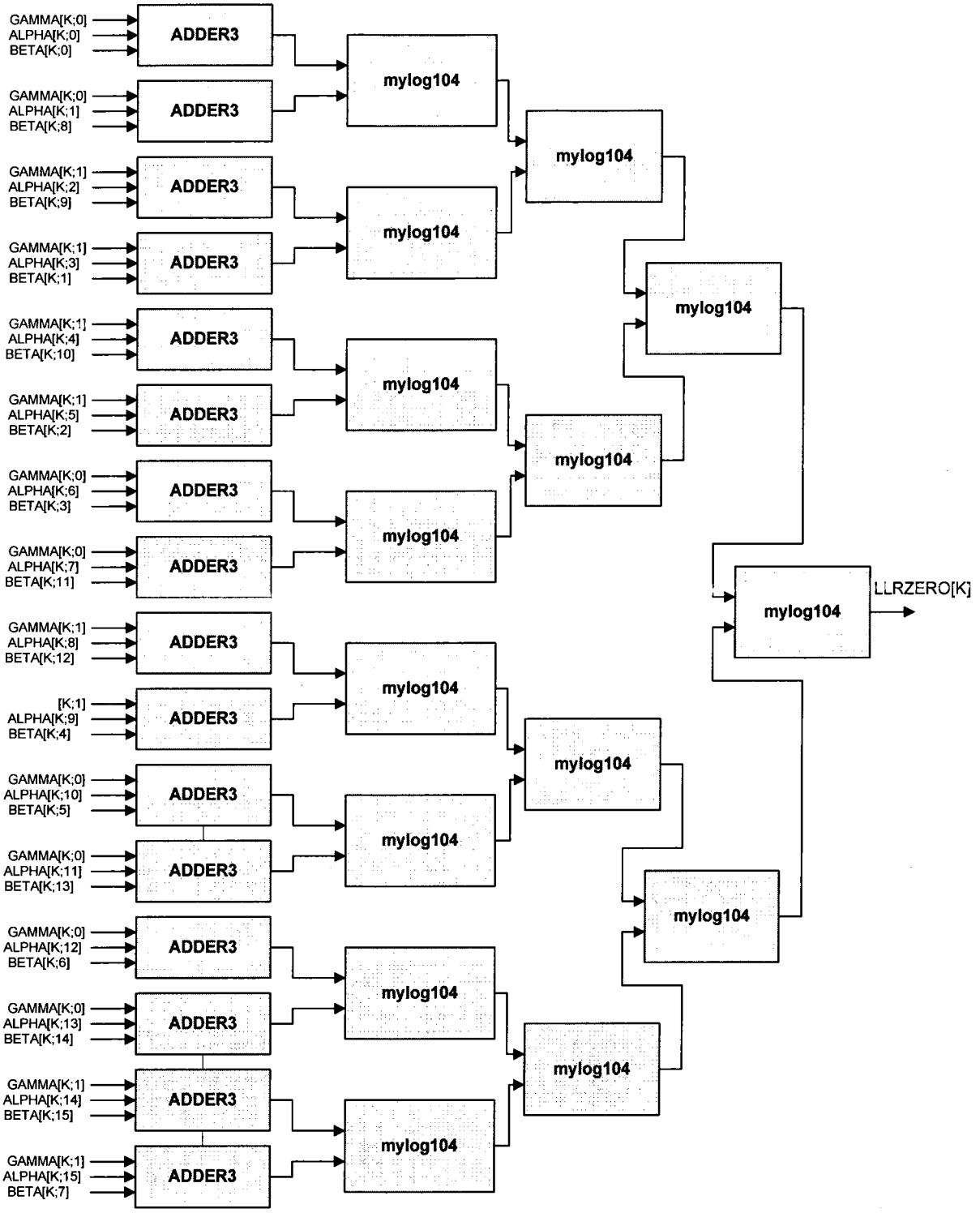

Figure 4.13: Architecture for computing LLRzero.

By combining both modules and an adder the LLRCORE is implemented.

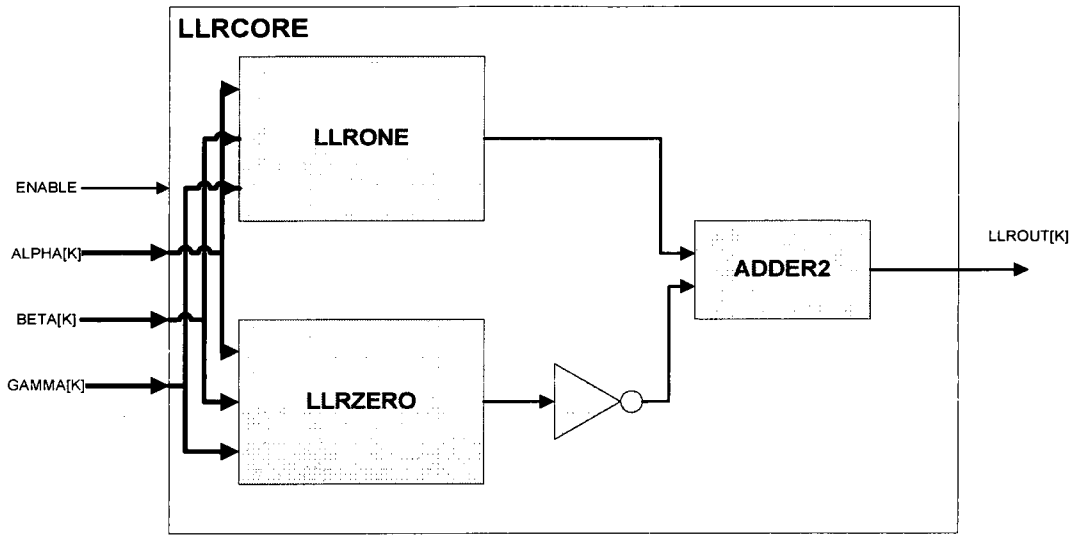

Figure 4.14: LLR module

4.1.1.6 LECore of Turbo Decoder

LECORE is implemented to find the extrinsic information that each decoder is learnt itself. Since both decoders are accessible to channel information, LECORE must suppress the channel information as well as extrinsic information learnt from the other decoder. The Figure 4.15 shows the implementation of the LECORE.

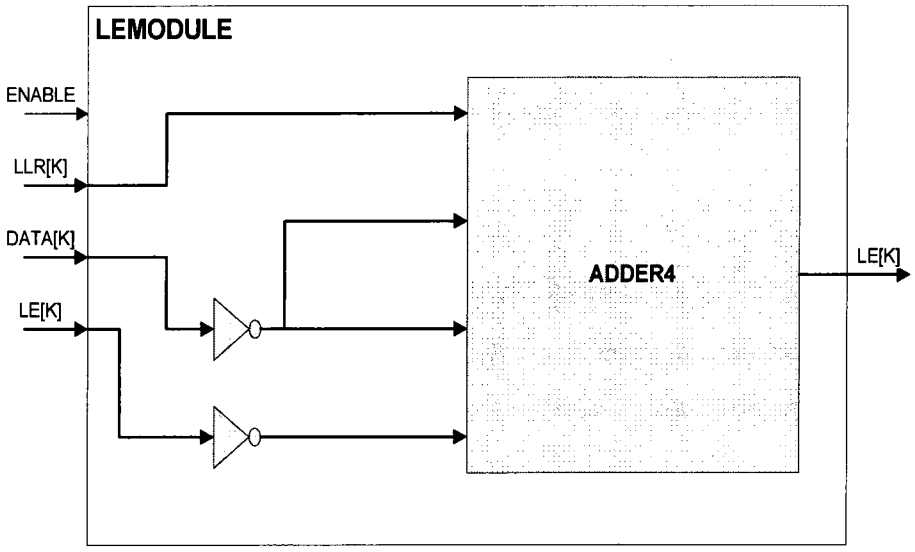

Figure 4.15: Block diagram for extrinsic computation

4.1.1.7 Control Module of Turbo Decoder

The flow chart for the control module is shown in Figure 4.16. The control module has only two inputs; clk and start. Since the control module provides the addresses of memory of data, parity, and extrinsic values and the decoder decodes the data frame from both directions, control module has two counters; one is up counting and the other is down counting. And also when decoding process is switched to second decoder, control module should be able to provide the interleaved addresses for data and extrinsic memory location. To get around the clock transition, the interleaver addresses are defined within the control signal as hard wired constants. This will make sure that during the clock transition, the data and extrinsic addresses are readily available along with the parity address.

The control module has 5 states such as IDLE, STA0, STA1, STA2, and STA3. Table describes what signal assignments are carried out by the control module in each state. Detailed information can be found in Appendix under the VHDL description for control module.

Table 4.7: State and Signal assignment.

STATE	SIGNAL ASSIGNMENT
IDLE	All memories are enabled. Alphawr, betawr, gammawr, gammaenable, Ld_Areg, and Ld_Breg are set to 1. Alphaenable, betaenable, LLRenable, LEenable, and decode are set to 0.
STA0	Memory modules for gamma, beta, and alpha are disabled with a condition[1]. Alphawr, betawr, gammawr, gammaenable, Ld_Areg, and Ld_Breg are set to 0. Alphaenable and betaenable are set to 1. Upcounting and downcounting.
STA1	Memory modules for gamma, beta, and alpha are enabled with a condition[2]. Gammawr, alphawr, betawr, gammaenable, Ld_Areg, and Ld_Breg are set to 1 with a condition[2]. Alphaenable and betaenable are set to 0. Decode is set to 1 with a condition[3].

	Leenable is set to 0 with a condition[4].
STA2	Memories for gamma, alpha, and beta are enabled. Alphaenable and betaenable are set to 1. LLRenable, LEenable, LEwr, Ld_Areg, and Ld_Breg are set to 0. Upcounting and downcounting. Decoder is switiched with a condition[5]. Iteration is decreased by 1 with a condition[6].
STA3	Memories for gamma, alpha, and beta are disabled. LLRenable, LEwr, Ld_Areg, and Ld_Breg are set to 1. LEenable is set to 0 or 1 with a condition[7].

Conditions:

1. *up counter is less than 511.*
2. *up counter is less than or equal to 511.*
3. *iteration = 0, decoder = 1, and upcounter = 512.*
4. *iteration = 0 and decoder = 1.*
5. *upcounter = 1023.*
6. *upcounter = 1023 and deocoder = 1.*
7. *iteration = 0 and decoder = 1.*

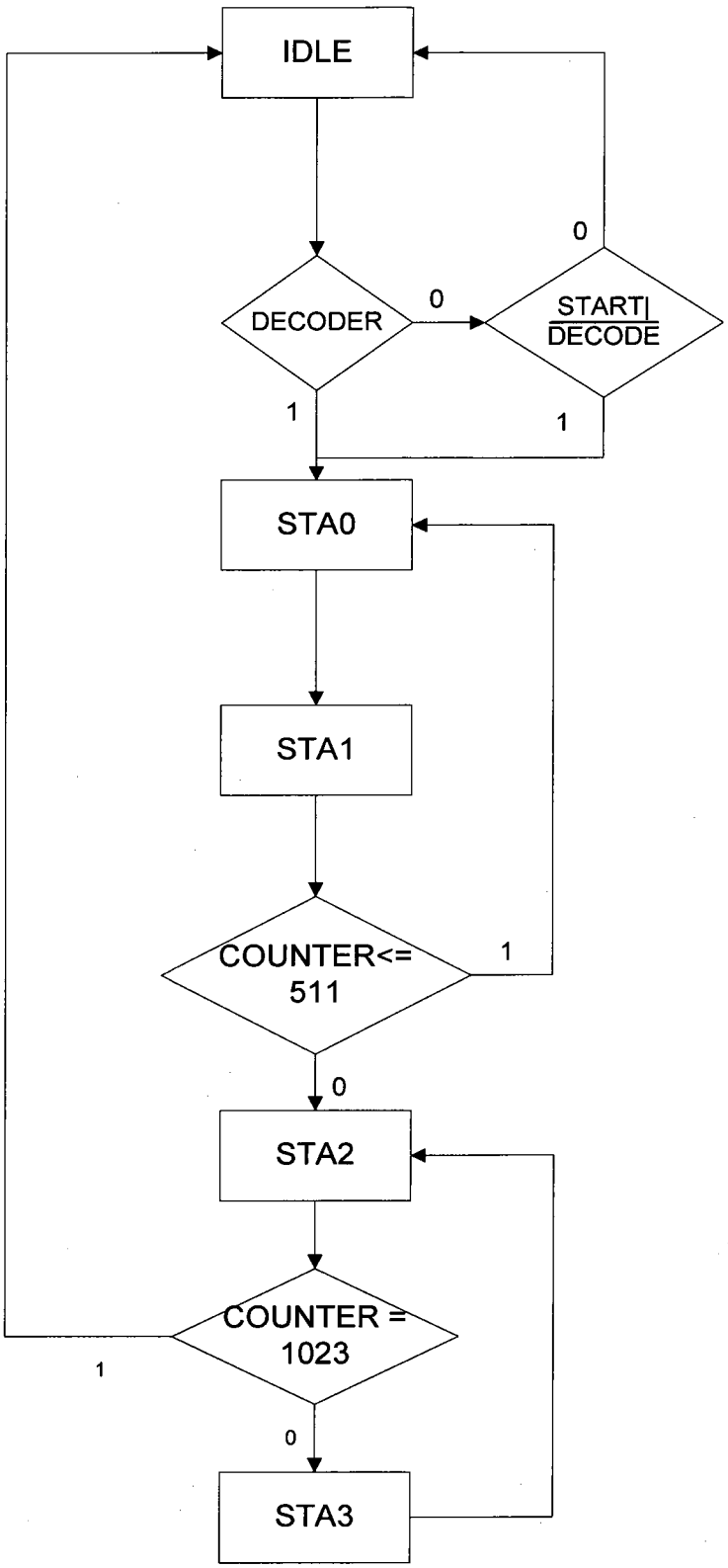

Figure 4.16 : Flow chart for the control module.

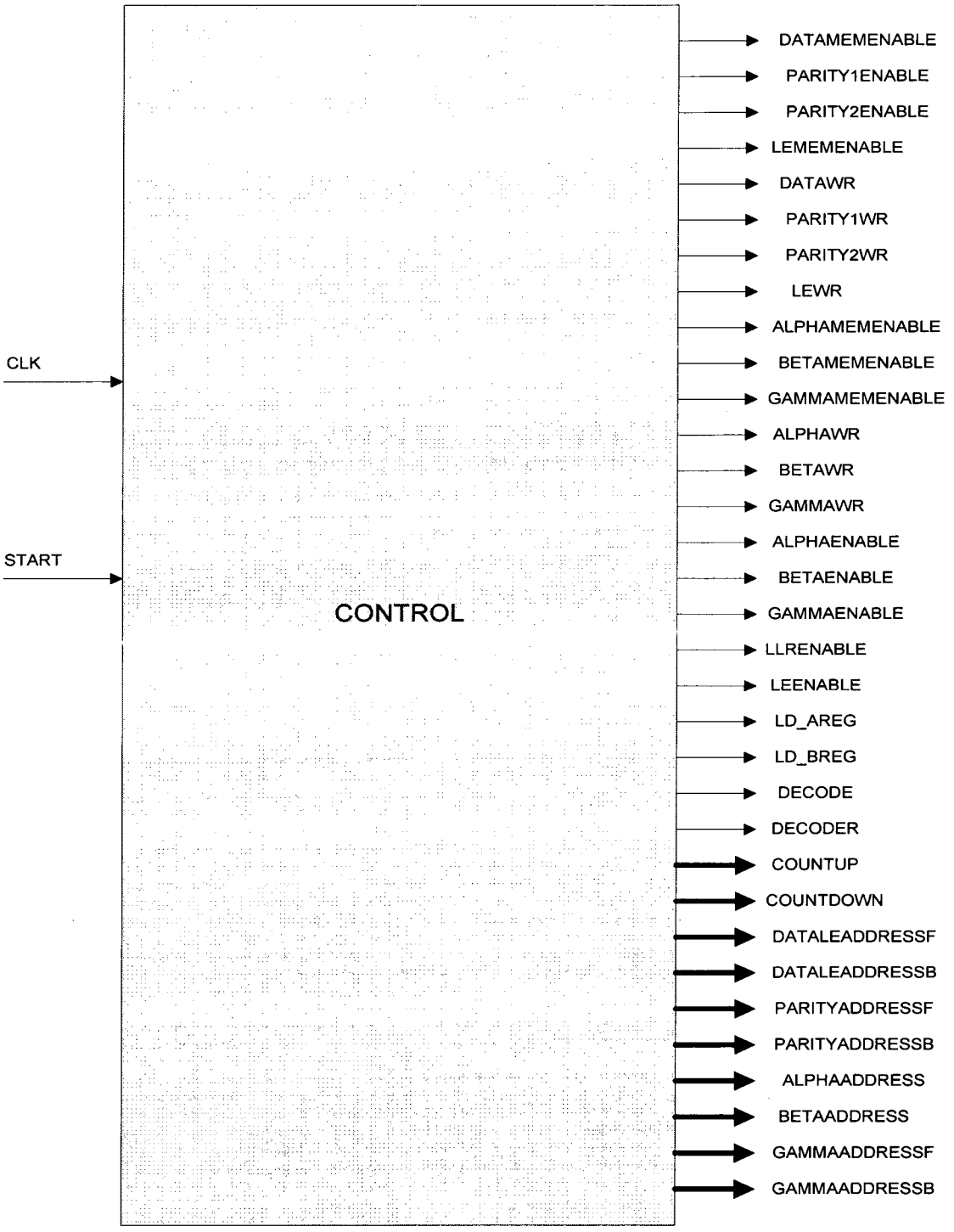

Figure 4.17: Control module

**Thick lines are 10 bits width and others are single bit.*

Apart from the state transition, the control signal has two other processes, which are activated by the upcounter and decoder signals. One of the processes handles which parity memory must be enabled based on the decoder, and the other process handles memory for data and extrinsic values to be interleaved or not, and how alpha and beta are addressed when counter is less than or equal to 511. These processes are vital in this design by serving two purposes; one is we need only one decoder, as a result the area is reduced by 50% and the second one is memory needed for storing alpha and beta are also reduced by 50%.

The novel implementation of the turbo decoder comes in the form of integration of single modules needed for each alpha, beta, gamma, and LLR modules, together within the desired module in order to increase each module throughput by means of reducing the interconnection delays, which is the bottleneck for the most of the digital design.

4.1.2 MyLog104 Module for MAX*

The MYLOG104 modules implements the new algorithm described in detail in Chapter 3. This module is made up of MAX, ABS, COMPARATOR, and a MULT units. It has two 10 bit inputs and one 10 bits output representing the integer part and fractional part with signed bit. The Figure shows the block diagram of the module used in alpha, beta, gamma, and LLR modules to calculate very preciously the log values of a function log(x+y).

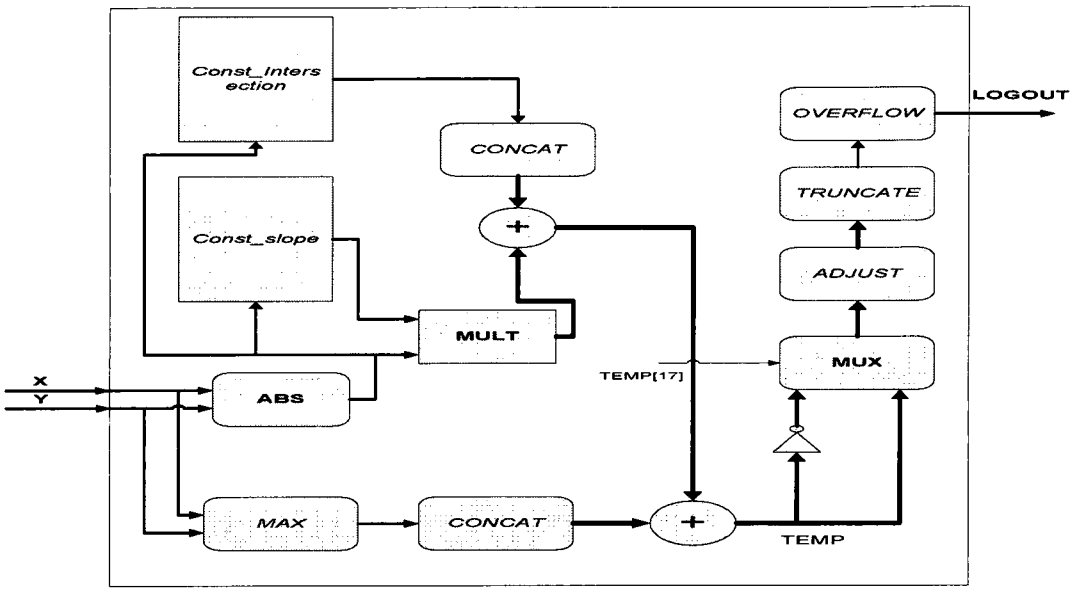

Figure 4.18: Signal flow diagram for the new algorithm.

The intersection and slope used in the correction function are represented in (8,7) since these values are always less than one, the representation does not need any bit for integer. First the absolute different of x and y is calculated and it is used as selection signal for the intersection and slope which are hardwired constants. In the meantime, the maximum value of x and y is computed. The multiplier captures the output from the ABS and the output from the slope and produces its output. Since the ABS output is (10,4) and the slope is (8,7) the resultant output of the MULT is (18,11), as a result the output from the intersection and the MAX should be concatenated with tail bits and leading bits.

The output from the adder which captures the output from the multiplier and the intersection is the correction value which needs to be added with the maximum value. The signed output of the second adder needs to be truncated in order to get the output of the module in (10,4) representation. The *ADJUST* process is used to increase the precision of the output.

4.2 Optimization of Data Rate

The BER performance of the Turbo Decoder mostly depends on what algorithm is used to compute the correction values for forward, backward, and branch matrices. Since the longest critical path introduced in alpha and beta modules relies on the normalization module, which does not have any significant impact on the accuracy of these matrices, the normalization module can be employed with MAX operation in order to increase the throughput of these modules without affecting the BER performance in any manner. And also the longest critical path of the LLR module can be reduced by employing same algorithm as well. This will slightly decrease the BER performance compared to LOG-MAP in low SNR regions. To overcome this problem, an error scaling factor is added to the MAX operation. Figure shows the MatLab simulation with original algorithm and original algorithm with MAX for normalization and LLR computation.

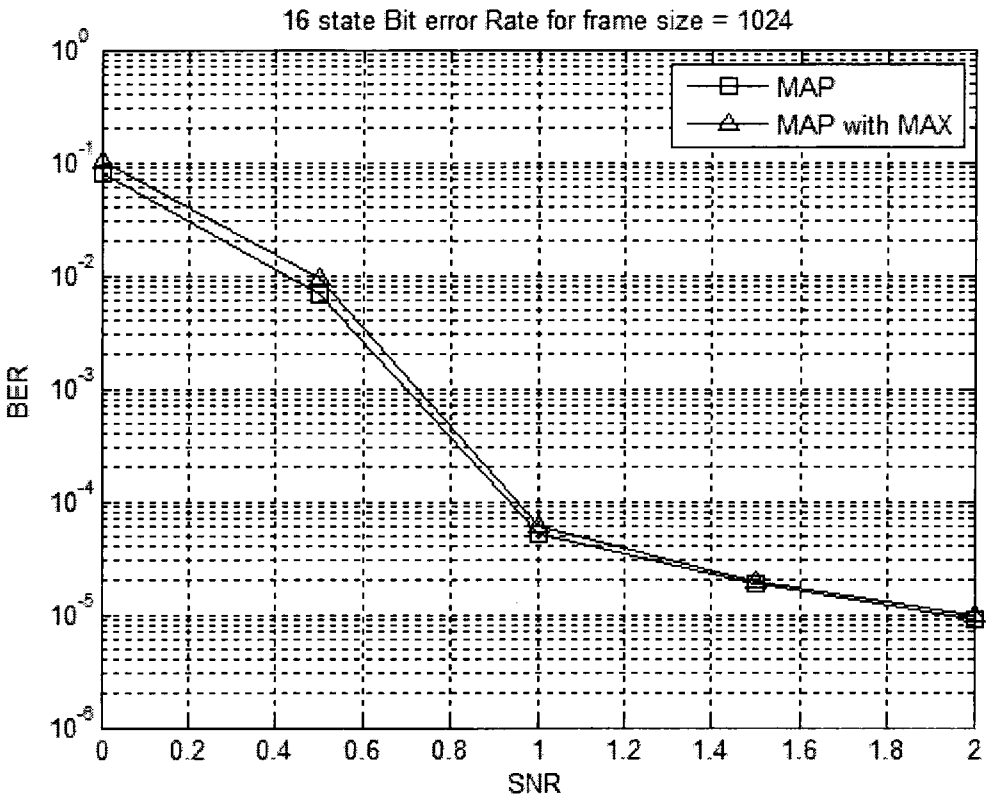

Figure 4.19: Comparison of single algorithm with double algorithm.

For simplicity, we call this method as hybrid algorithm for turbo decoder. The hybrid algorithm has some degradation in BER, but it can be seen from the hardware simulation that it has significant improvement in throughput of individual module, and also area is very much reduced.

4.3 Hardware Synthesis And Simulation

Hardware synthesis was done using Quartus II version 9.0. Each module is analyzed and synthesized separately to find the resources needed by each individual module. The Time Quest Analysis tool from Altera was used to find the maximum frequency for each individual module. The result is compared with [20] in Table 5.7.

Table 4.8: Resources and Frequency comparison

Module	HDL		Hybrid HDL		[20]	
	ALUTs	F_{max}/MHZ	ALUTs	F_{max}/MHZ	ALUTs	F_{max}/MHZ
GAMMA	356	82.66	356	82.66	1138	141.22
BETA	7616	20.21	5911	63.34	1055	139.86
ALPHA	7616	20.31	5911	63.41	1138	141.22
LLR	6766	20.46	3760	63.98	2111	143.02

The test bench was created to capture the data out from the decoder at the end of five iterations. The simulation was run with Altera ModelSim. The data and parity bits were created using MatLab and the received data was modified to fixed point representation with (10,4), and these values are copied to the memory initialization files linked to the memory module. Once the data and parity are available, the test bench is simulated for 210 us in order to have iterated five times. At the end of the simulation the MatLab script is run to find the number of errors from the hardware simulation.

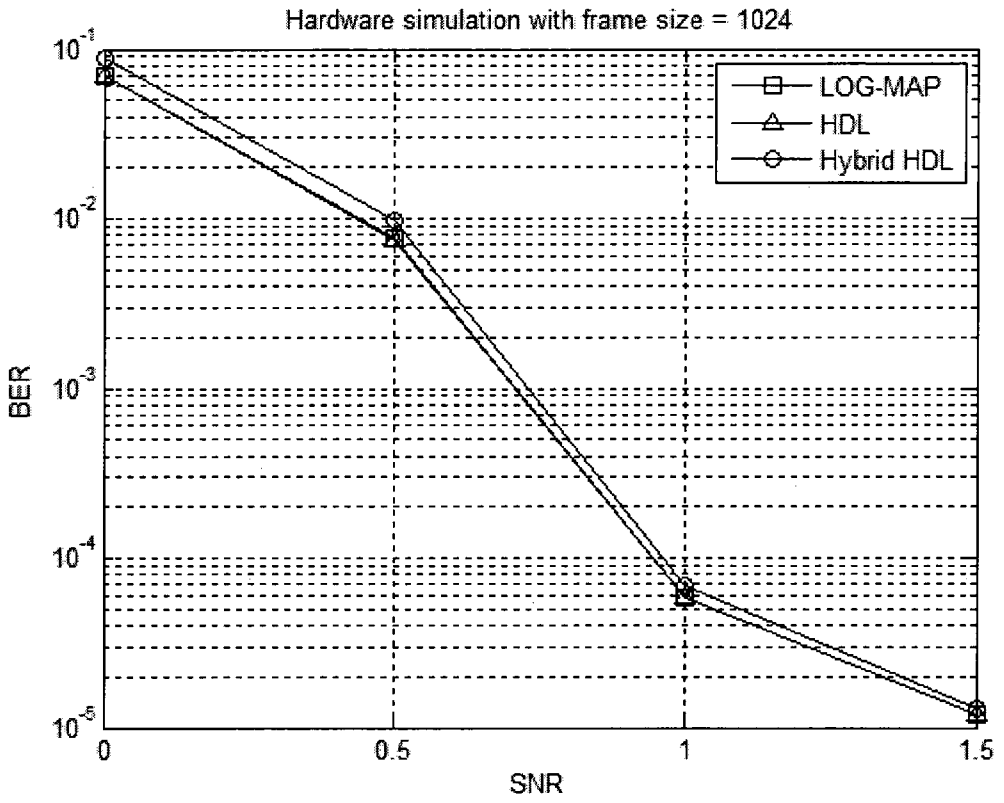

Figure 4.20: Hardware simulation for both realizations to compare with MatLab simulation.

4.3.1 Hardware Synthesis of Turbo Decoder

After each individual module is analyzed and synthesized the full turbo decoder was implemented by integrating all the modules including memory modules and control module to estimate the resources. Table 4.8 shows the resource usage for both models considered above.

Table 4.9: Resource comparison for the Turbo Decoder.

Resource	HDL with mylog104		HDL with hybrid algorithm	
	# of resources	Percentage	# of resources	Percentage
ALUTs	31,191	22%	21,159	14%
DSPs	256	33%	68	9%
MEMORY	409,600	4%	409,600	4%
LOGIC REG	36	1%	36	1%

4.4 Novelties of Implementation

1. All the modules needed for alpha, beta, gamma, and LLR are integrated within each individual module in order to reduce the internal connection delays so that the throughput of each module is increased.

2. Number of adders needed for both alpha and beta normalization are reduced to 4 from 124. This will reduce the area needed and the power consumption by the turbo decoder.

3. Number of clock cycle is minimized by means of integrating combinational module into a single module that is enabled with clock transition.

4. Normalization of alpha and beta is done using total alpha and total beta respectively in order to avoid the memory needed to save the total alpha.

5. Gamma values are written into the memory while alpha and beta values are calculated and alpha and beta are written while gamma values are computed. This will reduce the number of clock cycle by 50%.

4.5 Observation from the Hardware Simulation

The HDL realization of Turbo decoder with mylog104 algorithm has the BER performance similar to LOG-MAP, whereas the implementation of turbo decoder with two algorithm; one for alpha, gamma, and beta, and the other for normalization and LLR has the throughput similar to MAX-LOG-MAP. The second implementation has very small deviation from the LOG-MAP in terms of BER performance.

Table 4.10 compares the throughput with some of the implementations. Our implementation shows a significant improvement in throughput while it maintains the performance, in terms of BER, of the Turbo decoder.

Table 4.10 Comparison of throughput with recent implementation

Ref	# of Processor	Technology	Throughput/(Mbps)
[17]	64	VLSI	930
[18]	2	VLSI	27.6
[19]	16	FPGA	1600*
[16]	1	FPGA	27
[20]	1	FPGA	14.4
[8]	1	FPGA	26
This work	1	FPGA	51

**The implementation is targeted to Virtex 5 which has 65nm gate technology*

CHAPTER 5
CONCLUSIONS

In this thesis we have implemented a very high speed turbo decoder with the new algorithm which gives optimal performance in terms of BER. We have also shown an architecture for normalization module to reduce the number of adders and MAX*, while improving the critical path delay, and also a hybrid architecture was implemented to increase the data rate while reducing the area needed by the turbo decoder, without affecting the BER performance.

It can be seen from the MatLab simulationan hardware simulation, the architecture of the turbo decoder does not degrade the BER performance of the decoder at any SNR, ranging from very low to high, compared to the LOG-MAP algorithm. This does not imply that the algorithm is not sensitive to the SNR mismatch. Only MAX-LOG-MAP is not sensitive to SNR mismatch. But all the log versions of the MAP algorithm has minimal sensitive to SNR mismatch, whereas the MAP algorithm is totally dependent upon how accurately the SNR can be evaluated from the channel information.

In hardware implementation of turbo decoder, there is always a tradeoff between performance parameters, such as area, power, cost, speed, and BER. If the decoder is implemented with the algorithm proposed for the branch matrix, forward and backward matrices and the normalization and the LLR with MAX-LOG-MAP, we will end up with very high data rate with slight degradation on BER performance, which results in giving reduction in area and power. On the other hand, if the matrices calculations are powered with the new MAX* algorithm and the others with MAX-LOG-MAP with ESF, the decoder exhibits a similar BER performance compared to LOG-MAP with significant data rate reduction and slight increase in area. Therefore it is totally dependent upon the application where the turbo decoder is to be implemented.

The Stratix II has 90nm gate technology. If we implement the same architecture in top end FPGA, the data rate will have significant improvement over the old technology and the area is reduced as well.

REFERENCES

1. C.Berrou, A. Glavieus, and P. Thitimajshima, "Near Shannon Limit Error-correcting Coding: Turbo codes," *Proc. 1993 IEEE International conference on Communication*, pp. 1064-1070, May 1993.
2. W.J. Gross and P.G.Gulak, "Simplified MAP algorithm suitable for implementation of turbo decoders," *Electronics Letters*, vol. 34, no. 16, pp. 1577-1578, August 1998.
3. Patric Robertson, Peter Hoeher, and Emmanuelle Villebrun "Optimal and sub-optimal MAP algorithms suitable for turbo decoding," *European Trans. On Telecomm,* Vol. 8 no. 2, pp. 119-126, March –April 1997.
4. Shahram Talakoub, Leila Sabeti, Behnam Shahrrava, and Majid Ahmadi, "An Improved Max-Log-MAP algorithm for Turbo decoding and Turbo Equalization," *IEEE transactions on Instrumentation and measurement*, vol. 56, NO. 3, pp. 1058-1063, June 2007.
5. Ashwani Singh, E.Boutillon, and G. Masera, "Bit width optimization of extrinsic information in Turbo decoder," *5th international symposium on Turbo codes and Related topics*, pp. 134-138, 2008.
6. L.Bahl, J. Cocke, F. Jelinek, and J.Raviv, "Optimum decoding of linear codes for minimizing symbol error rate," *IEEE Trans. On Inf. Theory*, vol. IT-20, pp. 284-287, Mar 1974.
7. B.P.Lathi and Zhi Ding, "Error Correcting Codes" in *Modern digital and analog communication systems*, Forth edition. New York: Oxford University press, 2009, pp.951-959.
8. Michel J.Thul, and Norber When, *"FPGA implementation of parallel turbo decoders," Integrated circuits and systems Design*, SBCCI 2004, 17th Symposium, pp. 198 – 203, 2004.
9. Alexander Worm, Peter Hoeher, and Norbert When, "Turbo decoding without SNR estimation," *IEEE Communications,* Vol.4, N0.6, pp. 193-195, June 2000.
10. Third Generation Partnership Project 2(3GPP2), Physical layer standard for cdma2000 spread spectrum systems, Release D, version 1, Feb 2004.
11. IEEE standard for local and metropolitan area networks. Part 16: air interface for fixed broadband wireless access systems, Nov 2004.
12. Hamid R Sadjadpour, "Maximum a posteriori decoding algorithms for turbo codes," *Proceedings of SPIE*, vol. 4045, pp. 73-83, 2000.
13. M.J.Thuul, F. Gilbert, T. Vogt, G. Kreiselmaier, and N. When, "A Scalable system Architecture for High Throughput Turbo decoders," *Journal of VLSI Signal Processing systems*, Vol. 39, pp. 63-77, 2005.
14. G. Prescher, T. Gemmeke, and T. Noll, *"A Parameterizable Low-Power High-throughput Turbo Decoder," IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 25-28, Mar 2005.
15. Oscar Y. Takeshita, "On Maximum Contention Free Interleavers and Permutation Polynomials Over Integer Rings," *IEEE Transactions on Information Theory*, vol. 52, NO. 3, pp. 1249-1253 March 2006.

16. Duk Gun Choi, Min-Hyuk Kim, Jin Hee Jeong, and Ji Won Jung, "An FPGA Implementation of High speed Flexible 27-Mbps 8-state Turbo Decoder," *ETRI Journal*, Vol 29, NO. 3, pp. 363-370, June 2007.
17. Karim. S. M, and Chakrabarti. I, "An improved low power high throughput LOG-MAP turbo decoder," *Consumer Electronics, IEEE transactions on*, Issue 2, pp. 450-457, May 2010.
18. S.J.Lee, N.R. Shanbhag, and A.C. Singer, "A 285-MHz pipelined MAP decoder in 0.18 um CMOS," *IEEE journal of solid state circuits*, vol. 40, NO. 8, pp. 1718-1725, August 2005.
19. Martin.I.del Barco, Gabriel N. Maggio, and Damian. A .Morero, "FPGA Implementation of high speed parallel maximum a posteriori(MAP) decoders," *Proceeding of the Argentine School of Micro-Nano electronics Technology and Applications*, pp. 98-102, 2009.
20. Roberto Ramirez Martin, Andres David Garcia Garcia, Luis Fernando Gonzalez Perez, and Javier Eduardo Gonzalez Villarruel, "Hardware architecture of MAP algorithm for Turbo Codes implemented in a FPGA," *Proceedings of the 15th international Conference on Electronics, Communications and Computers*, pp. 70-75, 2005.
21. Boutillon E., Douillard C, and Montorsi G, "Iterative decoding of concatenated convolutional codes: Implementaiton Issues," *Proceeding of the IEEE*, vol 95, Issue 6, pp. 1201-1227, June 2007.
22. Yuping Zhang, and Keshab K.Parhi, "Parallel Turbo Decoding," *IEEE International Symposium on Circuits and Systems*, pp. 509-512, 2004.
23. F. Gilbert, F. Kienle, and N. When, "Low complexity stopping criteria for UMTS turbo decoders," Vehicular *Technology Conference*, vol.4, pp. 2376-2380, 2003.
24. J. Hagenauer, E. Offer, and L. Papke. "Iterative Decoding of Binary Block and Convolutional Codes," *IEEE Transactions on Information Theory*, vol.42, no 2, pp. 429-445, Mar 1996.
25. F. Zhai and I. J .Fair. "New Error Detection Techniques and Stopping Criteria for Turbo Decoding," *in Proc. 2000 IEEE Canadian Conference on Electrical and Computer Engineering*, pp. 58-62, Mar 2000.
26. Z. Whang and K. K. Parhi, "Decoding Metrics and their Applications in VLSI Turbo Decoders," *In Proc. 2000 Conference on Acoustics, Speech, and Signal Processing*, pp. 3370-3373, Sept 2000.
27. R. Y. Shao, S. Lin, and M. C. P. Fossorier, "Two Simple Stopping Criteria for Turbo Decoding," *IEEE Transactions on Communications*, vol. 47, no. 8, pp. 1117-1120, Aug 1999.
28. A. Shibutani, H. Suda, and F. Adachi, "Reducing average number of turbo decoding iterations," *Electron. Lett.*, vol.35, no. 9, pp. 701-702, Apr 1999.
29. Ajit Nimbalker, T. Blankenship, Brian Classon, Thomas E. Fuja, and Daniel J. Costello, "Contention-Free Interleavers for High-throughput Turbo Decoding," *IEEE Transactions on Communications*, vol. 56, no. 8, pp. 1258-1266, August 2008.
30. T. K. Blankenship, B. Classon, and V. Desai, "Channel coding for 4G systems with adaptive modulation and coding," *IEEE Wireless Communication Mag.*, vol. 9, pp. 8-13, Apr 2002.
31. A. Giuletti, L. van der Perre, and M. Strun, *"Parallel turbo decoding interleavers : avoiding collisions in accesses to storage elements," Electron. Lett.*, vol. 38, pp. 232-234, Feb 2002.

32. A. Nimbalker, T. K. Blankenship, B. Classon, T.E. Fuja, and D. J. Costello, "Inter-Window shuffle interleavers for high throughput turbo decoding," *in Proc. Int. Symp. Turbo Codes and Related Topics*, pp. 355-358, Sept 2003.
33. M. J. Thul, F. Gillbert, and N. When, "Optimized Concurrent interleaving architecture for high throughput turbo decoding," *in Proc, Int. Conf. Electronics Circuits and Systems*pp 1099-1102, Sept 2002.
34. A. Tarable, S Benedetto, and B Montorsi, "Mapping Interleavers laws to parallel turbo and LDPC decoder architectures," *IEEE Trans. Information theory*, vol.50, pp. 2002-2009, Sept 2004.
35. C. Schurger, F. Catthoor, and M. Engel, "Memory optimization of MAP turbo decoder algorithms," *IEEE Trans. on VLSI system*, vol. 9, no. 2, pp. 305-312, April 2001.

APPENDIX

A1. Matlab codes used in this thesis

```
% This m file simulates all algorithms
% MAX-LOG-MAP, Improved-MAX-LOG-MAP, LOG-MAP, Simplified-MAX-LOG-MAP
% and the proposed algorithms are considered.
% Channel is assumed to be Assumed to be AWGN.
% for different generator g must be changed
% Frame size and frame limit can be changed
% iteration can be changed
clear all
%diary myfileallalgo.txt
a = 1;                %channel fading factor
ferrlim = 25;          %Frame limit
niter = 5;             %Iteration
Infty = 1e20;          % DEfine the infinity
g = [1 0 0 0 1;1 1 1 1 1]; %generator polynomial
%Find the next state and previous states
[n,k] = size(g);   %find the number of rows and columns
 m = k-1;         %memory elements in the encoder
[next0,next1] = stateout(g); %call the function stateout
%generate the next_state matrix containing state transitions and
% the input and output associated with it.
next_state = [next0(:,2) next1(:,2) next0(:,3) next1(:,3)];
[temp,alp] = sort(next0(:,2));
[temp,alpp] = sort(next1(:,2));
%previous states are generated.
pre_state = [alp alpp];

%no punctures
puncture = 1;
rate = 1/(2+puncture);     % rate is assumed to be 1/3
nstates = 2^m;          % number of states in the encoder
N = 1024;              % Frame length.
L_total =N+m;          % Tail bits for terminating first encoder
[temp alphaD] = sort(rand(1,L_total-m)); %Random interleaver.
temp1 = max(alphaD);
%Tail bits are not interleaved
newalphaD = [alphaD temp1+1:temp1+m];


EbN0db =[0  0.5 1 1.5 2 ]; %define the SNR range
nEN = 1;
for nEN = 1:length(EbN0db)
  %define the matirx for holding errors for each SNR
```

```
  %Initialize to zeros.
  errsLogD(nEN,1:niter) = zeros(1,niter);  % erros for LOG-MAP
  errsCor1D(nEN,1:niter) = zeros(1,niter); % errors for PROPOSED
  errsSimD(nEN,1:niter) = zeros(1,niter);  % Errors for Simplified
  errsImpD(nEN,1:niter) = zeros(1,niter);  % Errors for Improved
  nferrImpD(nEN,1:niter) = zeros(1,niter); % Frame errors for Improved
  nferrCor1D(nEN,1:niter) = zeros(1,niter); %frame errors for proposed
  nferrLogD(nEN,1:niter) = zeros(1,niter);  %Frame errors for Log-map
nferrSimD(nEN,1:niter) = zeros(1,niter);  %Frame errors for simplified

  en = 10^(EbN0db(nEN)/10);  %Calculate in terms of energy
  L_c = 4*a*en*rate;       %Channel coefficient
  sigma = 1/sqrt(2*rate*en); %Noise Variant
  nframe = 0;              %Initialize number of frames sent to zero

while nferrImpD(nEN,niter) < ferrlim
   nframe = nframe+1;
   x = round(rand(1,L_total-m)); % Create random data
   % Call myencode_bit function to encode the data
   en_outputD = myencode_bit(x,g,alphaD); %encode the data
   rD = en_outputD + sigma*randn(1,L_total*3); %add the random noise
   %extract the data for decoder 1 and decoder 2
   %Call mydemultiplex function
   ykD = mydemultiplex(rD,alphaD,m);
   ykmD = 0.5*L_c*ykD;  %Multiply with the channel coefficient
   rec_sD = ykmD(1,:);  %Data for decoder 1
   rec_s2D = ykmD(2,:); %Data for decoer 2

  %Initally extrinsic values are set to 0
  L_ecor1D(1:L_total) = zeros(1,L_total); %Extrinsic for proposed
  L_eImpD(1:L_total) = zeros(1,L_total);  %Extrinsic for Improved
  L_eLogD(1:L_total) = zeros(1,L_total);  %Extrinsic for Log-MAP
  L_eSimD(1:L_total) = zeros(1,L_total);  %Extrinsic for Simplified

  for iter = 1:niter
  %Alpha and Beta matrices are initialized for decoder 1
  %for proposed algorithm
  cor1Alpha1D(1,1) = 0;
  cor1Alpha1D(1,2:nstates) = -Infty;
  cor1Beta1D = -Infty*ones(L_total+1,nstates);
  cor1Beta1D(L_total+1,1) = 0;
  %for LOG-MAP algorithm
  LogAlpha1D(1,1) = 0;
  LogAlpha1D(1,2:nstates) = -Infty;
  LogBeta1D = -Infty*ones(L_total+1,nstates);
  LogBeta1D(L_total+1,1) = 0;
```

```
  %for Simplified algorithm
  SimAlpha1D(1,1) = 0;
  SimAlpha1D(1,2:nstates) = -Infty;
  SimBeta1D = -Infty*ones(L_total+1,nstates);
  SimBeta1D(L_total+1,1) = 0;

  %for Improved algorithm
  ImpAlpha1D(1,1) = 0;
  ImpAlpha1D(1,2:nstates) = -Infty;
  ImpBeta1D = -Infty*ones(L_total+1,nstates);
  ImpBeta1D(L_total+1,1) = 0;
%deinterleave the extrinsic values from the second decoder
  L_acor1D(newalphaD) = L_ecor1D;
  L_aLogD(newalphaD) = L_eLogD;
  L_aImpD(newalphaD) = L_eImpD;
  L_aSimD(newalphaD) = L_eSimD;

for i1 = 1:L_total
   %Branch matrices associated with input 1 and input 0 for
   %Log-MAP, Simplifed, proposed, and Improved are initialized to
   % negative infinity.
   gamma0LogD(i1,1:nstates) = -Infty;
   gamma1LogD(i1,1:nstates) = -Infty;
   gamma0SimD(i1,1:nstates) = -Infty;
   gamma1SimD(i1,1:nstates) = -Infty;
   gamma0cor1D(i1,1:nstates) = -Infty;
   gamma1cor1D(i1,1:nstates) = -Infty;
   gamma0ImpD(i1,1:nstates) = -Infty;
   gamma1ImpD(i1,1:nstates) = -Infty;

for state = 1:nstates
     %find the  state transition probability for each algorithm.
     %findlog,Impfindlog,Simfindlog are used to calculate
     % log(exp(x)+exp(y)) based on the algorithm of its own.
     llrcor1D = findlog([0 L_acor1D(i1)]);
     llrLogD = log(1+exp(L_aLogD(i1)));
     llrImpD = Impfindlog([0 L_aImpD(i1)]);
     llrSimD = Simfindlog([0 L_aSimD(i1)]);

     %find the branch metric for the transition
     %This gives the transition from the previous to next state
   gamma0cor1D(i1,state)=-rec_sD(2*i1-1)+rec_sD(2*i1)...
              *next_state(state,3)-llrcor1D;
   gamma1cor1D(i1,state) = rec_sD(2*i1-1)+rec_sD(2*i1)...
            *next_state(state,4)+L_acor1D(i1)-llrcor1D;
   gamma0ImpD(i1,state) = -rec_sD(2*i1-1)+rec_sD(2*i1)...
```

```
            *next_state(state,3)-llrImpD;
    gamma1ImpD(i1,state) = rec_sD(2*i1-1) + rec_sD(2*i1)...
            *next_state(state,4)+L_aImpD(i1)-llrImpD;
    gamma0LogD(i1,state) = (-rec_sD(2*i1-1)+rec_sD(2*i1)...
            *next_state(state,3)-llrLogD);
    gamma1LogD(i1,state) = (rec_sD(2*i1-1)+rec_sD(2*i1)...
            *next_state(state,4)+L_aLogD(i1)-llrLogD);
    gamma0SimD(i1,state) = -rec_sD(2*i1-1)+ rec_sD(2*i1)...
             *next_state(state,3)-llrSimD;
    gamma1SimD(i1,state) = rec_sD(2*i1-1) + rec_sD(2*i1)...
             *next_state(state,4)+L_aSimD(i1)-llrSimD;

end % end of "for state=..."


% to normalize alpha and beta
total_gamAlcor1D(i1) = findlog([gamma0cor1D(i1,:)+cor1Alpha1D(i1,:)...
             gamma1cor1D(i1,:)+cor1Alpha1D(i1,:)]);
total_gamAlLogD(i1) =log(sum(exp(gamma0LogD(i1,:) +LogAlpha1D(i1,:))...
             +exp(gamma1LogD(i1,:)+LogAlpha1D(i1,:))));
total_gamAlImpD(i1) =Impfindlog([gamma0ImpD(i1,:)+ImpAlpha1D(i1,:)...
             gamma1ImpD(i1,:)+ImpAlpha1D(i1,:)]);
total_gamAlSimD(i1) = Simfindlog([gamma0SimD(i1,:)+SimAlpha1D(i1,:)...
             gamma1SimD(i1,:)+SimAlpha1D(i1,:)]);

 %Assign negative infinity to Alpha
 cor1Alpha1D(i1+1,:) = -Infty;
 LogAlpha1D(i1+1,:) = -Infty;
 ImpAlpha1D(i1+1,:) = -Infty;
 SimAlpha1D(i1+1,:) = -Infty;

for sta1 = 1:nstates
 tempAlpha1 = exp(gamma0LogD(i1,pre_state(sta1,1))...
       +LogAlpha1D(i1,pre_state(sta1,1)))...
        +exp(gamma1LogD(i1,pre_state(sta1,2))...
        +LogAlpha1D(i1,pre_state(sta1,2)));
  %to avoid getting NaN
  if tempAlpha1 < 1e-300
    LogAlpha1D(i1+1,sta1) = -Infty;
  else
  LogAlpha1D(i1+1,sta1) = log(tempAlpha1)- total_gamAlLogD(i1);
  end

cor1Alpha1D(i1+1,sta1)=findlog([gamma0cor1D(i1,pre_state(sta1,1))...
             +cor1Alpha1D(i1,pre_state(sta1,1))...
             gamma1cor1D(i1,pre_state(sta1,2))...
```

```
                + cor1Alpha1D(i1,pre_state(sta1,2))])...
                -total_gamAlcor1D(i1);
SimAlpha1D(i1+1,sta1)=Simfindlog([gamma0SimD(i1,pre_state(sta1,1))...
                +SimAlpha1D(i1,pre_state(sta1,1))...
                 gamma1SimD(i1,pre_state(sta1,2))...
                 + SimAlpha1D(i1,pre_state(sta1,2))])...
                 -total_gamAlSimD(i1);
ImpAlpha1D(i1+1,sta1)=Impfindlog([gamma0ImpD(i1,pre_state(sta1,1))...
                +ImpAlpha1D(i1,pre_state(sta1,1))...
                 gamma1ImpD(i1,pre_state(sta1,2))...
                + ImpAlpha1D(i1,pre_state(sta1,2))])...
                - total_gamAlImpD(i1);
 end

 end
%find the Beta
for i2 = L_total:-1:1
 for sta1 = 1:nstates
  tempBeta1 = exp(gamma0LogD(i2,sta1)+LogBeta1D(i2+1,...
          next_state(sta1,1)))+exp(gamma1LogD(i2,sta1)...
          +LogBeta1D(i2+1,next_state(sta1,2)));
  %to avoid NaN
  if tempBeta1 < 1e-300
    LogBeta1D(i2,sta1) = -Infty;
  else
  LogBeta1D(i2,sta1) = log(tempBeta1)- total_gamAlLogD(i2);
  end

 cor1Beta1D(i2,sta1) = findlog([ gamma0cor1D(i2,sta1)+cor1Beta1D...
             (i2+1,next_state(sta1,1)) gamma1cor1D(i2,sta1)...
              +cor1Beta1D(i2+1,next_state(sta1,2))])...
              - total_gamAlcor1D(i2);
 ImpBeta1D(i2,sta1) = Impfindlog([ gamma0ImpD(i2,sta1)...
               +ImpBeta1D(i2+1,next_state(sta1,1))...
                gamma1ImpD(i2,sta1)+ImpBeta1D(i2+1,...
                next_state(sta1,2))])- total_gamAlImpD(i2);
 SimBeta1D(i2,sta1) = Simfindlog([ gamma0SimD(i2,sta1)...
                +SimBeta1D(i2+1,next_state(sta1,1))...
                  gamma1SimD(i2,sta1)+SimBeta1D(i2+1,...
                  next_state(sta1,2))])- total_gamAlSimD(i2);
 end

end

for da = 1:L_total
%temporarily assign the beta values correspoding to alpha and gamma
```

```
    cor1tempBeta10D = cor1Beta1D(da+1,next_state(:,1)');
    cor1tempBeta11D = cor1Beta1D(da+1,next_state(:,2)');
    LogtempBeta10D = LogBeta1D(da+1,next_state(:,1)');
    LogtempBeta11D = LogBeta1D(da+1,next_state(:,2)');
    SimtempBeta10D = SimBeta1D(da+1,next_state(:,1)');
    SimtempBeta11D = SimBeta1D(da+1,next_state(:,2)');
    ImptempBeta10D = ImpBeta1D(da+1,next_state(:,1)');
    ImptempBeta11D = ImpBeta1D(da+1,next_state(:,2)');
  %Calculate LLR for each algorithm
  L_allcor1D(da) = findlog([gamma1cor1D(da,:)+cor1Alpha1D(da,:)...
           +cor1tempBeta11D])-findlog([gamma0cor1D(da,:)...
           +cor1Alpha1D(da,:)+cor1tempBeta10D]);
  L_allSimD(da) = Simfindlog([gamma1SimD(da,:)+SimAlpha1D(da,:)...
           +SimtempBeta11D])-Simfindlog([gamma0SimD(da,:)...
           +SimAlpha1D(da,:)+SimtempBeta10D]);
  L_allImpD(da) = Impfindlog([gamma1ImpD(da,:)+ImpAlpha1D(da,:)...
           +ImptempBeta11D])-Impfindlog([gamma0ImpD(da,:)...
           +ImpAlpha1D(da,:)+ImptempBeta10D]);
  L_allLogD(da)=log(sum(exp(gamma1LogD(da,:)+LogAlpha1D(da,:)...
          +LogtempBeta11D)))- log(sum(exp(gamma0LogD(da,:)...
          +LogAlpha1D(da,:)+LogtempBeta10D)));
  end
   % Calculate the extrinsic values for each algorithm
   % these need to be interleaved for the second decoder
   L_ecor1D = L_allcor1D - 2*rec_sD(1,1:2:2*L_total) - L_acor1D;
   L_eSimD = L_allSimD - 2*rec_sD(1,1:2:2*L_total) - L_aSimD;
   L_eLogD = L_allLogD - 2*rec_sD(1,1:2:2*L_total) - L_aLogD;
   L_eImpD = L_allImpD - 2*rec_sD(1,1:2:2*L_total) - L_aImpD;

  %First decoder finishes its process and passes extrinsic values
  %to the second decoder
  % Initialize Alpha and Beta for the second decoder
  % Second decoder is not terminated; Left open
  % for proposed algorithm
  cor1Alpha2D(1,1) = 0;
  cor1Alpha2D(1,2:nstates) = -Infty;
  cor1Beta2D = -Infty*ones(L_total+1,nstates);
  cor1Beta2D(L_total+1,1:nstates) = log(1/nstates);
   %for simplified log-map
  SimAlpha2D(1,1) = 0;
  SimAlpha2D(1,2:nstates) = -Infty;
  SimBeta2D = -Infty*ones(L_total+1,nstates);
  SimBeta2D(L_total+1,1:nstates) = log(1/nstates);
   %for log-map algorithm
  LogAlpha2D(1,1) = 0;
  LogAlpha2D(1,2:nstates) = -Infty;
```

```
LogBeta2D = -Infty*ones(L_total+1,nstates);
LogBeta2D(L_total+1,1:nstates) = log(1/nstates);
%for improved log-map algorithm
ImpAlpha2D(1,1) = 0;
ImpAlpha2D(1,2:nstates) = -Infty;
ImpBeta2D = -Infty*ones(L_total+1,nstates);
ImpBeta2D(L_total+1,1:nstates) = log(1/nstates);

% Interleave extrinsic values from the first decoder
% Temporary variables to hold the interleaved extrinsic values
% from decoder 1.
L_a2cor1D = L_ecor1D(newalphaD);
L_a2ImpD = L_eImpD(newalphaD);
L_a2SimD = L_eSimD(newalphaD);
L_a2LogD = L_eLogD(newalphaD);


for i1 = 1:L_total

  % Initialize all gamma matrices associated with input 1 and 0 for
  % decoder 2 to Negative Infinity.

  gamma20cor1D(i1,1:nstates) = -Infty;
  gamma21cor1D(i1,1:nstates) = -Infty;
  gamma20ImpD(i1,1:nstates) = -Infty;
  gamma21ImpD(i1,1:nstates) = -Infty;
  gamma20SimD(i1,1:nstates) = -Infty;
  gamma21SimD(i1,1:nstates) = -Infty;
  gamma20LogD(i1,1:nstates) = -Infty;
  gamma21LogD(i1,1:nstates) = -Infty;


 for sta2 = 1:nstates
  %find the probability for state transition
   llrcor1D = findlog([0 L_a2cor1D(i1)]);
   llrSimD = Simfindlog([0 L_a2SimD(i1)]);
   llrLogD = log(1+exp(L_a2LogD(i1)));
   llrImpD = Impfindlog([0 L_a2ImpD(i1)]);
  % Calculateee the branch metric from the systematic data
  % and the parity bits from the second decoder
 gamma20cor1D(i1,sta2) = -rec_s2D(2*i1-1) + rec_s2D(2*i1)...
                *next_sta2(sta2,3)-llrcor1D;
 gamma21cor1D(i1,sta2) = rec_s2D(2*i1-1) + rec_s2D(2*i1)...
               *next_sta2(sta2,4)+L_a2cor1D(i1)-llrcor1D;
 gamma20ImpD(i1,sta2) = -rec_s2D(2*i1-1) + rec_s2D(2*i1)...
                *next_sta2(sta2,3)-llrImpD;
 gamma21ImpD(i1,sta2) = rec_s2D(2*i1-1) + rec_s2D(2*i1)...
```

```
                *next_sta2(sta2,4)+L_a2ImpD(i1)-llrImpD;
  gamma20LogD(i1,sta2) = -rec_s2D(2*i1-1) + rec_s2D(2*i1)...
                *next_sta2(sta2,3)-llrLogD;
  gamma21LogD(i1,sta2) = rec_s2D(2*i1-1) + rec_s2D(2*i1)...
                *next_sta2(sta2,4)+L_a2LogD(i1)-llrLogD;
  gamma20SimD(i1,sta2) = -rec_s2D(2*i1-1) + rec_s2D(2*i1)...
                *next_sta2(sta2,3)-llrSimD;
  gamma21SimD(i1,sta2) = rec_s2D(2*i1-1) + rec_s2D(2*i1)...
               *next_sta2(sta2,4)+L_a2SimD(i1)-llrSimD;

 end %sta2
 % Normalization values for alpha and beta
total_gamA2cor1D(i1) = findlog([gamma20cor1D(i1,:)+cor1Alpha2D...
            (i1,:) gamma21cor1D(i1,:)+cor1Alpha2D(i1,:)]);
total_gamA2SimD(i1) = Simfindlog([gamma20SimD(i1,:)+SimAlpha2D...
             (i1,:) gamma21SimD(i1,:)+SimAlpha2D(i1,:)]);
total_gamA2LogD(i1) =  log(sum(exp(gamma20LogD(i1,:) +LogAlpha2D...
           (i1,:))+exp(gamma21LogD(i1,:)+LogAlpha2D(i1,:))));

total_gamA2ImpD(i1) = Impfindlog([gamma20ImpD(i1,:)+ImpAlpha2D...
            (i1,:) gamma21ImpD(i1,:)+ImpAlpha2D(i1,:)]);

cor1Alpha2D(i1+1,:) = -Infty;
SimAlpha2D(i1+1,:) = -Infty;
LogAlpha2D(i1+1,:) = -Infty;
ImpAlpha2D(i1+1,:) = -Infty;


for sta1 = 1:nstates
   tempAlpha2 = exp(gamma20LogD(i1,pre_sta2(sta1,1))...
          +LogAlpha2D(i1,pre_sta2(sta1,1)))...
            +exp(gamma21LogD(i1,pre_sta2(sta1,2))...
             +LogAlpha2D(i1,pre_sta2(sta1,2)));
   if tempAlpha2 < 1e-300
     LogAlpha2D(i1+1,sta1) = -Infty;
   else
   LogAlpha2D(i1+1,sta1) = log(tempAlpha2)- total_gamA2LogD(i1);
   end

cor1Alpha2D(i1+1,sta1) = findlog([gamma20cor1D(i1,pre_state(sta1,1))...
          +cor1Alpha2D(i1,pre_states(sta1,1)) gamma21cor1D...
          (i1,pre_states(sta1,2))+cor1Alpha2D(i1,...
           pre_states(sta1,2))])-total_gamA2cor1D(i1);
ImpAlpha2D(i1+1,sta1) =Impfindlog([gamma20ImpD(i1,pre_state(sta1,1))...
           +ImpAlpha2D(i1,pre_states(sta1,1)) gamma21ImpD...
           (i1,pre_states(sta1,2))+ ImpAlpha2D(i1,...
```

```
            pre_states(sta1,2))])-total_gamA2ImpD(i1);

SimAlpha2D(i1+1,sta1)=Simfindlog([gamma20SimD(i1,pre_state(sta1,1))...
          +SimAlpha2D(i1,pre_state(sta1,1)) gamma21SimD(i1,...
          pre_state(sta1,2))+ SimAlpha2D(i1,pre_sta2(sta1,2))])...
          -total_gamA2SimD(i1);
end

end

for i2 = L_total:-1:1
 for sta1 = 1:nstates
  tempBeta2 = exp(gamma20LogD(i2,sta1)+LogBeta2D(i2+1,...
          next_state(sta1,1)))+exp(gamma21LogD(i2,sta1)...
          +LogBeta2D(i2+1,next_state(sta1,2)));

  if tempBeta2< 1e-300
    LogBeta2D(i2,sta1) = -Infty;
  else
  LogBeta2D(i2,sta1) = log(tempBeta2)- total_gamA2LogD(i2);
  end


 cor1Beta2D(i2,sta1) = findlog([ gamma20cor1D(i2,sta1)+cor1Beta2D...
          (i2+1,next_state(sta1,1)) gamma21cor1D(i2,sta1)...
           +cor1Beta2D(i2+1,next_state(sta1,2))])...
           - total_gamA2cor1D(i2) ;
 ImpBeta2D(i2,sta1) = Impfindlog([ gamma20ImpD(i2,sta1)+ImpBeta2D...
             (i2+1,next_state(sta1,1)) gamma21ImpD(i2,sta1)...
             +ImpBeta2D(i2+1,next_state(sta1,2))])...
             - total_gamA2ImpD(i2) ;

SimBeta2D(i2,sta1) = Simfindlog([ gamma20SimD(i2,sta1)+SimBeta2D...
             (i2+1,next_state(sta1,1)) gamma21SimD(i2,sta1)...
             +SimBeta2D(i2+1,next_state(sta1,2))])...
             - total_gamA2SimD(i2) ;

   end

 end

 for da = 1:L_total
   %temporarily assign the beta values associated with alpha
   % and gamma
   cor1tempBeta20D = cor1Beta2D(da+1,next_state(:,1)');
   cor1tempBeta21D = cor1Beta2D(da+1,next_state(:,2)');
```

```
    SimtempBeta20D = SimBeta2D(da+1,next_state(:,1)');
    SimtempBeta21D = SimBeta2D(da+1,next_state(:,2)');

    LogtempBeta20D = LogBeta2D(da+1,next_state(:,1)');
    LogtempBeta21D = LogBeta2D(da+1,next_state(:,2)');
    ImptempBeta20D = ImpBeta2D(da+1,next_state(:,1)');
    ImptempBeta21D = ImpBeta2D(da+1,next_state(:,2)');

 %LLR values are calculated for decoder 2
 L_all2cor1D(da) = findlog([gamma21cor1D(da,:)+cor1Alpha2D(da,:)...
             +cor1tempBeta21D])- findlog([gamma20cor1D(da,:)...
              +cor1Alpha2D(da,:)+cor1tempBeta20D]);
 L_all2ImpD(da) = Impfindlog([gamma21ImpD(da,:)+ImpAlpha2D(da,:)...
             +ImptempBeta21D])- Impfindlog([gamma20ImpD(da,:)...
             +ImpAlpha2D(da,:)+ImptempBeta20D]);
  L_all2LogD(da)=log(sum(exp(gamma21LogD(da,:)+LogAlpha2D(da,:)...
             +LogtempBeta21D)))- log(sum(exp(gamma20LogD(da,:)...
             +LogAlpha2D(da,:)+LogtempBeta20D)));
  L_all2SimD(da) = Simfindlog([gamma21SimD(da,:)+SimAlpha2D(da,:)...
             +SimtempBeta21D])- Simfindlog([gamma20SimD(da,:)...
             +SimAlpha2D(da,:)+SimtempBeta20D]);
 end
  %Extrinsic values are calculated for decoder 1
  L_ecor1D = L_all2cor1D - 2*rec_s2D(1:2:2*L_total) - L_a2cor1D;
  L_eSimD = L_all2SimD - 2*rec_s2D(1:2:2*L_total) - L_a2SimD;
  L_eImpD = L_all2ImpD - 2*rec_s2D(1:2:2*L_total) - L_a2ImpD;
  L_eLogD = L_all2LogD - 2*rec_s2D(1:2:2*L_total) - L_a2LogD;
  % decode the data from LLR
   xhat2cor1D(newalphaD) = (sign(L_all2cor1D)+1)/2;
   xhat2SimD(newalphaD) = (sign(L_all2SimD)+1)/2;
   xhat2ImpD(newalphaD) = (sign(L_all2ImpD)+1)/2;
   xhat2LogD(newalphaD) = (sign(L_all2LogD)+1)/2;
   % find the errors in each algorithm
   errcor1D(iter)=length(find(xhat2cor1D(1:L_total-m) ~= x));
   errSimD(iter)=length(find(xhat2SimD(1:L_total-m) ~= x));
   errLogD(iter)=length(find(xhat2LogD(1:L_total-m) ~= x));
   errImpD(iter)=length(find(xhat2ImpD(1:L_total-m) ~= x));



% if any errors update the frame error for the particular iteration

   if errcor1D(iter) > 0
     nferrCor1D(nEN,iter) = nferrCor1D(nEN,iter) + 1;
   end
```

```
  if errLogD(iter) > 0
     nferrLogD(nEN,iter) = nferrLogD(nEN,iter) + 1;
  end

  if errImpD(iter) > 0
     nferrImpD(nEN,iter) = nferrImpD(nEN,iter) + 1;
  end
  if errSimD(iter) > 0
     nferrSimD(nEN,iter) = nferrSimD(nEN,iter) + 1;
  end


  end

 % accumulate the errors for each iteration
errsCor1D(nEN,1:niter)=errsCor1D(nEN,1:niter) + errcor1D(1:niter);
errsImpD(nEN,1:niter)=errsImpD(nEN,1:niter) + errImpD(1:niter);
errsLogD(nEN,1:niter)=errsLogD(nEN,1:niter) + errLogD(1:niter);
errsSimD(nEN,1:niter)=errsSimD(nEN,1:niter) + errSimD(1:niter);
% display the number of errors and frame
%transmitted for every three frame
 if rem(nframe, 3) == 0 || nferrImpD(nEN,niter) >= ferrlim
  %Bit error rate is calculated for each algorithm
 berCor1(nEN,1:niter) = errsCor1D(nEN,1:niter)/nframe/(L_total -m);
 berLog(nEN,1:niter) = errsLogD(nEN,1:niter)/nframe/(L_total -m);
 berImp(nEN,1:niter) = errsImpD(nEN,1:niter)/nframe/(L_total -m);
 berSim(nEN,1:niter) = errsSimD(nEN,1:niter)/nframe/(L_total -m);
 fprintf('********Frame size = %d **********\n',L_total);
 fprintf('********EbN0      = %5.2f ********\n',EbN0db(nEN));
 fprintf('%d frames transmitted, %d frames in Mul_Cor1 error.\n'...
       ,nframe,nferrCor1D(nEN,niter));
 fprintf('%d frames transmitted, %d frames in Sim error.\n'...
       ,nframe,nferrSimD(nEN,niter));
 fprintf('%d frames transmitted, %d frames in Log error.\n'...
       ,nframe,nferrLogD(nEN,niter));
 fprintf('%d frames transmitted, %d frames in Imp error.\n'...
       ,nframe,nferrImpD(nEN,niter));
 fprintf('Bit error rate\n');
 fprintf('Log Error    MulCor Error    Imp Error   Sim Error\n');
 for i7 = 1:niter
  fprintf('%8.4e    %8.4e    %8.4e  %8.4e  \n',...
    berLog(nEN,i7),berCor1(nEN,i7),berImp(nEN,i7),berSim(nEN,i7));
 end
 end
% Number of frame transmitted is limited to 5000
if nframe >= 5000
```

```
   break;
end
 end %while

end %nDF
% plot the result once the simulation is done
semilogy(EbN0db,berLog(:,niter),'ks-',EbN0db,berCor1(:,niter)...
    ,'k^-',EbN0db,berImp(:,niter),'k*-',...
      EbN0db,berSim(:,niter),'kv-');
legend('MAP','Proposed1','Imp-MAP','Sim-MAP');
title(['Bit error Rate for frame size = ',num2str(N)])
xlabel('SNR');
ylabel('BER');
```

```
% This function changes an integer number to binary for a given      % wordlength b
function binary = bin_num(a,b)
 state = zeros(1,b);  %Assign all zeros
 k = a;
 for i = 1:b
    if a >= 1 || a == 0
      state(1,b-i+1) = mod(a,2);
      a = floor(a/2);
    elseif a > 0 && a < 1
       k = k*2;
       if k < 1
          state(1,i) = 0;
       else
          state(1,i) = 1;
          l = k-1;
          k = l;
       end;
    end;
   end;

 binary = state;
```

```
% This function find the log(lx)
% lx is a matrix.
%fnmycorrection function calculates the correction value in log(exp(x)
%+exp(y)).
% Iteratively calculates the final result of %log(exp(x1)+exp(x2)+exp(x3)+ ....)
% x1,x2,x2 ... are elements of lx.
function y1 =findlog(lx)
```

```
 llenx = length(lx);
 cordelta = zeros(1,llenx);
 cordelta(1,1) = lx(1);
for i = 1:llenx-1
   cordelta(1,i+1) = max(lx(i+1),cordelta(1,i)) ...
              + fnmycorrection(lx(i+1),cordelta(1,i));
end
y1 = cordelta(1,llenx);
```

```
% This fuction computes the integer value from a binary number
% This function is used to encode the data.
function integer = intnum(a)
ele = length(a);
temp = 0;
for i = 1:ele
   temp = temp + a(i)*2^(ele-i);
end
integer = temp;
```

```
function correction = fnmycorrection(x,y)
%define the slope and intersection.
m0 = -0.3798;
c0 = 0.6931;
m115 = -0.2238;
c115 = 0.5371;
m152 = -0.1490;
c152 = 0.4249;

m2 = -0.0783;    %-(0.3133-0.1269); %-0.1864;
c2 =  0.2835;      %m2*(-1)+0.3133;  %0.4997
m3 = -0.0305;    %(0.1269-0.0489); %-0.078
c3 = 0.1401;      %   m3*(-2)+0.1269;  %0.2829


diff = abs(x-y); %Absolute value of x - y;
     % select the slope and intersection based on the abs of (x-y)
     if diff >= 0  && diff <=1
        correction = m0*diff + c0;

     elseif diff >1 && diff <= 1.5
```

```
        correction = m115*diff + c115;

    elseif diff > 1.5 && diff <= 2
        correction = m152*diff + c152;
    elseif diff >2 && diff <= 3
        correction = m2*diff + c2;

    elseif diff > 3 && diff <= 4
      correction =  m3*diff + c3;


    else
        correction = 0;

    end
```

```
% This function encodes the data for both encoders.
% First encoder is terminated.
% Second encoder is left open
% input1 is the randomly generated data.
% Alpha is the random interleaver .
function en_out  = myencode_bit(input1,g,alpha)
data = input1;
% Interleave the data for the second encoder
input2 = input1(alpha);
len_input = length(input1);
% Inital state is always at 1.
ini_state = 1;
[n,k] = size(g);
m = (k-1);
% Call the function stateout
[state0,state1] = stateout(g);

state_transition = ini_state;
% iteration for the input
for i = 1:len_input
  if(input1(i) == 0)  % if the input is 0
    temp_out(1,i) = state0(ini_state,3); %Parity from encoder1
    ini_state = state0(ini_state,2);
    state_transition(i+1) = ini_state;
  else  % if the input is 1
    temp_out(1,i) = state1(ini_state,3);
```

```
      ini_state = state1(ini_state,2);
       state_transition(i+1) = ini_state;
   end
end

% iteration for the termination
% Needs to find the data sequence for the termination
for j = 1:m
   %to encode the data, state should be in binary form
   % Call the function bin_num
   bin = bin_num(ini_state-1,m);
   dk = mod(bin*g(2,2:k)',2); % dk is 1 or 0;
   data(1,i+j) = dk; % Data is stored for the 2nd encoder
   if(dk == 0)
    temp_out(1,i+j) = state0(ini_state,3); %Parity from encoder1
    ini_state = state0(ini_state,2);
    state_transition(j+i+1) = ini_state;
   else
      temp_out(1,i+j) = state1(ini_state,3);
      ini_state = state1(ini_state,2);
      state_transition(j+i+1) = ini_state;
   end

end
temp_data = 2*data-1; % BPSK moudulation for original data
ini_state = 1;
state2_transition = ini_state;
for h = 1:len_input
  if(input2(h) == 0) % if the input is 0
   % the ouput is already moudulated
   temp_out(1,i+j+h) = state0(ini_state,3); %Parity from encoder2
   ini_state = state0(ini_state,2);
   state2_transition(h+1) = ini_state;
  else
   temp_out(1,i+j+h) = state1(ini_state,3);
   ini_state = state1(ini_state,2);
   state2_transition(h+1) = ini_state;
   end
end
for d = 1:m
%The same tail bits used to terminate encoder1 is used at encoder 2.

 if(data(1,i+d) == 0)
    temp_out(1,i+j+h+d) = state0(ini_state,3);
    ini_state = state0(ini_state,2);
     state2_transition(d+h+1) = ini_state;
```

```
else
   temp_out(1,i+j+h+d) = state1(ini_state,3);
   ini_state = state1(ini_state,2);
    state2_transition(d+h+1) = ini_state;
end

end
len_out = length(temp_out); %Length of the parity vector
%Extract data and create a matrix
%that holds the data, parity1, and parity2 in an order
%in order to emulate the actual transmission
en_out(1:3:3*(len_input+m)) = temp_data;
en_out(2:3:3*(len_input+m)) = temp_out(1:len_out/2);
en_out(3:3:3*(len_input+m)) = temp_out(len_out/2+1:lenp_out);
```

```
% This function computes the next state
% for the input 1 and 0
% and also computes the output and applies BPSK modulation
function [next_state0,next_state1] = stateout(g)
 [n,k] = size(g);
 m = k-1;     % number of memory elements
 % input is 0
 d_k = 0;
 for i = 1:2^m
    next_state0(i,1) = i;    % initial state
    bin = bin_num(i-1,k-1);  % Change the state into binary
    fd = mod(bin*g(2,2:k)',2);
    ak = xor(d_k,fd);
    fa(1,1) = ak;
    fa(2:k) = bin;
    % find the output due to the transition
    next_state0(i,3) = 2*mod(fa*g(1,:)',2)-1; %BPSK modulation
    bin(2:m) = bin(1:m-1);
    bin(1,1) = ak;
    next_state0(i,2) = intnum(bin)+1;  %next state
 end
 %input 1
 d_k = 1;
 for i = 1:2^m
    next_state1(i,1) = i;     %initial state
    bin = bin_num(i-1,k-1);    % Change the state into binary
    fd = mod(bin*g(2,2:k)',2);
    ak = xor(d_k,fd);
    fa(1,1) = ak;
```

```
    fa(2:k) = bin;
    % Find the output for the transition
    next_state1(i,3) = 2*mod(fa*g(1,:)',2)-1; %BPSK modulation
    bin(2:m) = bin(1:m-1);
    bin(1,1) = ak;
    next_state1(i,2) = intnum(bin)+1; % Next state in integer
  end
```

```
% This function extracts the received data and stores in a
% matrix with two rows one for decoder 1 and the other one
% is for the second decoder, Systematic data is interleaved.

function mydemux = mydemultiplex(datain,alpha,m)
data_len = length(datain);
temp = max(alpha);
newalpha = [alpha temp+1:temp+m];
tempdata = datain(1:3:data_len);

mydemux(1,1:2:2*data_len/3) = datain(1:3:data_len);
mydemux(1,2:2:2*data_len/3) = datain(2:3:data_len);
% Interleave the systematic data
mydemux(2,1:2:(2*data_len/3)) = tempdata(newalpha);
mydemux(2,2:2:2*data_len/3) = datain(3:3:data_len);
end
```

```
%Script for the hardware simulation
%Decoded data is stored in datatout_B_file
%and dataoutF_file using the Test bench
%created by the VHDL.
tempdataB = load('/home/vlsi/thanga2/dataoutB_file');
tempdataF = load('/home/vlsi/thanga2/dataoutF_file');
tempdatab = tempdataB';
tempdataf = tempdataF';
data1 = tempdatab(512:-1:1);
mydata = [data1 tempdataf];
oridata(newalphaD) = mydata;
% x is the data created randomly by Matlab.
HDLerrs = length(find(oridata(1:L_total-m)~=x))
```

A2. VHDL Codes for the implementation

```
--This module computes the log(exp(x)+exp(y)).
--Library and its packages are defined. Signed arithmetic is considered.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;



entity mylog104 is   -- entity name is mylog104
port( x,y     : in  std_logic_vector(9 downto 0) ;  --  two input definition; 10 bits
    logout  : out std_logic_vector(9 downto 0)  -- output 10 bits
    );
end entity mylog104;

architecture beha of mylog104 is
--component mult is generated using megafunction form Quartus.
-- Multiplier has two inputs with width 10bits and 8 bits, the output is 18 bits
component mult
        PORT
        ( dataa          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
          datab          : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
          result         : OUT STD_LOGIC_VECTOR (17 DOWNTO 0)
        );
end component;
--Slope and intersection are defined with 8 bits signed.
--ms(8,7) are slopes and cs(8,7) are intersection with signed representation.
Constant m0  : std_logic_vector(7 downto 0):="11000100";
constant m25 : std_logic_vector(7 downto 0):="11001100";
constant m50 : std_logic_vector(7 downto 0):="11010011";
constant m75 : std_logic_vector(7 downto 0):="11011010";
constant m2  : std_logic_vector(7 downto 0):="11101000";
constant m3  : std_logic_vector(7 downto 0):="11110110";
constant m4  : std_logic_vector(7 downto 0):="11111100";
constant m5  : std_logic_vector(7 downto 0):="00000000";
constant m6  : std_logic_vector(7 downto 0):="00000000";
constant c0  : std_logic_vector(7 downto 0):="01011001";
constant c25 : std_logic_vector(7 downto 0):="01010111";
constant c50 : std_logic_vector(7 downto 0):="01010011";
constant c75 : std_logic_vector(7 downto 0):="01001110";
constant c2  : std_logic_vector(7 downto 0):="01000000";
constant c3  : std_logic_vector(7 downto 0):="00100100";
constant c4  : std_logic_vector(7 downto 0):="00010010";
constant c5  : std_logic_vector(7 downto 0):="00001000";
constant c6  : std_logic_vector(7 downto 0):="00000100";
```

```
--module adder2 is defined. It consists a tempadder2 which is generated from
--megafunction. Overflow is saturated within adder 2 module.
Component adder2 is
 port (94ata : in std_logic_vector(9 downto 0);
      datab : in std_logic_vector(9 downto 0);
      dataout : out std_logic_vector(9 downto 0)
      );
end component;
--Intermediate signals are defined.
Signal maxy,tempout : std_logic_vector(9 downto 0);
signal tempm,tempc : std_logic_vector(7 downto 0);
signal tempout1,temp_out1 : std_logic_vector(17 downto 0);
signal tempout2 : std_logic_vector(17 downto 0);
signal tempdisp,tempdisp1,tempdisp2 : std_logic_vector(17 downto 0);
signal temp2 : std_logic_vector(10 downto 0);
signal display2 : std_logic_vector(9 downto 0);
signal tempabs,tempy : std_logic_vector(9 downto 0);
 begin  -- behavior of the architecture begins

 absprocess : process(tempabs,x,y)  -- process for calculating absolute value.
    Begin  -- absprocess
--if x and y are equal to -32 the abs is 0;
--tempabs is the output from the adder2(subtract)
--if tempabs is -32 then the absolute value is rounded to 31.75.

    if ( x = "1000000000" and y = "1000000000") then
     tempout <= "0000000000";
    else
    if tempabs(9) = '1' then
     if tempabs = "1000000000" then
      tempout <= "0111111111";
     else
     tempout <= (not tempabs) + 1;
     end if;
     else
      tempout <= tempabs;
      end if;
      end if;
    end process;  -- absprocess
--take the 2's compliment for input y so that the adder2 can be used to subtract
process(y)  --2's compliment process
 begin
  if y = "1000000000" then
   tempy <= "0111111111";
  else
   tempy <= (not y) + '1';
```

```
  end if;
end process;  --2's compliment process
--Intantiate adder2 to subtract
findabs : adder2 port map(x,tempy,tempabs);
 maxprocess :  process(x,y)  -- process for finding maximum of the two input x and y.
   begin  --maxprocess

    if(x > y) then
      maxy <= x;
    else
      maxy <= y;
    end if;


end process;  --maxprocess

--Select the slope and intersection based on the absolute difference of x and y.
--tempm is the slope and  tempc is the intersection.
Mux : process(tempout)
   begin  --mux process

   if(tempout >= "0000000000"  and tempout <= "0000000100") then
     tempm <= m0;
     tempc <= c0;
   elsif(tempout > "0000000100"  and tempout <= "0000001000") then
    tempm <= m25;
    tempc <= c25;

   elsif(tempout > "0000001000"  and tempout <= "0000001100") then
     tempm <= m50;
     tempc <= c50;

   elsif(tempout > "0000001100"  and tempout <= "0000010000" ) then
     tempm <= m75;
     tempc <= c75;
   elsif(tempout > "0000010000"  and tempout <= "0000100000" ) then
     tempm <= m2;
     tempc <= c2;
   elsif(tempout > "0000100000"  and tempout <= "0000110000"  ) then
     tempm <= m3;
     tempc <= c3;
   elsif(tempout > "0000110000"  and tempout <= "0001000000" ) then
     tempm <= m4;
     tempc <= c4;
   elsif(tempout > "0001000000"  and tempout <= "0001010000" ) then
     tempm <= m5;
```

```
      tempc <= c5;
   elsif(tempout > "0001010000"  and tempout <= "0001100000" ) then
      tempm <= m6;
      tempc <= c6;
   else
      tempm <= "00000000";
      tempc <= "00000000";

   end if;

end process mux; -- process mux
--Intantiate multiplier; inputs are absolute value(tempout) and the slope(tempm)
multiply : mult port map(tempout,tempm,temp_out1);
 --Add the intersection   Concatnate tempc according to fraction and integer place.
 tempout1 <= temp_out1 + ("000000" & tempc &"0000");
 --Add the maximum value of x and y  Concatnate maxy according to the sign and fraction
 tempdisp <= tempout1 + ('0'& maxy & "0000000") when maxy(9) = '0'  else
         tempout1 + ('1' & maxy & "0000000") ;

--Need to extract only 10 bits from 18 bits. Since the output is signed representation
--we need to take 2's compliment before extracting.
process(tempdisp)  --2's compliment for tempdisp = log(exp(x)+exp(y))
begin  --process
 if(tempdisp(17) = '1') then
  tempdisp1 <= (not tempdisp)+1;
 else
  tempdisp1 <= tempdisp;
 end if;
 end process;  --2's compliment for tempdisp
adjust : process(tempdisp1)  -- the value is rounded to the next highest number
    begin  --adjust process
      if (tempdisp1(6) = '1') then

        tempdisp2 <= tempdisp1 + "000000000010000000";

      else
       tempdisp2 <= tempdisp1;
       end if;
    end process adjust;
--10 bits are extracted and the sign is reassured.
temp2 <= tempdisp2(17 downto 7) when tempdisp(17) = '0'  else
       ((not tempdisp2(17 downto 7)) + 1);
 --Overflow is checked.
overflow : process(temp2)
        begin  -- over flow process
```

```
          case temp2(10 downto 9) is
            when  "00" => display2 <= temp2(9 downto 0) ;
            when  "01" => display2 <= "0111111111" ;
            when  "10" => display2 <= "1000000000" ;
            when  "11" => display2 <= temp2(9 downto 0) ;
           when others => null;
          end case;

    end process overflow;  -- end process

logout <= display2; -- Assign the output

end beha;       -- End the behavior of the architecture
```

```
-- Three input adder
-- Overflow is saturated based on the sign
library ieee;
use ieee.std_logic_1164.all;
--Entity definition, 3 inputs and an output with 10bits width
entity adder3 is
port ( dataa  : in std_logic_vector(9 downto 0);
     datab  : in std_logic_vector(9 downto 0);
     datac  : in std_logic_vector(9 downto 0);

     dataout  : out std_logic_vector(9 downto 0)
     );
end entity;
--Architecture description
architecture beha of adder3 is
-- component description
-- tempadder3 is a parallel adder generated form quartus megafunction
component tempadder3
          PORT
          (
                    data0x          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                    data1x          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                    data2x          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                    result          : OUT STD_LOGIC_VECTOR (11 DOWNTO 0)
          );
end component;
--Intermediate signals description
```

```
signal tempresult1 : std_logic_vector(11 downto 0);
signal overflow : std_logic;
signal tempout : std_logic_vector(9 downto 0);
begin  -- architecture behaviour

--Intantiate the tempadder3
--tempresult1 is the output from tempadder3
 add1 : tempadder3 port map(dataa,datab,datac,tempresult1);
 process(tempresult1)   --Overflow is detected
begin  --Overflow
  if ((tempresult1(11) = '0') and (tempresult1(10) = '0')   and (tempresult1(9) = '0')) or
((tempresult1(11) = '1')   and (tempresult1(10) = '1') and (tempresult1(9) = '1')) then
    overflow <= '0';
  else
    overflow <= '1';
  end if;
end process;  --Overflow
process(overflow,tempresult1)  --Overflow is saturated based on the sign.
Begin  -- Overflow saturataion
 if (overflow = '1' and tempresult1(11) = '1' ) then  --Overflow is detected and the sign is negative
    tempout <= "1000000000";  --minus 32
 elsif (overflow = '1' and tempresult1(11)  = '0') then  --Overflow is detected sign possitive
    tempout <= "0111111111"; --+31.75
  else
    tempout <= tempresult1(9 downto 0);  -- no overflow is detected
    end if;
end process;  -- overflow saturation
dataout <= tempout;   --output is assigned

end beha;  -- end the architecture for adder3
```

```
-- Two input adder for signed numbers
-- Overflow is saturated
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
--Entity definition for the adder2. Two inputs and an outpt with 10 bits width
entity adder2 is
  port (dataa : in std_logic_vector(9 downto 0);
        datab : in std_logic_vector(9 downto 0);
        dataout : out std_logic_vector(9 downto 0)
        );
end entity;
```

```
architecture beha of adder2 is
--Intermediate  signal definition.
signal tempout : std_logic_vector(9 downto 0);
signal over,sign    : std_logic;

--component description
--tempadder2 is generated by megafunction from Quartus.
component tempadder2
        PORT
        (
                dataa           : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                datab           : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                overflow        : OUT STD_LOGIC ;
                result          : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;

begin  --architecture
--Instantiate tempadder2
--inputs are dataa and datab, outputs are over and tempout.
--over is the overflow detector from tempadder2.
add : tempadder2 port map(dataa,datab,over,tempout);
sign <= dataa(9);  --sign of one of the input is chosen to saturate the output

process(tempout,over,sign)  --the output from the tempadder2 is saturated
 begin --process
        if (over = '1' and sign = '1') then  --overflow and negative number
          dataout <= "1000000000";      --negative 32
        elsif (over = '1' and sign = '0') then  --overflow and positive number
          dataout <= "0111111111";          --+31.75
        else
          dataout <= tempout;           --no overflow

         end if;
end process;
 end beha;  -- end behavior of the architecture
```

```
-- Four input adder
-- Overflow is saturated
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
--Entity definition; 4 inputs and an output
entity adder4 is
```

```
 port( datain1 : in std_logic_vector(9 downto 0);
     datain2 : in std_logic_vector(9 downto 0);
     datain3 : in std_logic_vector(9 downto 0);
     datain4 : in std_logic_vector(9 downto 0);
     dataout : out std_logic_vector(9 downto 0)
    );
end entity;
--Description of the architecture
architecture beha of adder4 is
-- Component description
-- tempadder4 is a parallel adder from quartus megafunction
--No overflow detection in this module
component tempadder4
        PORT
        (
                data0x          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                data1x          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                data2x          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                data3x          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                result          : OUT STD_LOGIC_VECTOR (11 DOWNTO 0)
        );
end component;
--Intermediate signals definition
signal tempresult1 : std_logic_vector(11 downto 0);
signal overflow : std_logic;
signal tempout   : std_logic_vector(9 downto 0);
begin  -- architecture behaviour
--Intantiate tempadder4, output is tempresult1
FINDTEMPRESULT1 : tempadder4 port map(datain1,datain2,datain3,datain4,tempresult1);
process(tempresult1)  -- overflow detection
begin  -- process
--two MSBs are considered to detect the overflow
 if ((tempresult1(11) = '0') and (tempresult1(10) = '0') and
  (tempresult1(9) = '0')) or ((tempresult1(11) = '1') and
  (tempresult1(10) = '1') and (tempresult1(9) = '1')) then
  overflow <= '0';
 else
  overflow <= '1';
 end if;
end process; -- end of overflow detection
process(overflow,tempresult1)  -- overflow is saturated
begin  -- process
 if (overflow = '1' and tempresult1(11) = '1' ) then  --overflow and negative output
  tempout <= "1000000000";
 elsif (overflow = '1' and tempresult1(11)  = '0') then  --overflow and positive output
  tempout <= "0111111111";
```

```
 else
  tempout <= tempresult1(9 downto 0);  -- no overflow
  end if;
end process;
dataout <= tempout;          -- Assign the output

end beha;   -- End of architecture for adder4
```

```
--This is the basic module for calculating forward metric from the branch matrix and previous
--forward metric. This module is used to calculate all the forward matrix in TotalAlphacore.
-- It has two inputs and one output.
-- It intantiate the "mylog104" module to calculate log(exp(x)+exp(y));
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

--Entity definition
--alpha1in and alpha2in are the previous alpha values
--gamma1 and gamma2 are the branch metrics associated with the alpha1in and alpha2in

entity AlphaCore is
 port ( alpha1in :  in std_logic_vector(9 downto 0);
      alpha2in :  in std_logic_vector(9 downto 0);
      gamma1   :  in std_logic_vector(9 downto 0);
      gamma2   :  in std_logic_vector(9 downto 0);
      alphaout : out std_logic_vector(9 downto 0)
       );
end entity;

architecture beha of AlphaCore is
--adder2 is defined
component adder2 is
 port (dataa : in std_logic_vector(9 downto 0);
     datab : in std_logic_vector(9 downto 0);
     dataout : out std_logic_vector(9 downto 0)
     );
end component; -- adder2
--mylog104 is defined
component mylog104 is
port( x,y     : in  std_logic_vector(9 downto 0) ;
    logout  : out std_logic_vector(9 downto 0)
   );
end component mylog104;
```

```
--Intermediate signals definition
signal tempalphagamma1, tempalphagamma2 : std_logic_vector(9 downto 0);
signal tempalpha : std_logic_vector(9 downto 0);
begin -- architecture
--Add alpha and gamma values
--Intantiate adder2 two times
Alphagamma1 : adder2 port map(alpha1in,gamma1,tempalphagamma1);
Alphagamma2 : adder2 port map(alpha2in,gamma2,tempalphagamma2);
 --Intantiate mylog104
calculateALPHA : mylog104 port map(tempalphagamma1,tempalphagamma2,tempalpha);
alphaout <= tempalpha;  --Assign the output from the mylog104
end beha;  --End architecture for AlphaCore
```

```
--This module consists of AlphaCore to calculate all alpha values for a particular
--generator polynomial. The inputs alpha and gamma must be assigned to the Alphacore
--based on the trellis diagram.
--Similar module should be implemented for beta(backward recursions).
--In our case 16 alpha values need to be calculated, 16 AlphaCore are intantiated.
--No normalization is done in this module
-- Output is 16x10 bus.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use work.AlphaType.all;  --AlphaType is a user defined package
--Entity definition
entity TotalAlphaCore is
 port ( alphain : in Alpha;  --16x10 bus defined in AlphaType package
      gammain : in Gamma;  --4x10 bus defined in AlphaType package
      Alphaout :out Alpha    --16x10 bus defined in AlphaType package
      );
end entity;

architecture beha of TotalAlphaCore is
component AlphaCore is  --Component declaration forAlphaCore
  port ( alpha1in :  in std_logic_vector(9 downto 0);
       alpha2in :  in std_logic_vector(9 downto 0);
       gamma1   :  in std_logic_vector(9 downto 0);
       gamma2   :  in std_logic_vector(9 downto 0);
       alphaout : out std_logic_vector(9 downto 0)
```

```
    );
end component;
--Intermediate signals definition.
signal tempalpha,tempout : Alpha ;
signal tempgamma : Gamma;
begin  --Architecure behaviour

 alphaout <= tempout;  -- output is assigned
  --temporary sigal assignments
--can be used directly
 tempAlpha <= alphain;
 tempgamma <= gammain;
 --Intantiate 16 AlphaCore modules and each output from the module is assigned to tempout
CALCULATEALPHA0 : Alphacore port map
        (tempalpha(0),tempalpha(1),tempgamma(0),tempgamma(3),tempout(0));
CALCULATEALPHA1 : Alphacore port map
         (tempalpha(2),tempalpha(3),tempgamma(2),tempgamma(1),tempout(1));
CALCULATEALPHA2 : Alphacore port map
         (tempalpha(5),tempalpha(4),tempgamma(1),tempgamma(2),tempout(2));
CALCULATEALPHA3 : Alphacore port map
         (tempalpha(6),tempalpha(7),tempgamma(0),tempgamma(3),tempout(3));
CALCULATEALPHA4 : Alphacore port map
         (tempalpha(9),tempalpha(8),tempgamma(1),tempgamma(2),tempout(4));
CALCULATEALPHA5 : Alphacore port map
        (tempalpha(10),tempalpha(11),tempgamma(0),tempgamma(3),tempout(5));
CALCULATEALPHA6 : Alphacore port map
        (tempalpha(12),tempalpha(13),tempgamma(0),tempgamma(3),tempout(6));
CALCULATEALPHA7 : Alphacore port map
        (tempalpha(15),tempalpha(14),tempgamma(1),tempgamma(2),tempout(7));
CALCULATEALPHA8 : Alphacore port map
         (tempalpha(1),tempalpha(0),tempgamma(0),tempgamma(3),tempout(8));
CALCULATEALPHA9 : Alphacore port map
         (tempalpha(2),tempalpha(3),tempgamma(1),tempgamma(2),tempout(9));
CALCULATEALPHAa : Alphacore port map
         (tempalpha(4),tempalpha(5),tempgamma(1),tempgamma(2),tempout(10));
CALCULATEALPHAb : Alphacore port map
         (tempalpha(7),tempalpha(6),tempgamma(0),tempgamma(3),tempout(11));
CALCULATEALPHAc : Alphacore port map
         (tempalpha(8),tempalpha(9),tempgamma(1),tempgamma(2),tempout(12));
CALCULATEALPHAd : Alphacore port map
        (tempalpha(11),tempalpha(10),tempgamma(0),tempgamma(3),tempout(13));
CALCULATEALPHAe : Alphacore port map
         (tempalpha(13),tempalpha(12),tempgamma(0),tempgamma(3),tempout(14));
CALCULATEALPHAf : Alphacore port map
         (tempalpha(14),tempalpha(15),tempgamma(1),tempgamma(2),tempout(15));
```

```
end beha; -- End the behavior of the architecture for TotalAlphacore
```

```
--This module calculates the normalization value used in FinalAlpha module.
--maxlog is used to calculate log(exp(x) + exp(y)); max(x,y) is considered.
--Tree architecture is used instead of recursive architecture to reduce the critical path delay
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;
use work.AlphaType.all;  --User defined package

entity total_gammaAlpha is  --Entity definition
 port  ( gammain : in Gamma;  --4x10 bus defined in AlphaType
           alphain : in Alpha;     --16x10 bus defined in AlphaType
           totalout : out std_logic_vector(9 downto 0)
                  );
end entity;

architecture beha of total_gammaAlpha is

component maxlog is  --component maxlog is declared
 port ( x,y   : in std_logic_vector(9 downto 0);

      logout : out std_logic_vector(9 downto 0)
     );
end component;
component adder2 is  --Adder2 declaration
 port (dataa : in std_logic_vector(9 downto 0);
     datab : in std_logic_vector(9 downto 0);
     dataout : out std_logic_vector(9 downto 0)
     );
end component;
--Intermediate signals definitions
signal tempgamma1,tempgamma2 : std_logic_vector(9 downto 0);
signal temp1,temp2,temp3,temp4 : std_logic_vector(9 downto 0);
signal temp5,temp6,temp7,temp8 : std_logic_vector(9 downto 0);
signal temp9,temp10,temp11,temp12 : std_logic_vector(9 downto 0);
signal temp13,temp14,temp15,temp16 : std_logic_vector(9 downto 0);
signal tempout : std_logic_vector(9 downto 0);

begin  -- architecture behavior
--The inputs are selected from the trellis diagram
--First branch calculation by intantiating 8 maxlog modules.
--12 means alpha(0) and alpha(1) are the imputs for maxlog module
```

```
FINDALPHA12      : maxlog port map(alphain(0),alphain(1),temp1);
FINDALPHA78      : maxlog port map(alphain(6),alphain(7),temp2);
FINDALPHA1112    : maxlog port map(alphain(10),alphain(11),temp3);
FINDALPHA1314    : maxlog port map(alphain(12),alphain(13),temp4);
FINDALPHA34      : maxlog port map(alphain(2),alphain(3),temp5);
FINDALPHA56      : maxlog port map(alphain(4),alphain(5),temp6);
FINDALPHA910     : maxlog port map(alphain(8),alphain(9),temp7);
FINDALPHA1516    : maxlog port map(alphain(14),alphain(15),temp8);
--Second branch calculation
--the inputs arethe ouputs from previous modules. 4 maxlog are intantiated.
--1278 means it gets the input from 12 and 78
FINDALPHA1278    : maxlog port map(temp1,temp2,temp9);
FINDALPHA11121314 : maxlog port map(temp3,temp4,temp10);
FINDALPHA3456    : maxlog port map(temp5,temp6,temp11);
FINDALPHA9101516 : maxlog port map(temp7,temp8,temp12);
--Third branch calculation
--FIRST gets the input from 1278 and 11121314
--TWO gets the input from 3456 and 9101516
FINDALPHAFIRST   : maxlog port map(temp9,temp10,temp13);
FINDALPHATWO     : maxlog port map(temp11,temp12,temp14);
FINDGAMMA1       : maxlog port map(gammain(0),gammain(3),tempgamma1);
FINDGAMMA2       : maxlog port map(gammain(1),gammain(2),tempgamma2);
ADDALPHAFIRSTGAMMA1: adder2 port map(temp13,tempgamma1,temp15);
ADDALPHATWOGAMMA2  : adder2 port map(temp14,tempgamma2,temp16);
FINDTOTAL        : maxlog port map(temp15,temp16,tempout);


totalout <= tempout;

end beha;     --End architecture.
```

```
--This module computes the normalized alpha values
--It instantiats TotalAlphaCore and TotalgammaAlpha
--Normalization in log domain is just subtracting the inputs.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use work.AlphaType.all;      -- User defined package
--Entity definition
entity FinalAlpha is
 port( enable   : in std_logic;
       alphain  : in Alpha;  --Alpha and Gamma  are defined in AlphaType
       gammain  : Gamma;
```

```
      Alphaout : out Alpha
    );
end entity;
--architectue of behavior of FinalAlpha
architecture beha of FinalAlpha is
component TotalAlphaCore is   --Component declaration  of TotalAlphaCore
 port ( alphain : in Alpha;
      gammain : in Gamma;
      Alphaout :out Alpha
      );
end component;
component adder2 is      --Component declaration of adder2
  port (dataa : in std_logic_vector(9 downto 0);
      datab : in std_logic_vector(9 downto 0);
      dataout : out std_logic_vector(9 downto 0)
      );
end component;
component total_gammaAlpha is    -- Component declaration of total_gammaAlpha
 port   ( gammain : in Gamma;
                    alphain : in Alpha;
                    totalout : out std_logic_vector(9 downto 0)
                   );
end component;
--Intermediate signals definition
signal tempalpha,temptotalAlpha : Alpha;
signal temptotal,temptotal1  : std_logic_vector(9 downto 0);
begin
--The output from the total_gammaAlpha is 2's complemented
--to use the adder2 as subtractor
 process(temptotal)  -- 2's complement process
 begin
  if temptotal = "1000000000" then
    temptotal1 <= "0111111111";
   else
  temptotal1 <= (not temptotal) + 1;
  end if;
end process;
 --Intantiate TotalAlphaCore to calculate the Alpha values without normalization
CALCULATEALPHA : TotalAlphaCore port map(alphain,gammain,tempalpha);
--Compute the normalization value to be subtracted, by intantiating total_gammaAlpha
CALCULTETOTAL  : total_gammaAlpha port map(gammain,alphain,temptotal);
--Intantiate 16 adder2 modules to normalize the output form TotalAlphaCore
CALTEMPALPHA0  : adder2 port map(tempalpha(0),temptotal1,temptotalAlpha(0));
CALTEMPALPHA1  : adder2 port map(tempalpha(1),temptotal1,temptotalAlpha(1));
CALTEMPALPHA2  : adder2 port map(tempalpha(2),temptotal1,temptotalAlpha(2));
CALTEMPALPHA3  : adder2 port map(tempalpha(3),temptotal1,temptotalAlpha(3));
```

```
CALTEMPALPHA4  : adder2 port map(tempalpha(4),temptotal1,temptotalAlpha(4));
CALTEMPALPHA5  : adder2 port map(tempalpha(5),temptotal1,temptotalAlpha(5));
CALTEMPALPHA6  : adder2 port map(tempalpha(6),temptotal1,temptotalAlpha(6));
CALTEMPALPHA7  : adder2 port map(tempalpha(7),temptotal1,temptotalAlpha(7));
CALTEMPALPHA8  : adder2 port map(tempalpha(8),temptotal1,temptotalAlpha(8));
CALTEMPALPHA9  : adder2 port map(tempalpha(9),temptotal1,temptotalAlpha(9));
CALTEMPALPHA10  : adder2 port map(tempalpha(10),temptotal1,temptotalAlpha(10));
CALTEMPALPHA11  : adder2 port map(tempalpha(11),temptotal1,temptotalAlpha(11));
CALTEMPALPHA12  : adder2 port map(tempalpha(12),temptotal1,temptotalAlpha(12));
CALTEMPALPHA13  : adder2 port map(tempalpha(13),temptotal1,temptotalAlpha(13));
CALTEMPALPHA14  : adder2 port map(tempalpha(14),temptotal1,temptotalAlpha(14));
CALTEMPALPHA15  : adder2 port map(tempalpha(15),temptotal1,temptotalAlpha(15));
--When there is a change in temptotalAlpha check the enable is one
--Enable is from the control module, the output is available when enabe is 1
process(enable,temptotalAlpha)
begin
if enable = '1' then
Alphaout <= temptotalAlpha;

end if;
end process;

end beha;  --End the behavior of FinalAlpha
```

```
--This module computes the branch matrix from the systematic data, parity data, and
--extrinsic values.
--mylog104 is used to calculate log(exp(x)+exp(y))
--Gammaout contains gamma00(0/0; input 0 and output 0), gamma01(0/1),gamma10(1/0),
--gamma11(1/1)
--The equations are derived from the LOG-MAP algorithm.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;
use work.AlphaType.all;       --User defined package
--Entity definition
entity gammaF is
 port  (enable   : in std_logic;
       datain   : in std_logic_vector(9 downto 0);  --Systematic data
        parityin : in std_logic_vector(9 downto 0);  --parity data
        L_eadd   : in std_logic_vector(9 downto 0);  --extrinsic value
        gammaout  : out Gamma                        --4x10 bus

       );
```

```
end entity;

architecture beha of gammaF is
--Intermediate signal definitions
 signal tempLe    : std_logic_vector(9 downto 0);
 signal gamma00   : std_logic_vector(9 downto 0);
 signal gamma01   : std_logic_vector(9 downto 0);
 signal gamma10   : std_logic_vector(9 downto 0);
 signal gamma11   : std_logic_vector(9 downto 0);
 signal tempdatain : std_logic_vector(9 downto 0);
 signal tempparityin : std_logic_vector(9 downto 0);
 signal tempL_E   : std_logic_vector(9 downto 0);
component mylog104 is     --Component mylog104 is declared
port( x,y     : in  std_logic_vector(9 downto 0) ;
      logout  : out std_logic_vector(9 downto 0)
     );
end component mylog104;
component adder3 is    --Component adder3 declaration
port ( dataa  : in std_logic_vector(9 downto 0);
      datab  : in std_logic_vector(9 downto 0);
      datac  : in std_logic_vector(9 downto 0);
      dataout : out std_logic_vector(9 downto 0)
      );
end component;
component adder4 is  --Component adder4 declaration
  port( datain1 : in std_logic_vector(9 downto 0);
       datain2 : in std_logic_vector(9 downto 0);
       datain3 : in std_logic_vector(9 downto 0);
       datain4 : in std_logic_vector(9 downto 0);
       dataout : out std_logic_vector(9 downto 0)
      );
end component;
begin
--2's complement
process(datain)
 begin
   if datain = "1000000000" then
    tempdatain <= "0111111111";
    else
     tempdatain <= (not datain)+1;
   end if;
end process;
process(parityin)  --2's complement
 begin
  if parityin = "1000000000" then
    tempparityin <= "0111111111";
```

```
  else
   tempparityin <= (not parityin) + 1;
  end if;
end process;
process(tempLe)  --2's complement
 begin
  if tempLe = "1000000000" then
   tempL_E <= "0111111111";
  else
   tempL_E <= (not tempLe) + 1;
  end if;
 end process;
--Calculate the state transition probability from the extrinsic value
--Intantiate mylog104 module
calculateL_e : mylog104 port map("0000000000",L_eadd,tempLe);
--The equations are derived under the Gamma Module in this thesis
--Gamma00 calculation
CALCULATEGAMMA00 : adder3 port map(tempdatain,tempparityin,tempL_E,gamma00);
--Gamma01 calculation
CALCULATEGAMMA01 : adder3 port map(tempdatain,parityin,tempL_E,gamma01);
--Gamma10 calculation
CALCULATEGAMMA10 : adder4 port map(datain,tempparityin,tempL_E,L_eadd,gamma10);
--Gamma11 calculation
CALCULATEGAMMA11 : adder4 port map(datain,parityin,tempL_E,L_eadd,gamma11);
--When the enable is 1 then the output is changed
--Output remains until enable is changed to 1, Latch operation
process(enable,gamma00,gamma01,gamma10,gamma11)
 begin
  if enable = '1' then
          gammaout(0) <= gamma00;
          gammaout(1) <= gamma01;
          gammaout(2) <= gamma10;
          gammaout(3) <= gamma11;

  end if;
end process;
 end beha;  --End the behavior of GammaF
```

```
--This module is used to calculate the LLR
--The inputs are Alpha(16x10 bus), Beta(16x10 bus), and Gamma(4x10 bus)
--maxlog is used to calculate log(exp(x)+exp(y)) to reduce the critical path delay.
--Tree architecture is used to calculate the LLR
--Inputs for the adder3 must be derived from the trellis diagram.
--Different generator polynomial produce different input sequence for adder3.
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;
use work.AlphaType.all;  --user defined package
--Entity definiton
entity LLRCore is
 port ( enable  : in std_logic;
       alphain : in Alpha;
       betain  : in Alpha;
       gammain : in Gamma;

       LLRout  : out std_logic_vector(9 downto 0)
      );
end entity;

architecture beha of LLRCore is
component adder2 is  --adder2 declaration
 port (dataa : in std_logic_vector(9 downto 0);
      datab : in std_logic_vector(9 downto 0);
      dataout : out std_logic_vector(9 downto 0)
      );
end component;

 component adder3 is  --adder3 declaration
port ( dataa  : in std_logic_vector(9 downto 0);
     datab  : in std_logic_vector(9 downto 0);
     datac  : in std_logic_vector(9 downto 0);
     dataout : out std_logic_vector(9 downto 0)
     );
end component;
component maxlog is  --maxlog declaration
 port(-- clk    : in std_logic;
    x,y     : in  std_logic_vector(9 downto 0) ;
    logout  : out std_logic_vector(9 downto 0)
    );
end component maxlog;
--Intermediate signal definitions for input 1
signal temp0,temp1,temp2,temp3    : std_logic_vector(9 downto 0);
signal temp4,temp5,temp6, temp7   : std_logic_vector(9 downto 0);
signal temp8,temp9,temp10,temp11  : std_logic_vector(9 downto 0);
signal temp12,temp13,temp14,temp15 : std_logic_vector(9 downto 0);
signal templog0,templog1,templog2  : std_logic_vector(9 downto 0);
signal templog3,templog4,templog5,templog6,templog7 : std_logic_vector(9 downto 0);
signal templog20,templog21,templog22,templog23 : std_logic_vector(9 downto 0);
signal templog30,templog31                : std_logic_vector(9 downto 0);
```

```
--Intermediate signal definitions for input 0
signal Ztemp0,Ztemp1,Ztemp2,Ztemp3   : std_logic_vector(9 downto 0);
signal Ztemp4,Ztemp5,Ztemp6, Ztemp7  : std_logic_vector(9 downto 0);
signal Ztemp8,Ztemp9,Ztemp10,Ztemp11  : std_logic_vector(9 downto 0);
signal Ztemp12,Ztemp13,Ztemp14,Ztemp15 : std_logic_vector(9 downto 0);
signal Ztemplog0,Ztemplog1,Ztemplog2,Ztemplog3 : std_logic_vector(9 downto 0);
signal Ztemplog4,Ztemplog5,Ztemplog6,Ztemplog7 : std_logic_vector(9 downto 0);
signal Ztemplog20,Ztemplog21,Ztemplog22,Ztemplog23 : std_logic_vector(9 downto 0);
signal Ztemplog30,Ztemplog31              : std_logic_vector(9 downto 0);
signal tempLLR1,tempLLR0                  : std_logic_vector(9 downto 0);
signal twosLLR0,tempLLRout                : std_logic_vector(9 downto 0);

begin
--When enable is one output is changed; latch
process(enable,tempLLRout)
begin
   if (enable = '1') then
    LLRout <= tempLLRout;
   end if;
end process;



--LLR is calculated by subtracting the probability due to input 0 from probability due to input 1
--Probability due to input 0 is 2's complemented.
process(tempLLR0)  --2's complement
 begin
  if tempLLR0 = "1000000000" then
   twosLLR0 <= "0111111111";
  else
   twosLLR0 <= (not tempLLR0) + 1;
  end if;
 end process;
--adder2 is intantiated to calculate the final LLR
FINDLLROUT : adder2 port map(tempLLR1,twosLLR0,tempLLRout);
--First branch calculations for input 1, just adding the inputs to feed them to maxlog.
--16 adder3 modules are instantiated for input 1.
 FINDTEMP0 : adder3 port map(gammain(3),alphain(0),betain(8),temp0);
 FINDtemp1 : adder3 port map( gammain(3),alphain(1),betain(0),temp1);
 FINDtemp2  : adder3 port map( gammain(2),alphain(2),betain(1),temp2);
 FINDtemp3  : adder3 port map( gammain(2), alphain(3), betain(9),temp3);
 FINDtemp4  : adder3 port map( gammain(2), alphain(4), betain(2),temp4);
 FINDtemp5  : adder3 port map( gammain(2),alphain(5), betain(10),temp5);
 FINDtemp6  : adder3 port map( gammain(3),alphain(6), betain(11),temp6);
 FINDtemp7  : adder3 port map( gammain(3),alphain(7) , betain(3),temp7);
 FINDtemp8  : adder3 port map( gammain(2) , alphain(8), betain(4),temp8);
```

```
FINDtemp9  : adder3 port map( gammain(2) , alphain(9), betain(12),temp9);
FINDtemp10 : adder3 port map( gammain(3) , alphain(10) , betain(13),temp10);
FINDtemp11 : adder3 port map( gammain(3) , alphain(11), betain(5),temp11);
FINDtemp12 : adder3 port map( gammain(3) , alphain(12), betain(14),temp12);
FINDtemp13 : adder3 port map( gammain(3) , alphain(13),betain(6),temp13);
FINDtemp14 : adder3 port map( gammain(2) , alphain(14), betain(7),temp14);
FINDtemp15 : adder3 port map( gammain(2) , alphain(15) , betain(15),temp15);

--First branch calculations for input 0, just adding the inputs to feed them to maxlog.
--16 adder3 modules are instantiated for input 0.

FINDZtemp0 : adder3 port map( gammain(0) , alphain(0) , betain(0),Ztemp0);
FINDZtemp1  : adder3 port map( gammain(0) , alphain(1) , betain(8),Ztemp1);
FINDZtemp2  : adder3 port map( gammain(1) , alphain(2) , betain(9),Ztemp2);
FINDZtemp3  : adder3 port map( gammain(1) , alphain(3) , betain(1),Ztemp3);
FINDZtemp4  : adder3 port map( gammain(1) , alphain(4) , betain(10),Ztemp4);
FINDZtemp5  : adder3 port map( gammain(1) , alphain(5) , betain(2),Ztemp5);
FINDZtemp6  : adder3 port map( gammain(0) , alphain(6) , betain(3),Ztemp6);
FINDZtemp7  : adder3 port map( gammain(0) , alphain(7) , betain(11),Ztemp7);
FINDZtemp8  : adder3 port map( gammain(1) , alphain(8) , betain(12),Ztemp8);
FINDZtemp9  : adder3 port map( gammain(1) , alphain(9) , betain(4),Ztemp9);
FINDZtemp10 : adder3 port map( gammain(0) , alphain(10) , betain(5),Ztemp10);
FINDZtemp11 : adder3 port map( gammain(0) , alphain(11) , betain(13),Ztemp11);
FINDZtemp12 : adder3 port map( gammain(0) , alphain(12) , betain(6),Ztemp12);
FINDZtemp13 : adder3 port map( gammain(0) , alphain(13) , betain(14),Ztemp13);
FINDZtemp14 : adder3 port map( gammain(1) , alphain(14) , betain(15),Ztemp14);
FINDZtemp15 : adder3 port map( gammain(1) , alphain(15) , betain(7),Ztemp15);

--CALCULATE INTERMIDIATE LOG VALUES FOR THE INPUT 1
--First branch modules(adder3) provide input to maxlog modules
 --Second branch calculations for input 1
--8 maxlog modules are intantiated

CALCULATETEMPLOG0 : maxlog port map(temp0,temp1,templog0);
CALCULATETEMPLOG1 : maxlog port map(temp2,temp3,templog1);
CALCULATETEMPLOG2 : maxlog port map(temp4,temp5,templog2);
CALCULATETEMPLOG3 : maxlog port map(temp6,temp7,templog3);
CALCULATETEMPLOG4 : maxlog port map(temp8,temp9,templog4);
CALCULATETEMPLOG5 : maxlog port map(temp10,temp11,templog5);
CALCULATETEMPLOG6 : maxlog port map(temp12,temp13,templog6);
CALCULATETEMPLOG7 : maxlog port map(temp14,temp15,templog7);
--Third branch calculation for input 1
--Second branch modules provide the input signals
--4 maxlog modules are intantiated, inputs are from the previous maxlog modules
CALCULATETEMPLOG20 : maxlog port map(templog0,templog1,templog20);
CALCULATETEMPLOG21 : maxlog port map(templog2,templog3,templog21);
```

```
CALCULATETEMPLOG22 : maxlog port map(templog4,templog5,templog22);
CALCULATETEMPLOG23 : maxlog port map(templog6,templog7,templog23);
--Forth branch calculation for input 1
--Third branch modules provide the input signals
--2 maxlog modules are intantiated, inputs are from the previous maxlog modules

CALCULATETEMPLOG30 : maxlog port map(templog20,templog21,templog30);
CALCULATETEMPLOG31 : maxlog port map(templog22,templog23,templog31);
--Probability for input 1
CALCULATETEMPLLR1 : maxlog port map(templog30,templog31,tempLLR1);
--CALCULTATE THE INTERMIDIAE LOG VALUES FOR INPUT 0
--First branch modules(adder3 for input 0) provide input to maxlog modules
 --Second branch calculations for input 0
--8 maxlog modules are intantiated

CALCULATEZTEMPLOG0 : maxlog port map(Ztemp0,Ztemp1,Ztemplog0);
CALCULATEZTEMPLOG1 : maxlog port map(Ztemp2,Ztemp3,Ztemplog1);
CALCULATEZTEMPLOG2 : maxlog port map(Ztemp4,Ztemp5,Ztemplog2);
CALCULATEZTEMPLOG3 : maxlog port map(Ztemp6,Ztemp7,Ztemplog3);
CALCULATEZTEMPLOG4 : maxlog port map(Ztemp8,Ztemp9,Ztemplog4);
CALCULATEZTEMPLOG5 : maxlog port map(Ztemp10,Ztemp11,Ztemplog5);
CALCULATEZTEMPLOG6 : maxlog port map(Ztemp12,Ztemp13,Ztemplog6);
CALCULATEZTEMPLOG7 : maxlog port map(Ztemp14,Ztemp15,Ztemplog7);
--Third branch calculation for input 0
--Second branch modules provide the input signals
--4 maxlog modules are intantiated, inputs are from the previous maxlog modules

CALCULATEZTEMPLOG20 : maxlog port map(Ztemplog0,Ztemplog1,Ztemplog20);
CALCULATEZTEMPLOG21 : maxlog port map(Ztemplog2,Ztemplog3,Ztemplog21);
CALCULATEZTEMPLOG22 : maxlog port map(Ztemplog4,Ztemplog5,Ztemplog22);
CALCULATEZTEMPLOG23 : maxlog port map(Ztemplog6,Ztemplog7,Ztemplog23);
--Forth branch calculation for input 0
--Third branch modules provide the input signals
--2 maxlog modules are intantiated, inputs are from the previous maxlog modules

CALCULATEZTEMPLOG30 : maxlog port map(Ztemplog20,Ztemplog21,Ztemplog30);
CALCULATEZTEMPLOG31 : maxlog port map(Ztemplog22,Ztemplog23,Ztemplog31);
--probability for input 0
CALCULATETEMPLLR0 : maxlog port map(Ztemplog30,Ztemplog31,tempLLR0);

END beha;       --End the behavior for LLR
```

```
--This module calculate extrinsic values for each data.
--The inputs are the output from LLRmodule, datain form datamem, and LEin from LEmem
--To calculate LEout, two times of datain and LEin must be subtracted from LLRin
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
--Entity definition
entity LEModule is
 port ( enable  : in std_logic;  --Signal form control module
      LLRin   : in std_logic_vector(9 downto 0);
      datain  : in std_logic_vector(9 downto 0);
      LEin    : in std_logic_vector(9 downto 0);
      LEout   : out std_logic_vector(9 downto 0)
      );
end entity;

architecture beha of LEModule is
--Component description
component adder4 is
 port( datain1 : in std_logic_vector(9 downto 0);
      datain2 : in std_logic_vector(9 downto 0);
      datain3 : in std_logic_vector(9 downto 0);
      datain4 : in std_logic_vector(9 downto 0);
      dataout : out std_logic_vector(9 downto 0)
     );
end component;
signal tempLEout,tempdatain,tempLEin : std_logic_vector(9 downto 0);
begin
 process(enable,tempLEout)  --process to assign the output when enable is 1
 begin
 if enable = '1' then
  LEout <= tempLEout;
 end if;
 end process;
process(datain)  --2's complement for datain
begin
 if datain = "1000000000" then
  tempdatain <= "0111111111";
 else
  tempdatain <= (not datain) + 1;
 end if;
end process;
process(LEin)  --2's complement for LEin
begin
 if LEin = "1000000000" then
  tempLEin <= "0111111111";
 else
  tempLEin  <= (not LEin) + 1;
 end if;
```

```
end process;
 --Instantiate adder4 to subtract datain and LEin from LLRin
FINDLE : adder4 port map(LLRin,tempdatain,tempdatain,tempLEin,tempLEout);
end beha;
```

```
--This module provides all the control signal and the addresss needed for data, parity
--and extrinsic values.
--Flow chart can be found in this thesis.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity control is
 port( clk       : in std_logic;                                  --Clock signal input
     start     : in std_logic;                                    --Start the decoding process
     countup         : out std_logic_vector(9 downto 0);  -- counter values for forward direction
     countdown       : out std_logic_vector(9 downto 0);  --counter values for backware direction
     dataLEaddressF   : out std_logic_vector(9 downto 0);  --Address for LE forward
     dataLEaddressB   : out std_logic_vector(9 downto 0); --Address for LE backward
     parityaddressF   : out std_logic_vector(9 downto 0);  --Parity address for forward
     parityaddressB   : out std_logic_vector(9 downto 0); --parity address for backward
     alphaAddress     : out std_logic_vector(9 downto 0);  --Address for alpha
     betaAddress      : out std_logic_vector(9 downto 0);  --Address for Beta
     gammaAddressF      : out std_logic_vector(9 downto 0);  --Gamma address for forward
     gammaAddressB      : out std_logic_vector(9 downto 0); --Gamma address for backward
     datamemenable : out std_logic;    --memory enable for datamem
     parity1enable : out std_logic;       --memory enable for parity1mem
     parity2enable : out std_logic;      -- memory enable for parity2mem
     LEmemenable     : out std_logic;   --memory enable for LEmem
     datawr       : out std_logic;           --Read/write enable for datamem
     parity1wr    : out std_logic;          --Read/write enable for parity1mem
     parity2wr    : out std_logic;          --Read/write enable for parity2mem
     LEwr        : out std_logic;           --Read/write enable for LEmem
     alphamemenable :out std_logic;   --memory enable for Alphamem
     betamemenable  : out std_logic;   --memory enable for betamem
     gammamemenable : out std_logic;   --memory enable for gamma memory
     alphawr      : out std_logic;            --Read/write enable for Alphamem
     betawr       : out std_logic;           --Read/write enable for betamem
     gammawr      : out std_logic;          --Read/write enable for gammamem
     alphaenable   : out std_logic;         --Enable for FinalAlpha module
     betaenable    : out std_logic;        --Enable for FinalBeta module
     gammaenable   : out std_logic;       --Enable for GammaF module
     LLRenable     : out std_logic;         --Enable for LLR module
```

```
    LEenable     : out std_logic;        --Enable for LEmodule
    Ld_Areg      : out std_logic;        --Load signal for Alpha register
    Ld_Breg      : out std_logic;        --Load signal for Beta register
    decode       : out std_logic;        -- decode  is set to one at the end of the iteration
    decoder      : out std_logic         -- Used to switch the decoder 1 to decoder 2
  );
end entity;

architecture beha of control is
--Define an array type to hold the interleaver address
type inter is array (0 to 1023) of std_logic_vector(9 downto 0);
-- INTERLEAVER ADDRESSES ARE DEFINED AS CONSTATANTS.
 constant  interleaver :inter :=  ("1101011100", -- more values are defined

);
signal  sig_countup  : std_logic_vector(9 downto 0) := "0000000000"; --initalized to zero
signal  sig_countdown : std_logic_vector(9 downto 0) := "1111111111";  --initialized to 1023
signal  sig_decoder        : std_logic := '0'; --DECODER IS INITIALIZED TO ZERO =>decoder 1
type  state_type is (IDLE,STA0,STA1,STA2,STA3);  --State transition for the control module
signal state,next_state : state_type;
signal iter : std_logic_vector(2 downto 0) := "100"; --NUMBER OF ITERATIONS
--
signal     sig_dataLEaddressF  :  std_logic_vector(9 downto 0):= "0000000000";
signal     sig_dataLEaddressB  :  std_logic_vector(9 downto 0):= "1111111111";
signal     sig_parityaddressF  :  std_logic_vector(9 downto 0);
signal     sig_parityaddressB  : std_logic_vector(9 downto 0);
signal     sig_alphaAddress    : std_logic_vector(9 downto 0);
signal     sig_betaAddress     : std_logic_vector(9 downto 0);
signal     sig_gammaAddressF   : std_logic_vector(9 downto 0) ;
signal     sig_gammaAddressB   : std_logic_vector(9 downto 0);
signal     sig_datamemenable :  std_logic := '1';
signal     sig_parity1enable :  std_logic :='1' ;
signal     sig_parity2enable :  std_logic := '0';
signal     sig_LEmemenable     :  std_logic := '1';
signal     sig_datawr       :  std_logic := '0' ;
signal     sig_parity1wr    :  std_logic := '0' ;
signal     sig_parity2wr    :  std_logic := '0';
signal     sig_LEwr         :  std_logic := '0' ;
signal     sig_alphamemenable : std_logic := '1';
signal     sig_betamemenable  :  std_logic := '1';
signal     sig_gammamemenable :  std_logic := '1';
signal     sig_alphawr      :  std_logic := '1';
signal     sig_betawr       :  std_logic := '1';
signal     sig_gammawr      :  std_logic := '1';
signal     sig_alphaenable   :  std_logic; --ALPHA MODULE
signal     sig_betaenable    :  std_logic; --BETA MODULE
```

```
signal     sig_gammaenable   : std_logic;--GAMMA MODULE
signal     sig_LLRenable     : std_logic; --LLR MODULE
signal     sig_LEenable      : std_logic; -- LE MODULE
signal   sig_Ld_Areg       : std_logic := '0';
signal     sig_Ld_Breg       : std_logic := '0';
signal     sig_decode        : std_logic := '1';
begin
--BASED ON THE DECODER PARITY ENALBLE IS SECLECTED

PARITYSELECT:  process(sig_decoder)
   begin
     if (sig_decoder = '0') then
      sig_parity1enable <= '1';
      sig_parity2enable <= '0';
     else
      sig_parity1enable <= '0';
      sig_parity2enable <= '1';
     end if;
   end process;

         process(sig_countup)
  -- DATA ADDRESSES ARE ASSIGNED ACCORDING TO THE DECODER
         begin
         --if (clk'event and clk = '1') then
   if sig_decoder = '0' then  --decoder 1

         sig_dataLEaddressF  <= sig_countup;
         sig_dataLEaddressB  <= sig_countdown;
        else       --decoder 2, interleaved addresses
         sig_dataLEaddressF  <= interleaver(to_integer(unsigned(sig_countup)));
         sig_dataLEaddressB  <= interleaver(to_integer(unsigned(sig_countdown)));
       end if;
  --ONCE REACHED THE MIDDELE OF THE FRAME
  -- ALPHA AND BETA ADDRESSES ARE SWITCHED TO READ FROM THE MEMORY
   if sig_countup <= "0111111111" then  --written to the alpha and beta memory
     sig_alphaaddress <= sig_countup;
     sig_betaaddress  <= sig_countdown;
   else
     sig_alphaaddress <= sig_countdown;  --read from alpha and beta memory
     sig_betaaddress  <= sig_countup;
    end if;
    --end if;
  end process;
process(clk)   --State transitions for the control module
begin
  if(clk'event and clk = '1') then  --at the rising edge of the clock
```

```
    state <= next_state;
  end if;
end process;
process(state)  --State transitions for the control module
  begin
   case state is
     when IDLE =>
      if (sig_decoder = '1') then  --if the decoding process is already started
         next_state <= STA0;
      elsif(sig_decoder = '0' and (start = '1' or sig_decode = '0')) then  --if start is active high
         next_state <= STA0;
      else      --Either decoding not started or start is active low
         next_state <= IDLE;    --remains until start is active high
      end if;
     when STA0 =>  --decoding is process

       next_state <= STA1;  --go to next state

     when STA1 =>
       if sig_countup <= "0111111111" then   --counter not reached the middle of the frame
         next_state <= STA0;   --go to STA0
       else
       next_state <= STA2;      --go to STA2; LLR calculation
      end if;
     when STA2 =>
        if sig_countup = "1111111111" then  --counter reached end of the frame
         next_state <= IDLE;
      else                                          --if not go to STA3
         next_state <= STA3;
       end if;

     when STA3 =>

         next_state <= STA2;

    end case;
end process;      --state transitions for control module ends
process(clk)      --Control Signal assignments  based on the state
   begin
  if (clk'event and clk = '1') then
    case state is
      when IDLE =>
       sig_alphamemenable  <= '1';
       sig_betamemenable   <= '1';
       sig_alphawr         <= '1';
       sig_betawr          <= '1';
```

```
    sig_gammamemenable  <= '1';
    sig_gammawr         <= '1';
    sig_gammaenable     <= '1';
    sig_alphaenable     <= '0';
    sig_betaenable      <= '0';
    sig_LLRenable       <= '0';
    sig_LEenable        <= '0';
    sig_Ld_Areg         <= '1';
    sig_Ld_Breg         <= '1';
    sig_decode          <= '0';

  when STA0 =>
   if sig_countup < "0111111111" then
   sig_gammamemenable  <= '0';
   sig_alphamemenable  <= '0';
   sig_betamemenable   <= '0';
   else
    sig_gammamemenable  <= '1';
   sig_alphamemenable  <= '1';
   sig_betamemenable   <= '1';
   sig_lewr            <= '0';
   end if;
   sig_gammawr         <= '0';
   sig_alphawr         <= '0';
   sig_betawr          <= '0';
   sig_alphaenable     <= '1';
   sig_betaenable      <= '1';
   sig_gammaenable     <= '0';
   sig_Ld_Areg         <= '0';
   sig_Ld_Breg         <= '0';

   sig_countup      <= sig_countup + 1;
   sig_countdown    <= sig_countdown - 1;
  when  STA1 =>

    if sig_countup <= "0111111111" then
     sig_gammamemenable  <= '1';
     sig_gammawr         <= '1';
     sig_betamemenable <= '1';
     sig_alphamemenable <= '1';
     sig_Ld_Areg         <= '1';
     sig_Ld_Breg         <= '1';
     sig_gammawr         <= '1';
     sig_alphawr         <= '1';
     sig_betawr          <= '1';
     sig_alphaenable     <= '0';
```

```
        sig_betaenable    <= '0';
        sig_gammaenable   <= '1';


       else
        sig_gammamemenable <= '0';
        sig_alphamemenable <= '0';
        sig_betamemenable  <= '0';
        sig_Ld_Areg       <= '1';
        sig_Ld_Breg       <= '1';
        sig_gammaenable   <= '0';
        sig_gammawr       <= '0';
        sig_alphawr       <= '0';
        sig_betawr        <= '0';
        sig_alphaenable    <= '0';
        sig_betaenable     <= '0';
        sig_llrenable      <= '1';
        sig_leenable       <= '1';
        sig_lewr          <= '1';
--
       end if;
       --ready to decode?
       if (iter = "000" and sig_decoder = '1' and
        sig_countup = "1000000000")then
        sig_decode <= '1';

        end if;
       if(iter = "000" and sig_decoder = '1') then
         sig_leenable <= '0';  --ready to decode? No need to calculate extrinsic values
       end if;

     when STA2 =>
       sig_gammamemenable <= '1';
        sig_alphamemenable <= '1';
        sig_betamemenable  <= '1';
      sig_alphaenable   <= '1';
      sig_betaenable    <= '1';
      sig_LLRenable     <= '0';
      sig_LEenable      <= '0';
      sig_LEwr         <= '0';
      sig_Ld_Areg       <= '0';
      sig_Ld_Breg       <= '0';
      sig_countup    <= sig_countup + 1;
      sig_countdown  <= sig_countdown - 1;
      if (sig_countup = "1111111111") then  --at the end of the frame
       sig_decoder <= not sig_decoder;       --Switch the decoder
```

```
        end if;

        --Iteration is reduced by one when the second decoder is reached the end of the frame
        if (sig_decoder = '1' and sig_countup = "1111111111") then
        iter <= iter - 1;
        end if;

      when STA3 =>
        sig_gammamemenable <= '0';
         sig_alphamemenable <= '0';
         sig_betamemenable <= '0';
        sig_LLRenable <= '1';
        sig_alphaenable <= '0';
        sig_betaenable <= '0';
         if (iter = "000" and sig_decoder = '1') then
         sig_LEenable <= '0';
         else
        sig_LEenable <= '1';
        end if;

        sig_LEwr <= '1';
        sig_Ld_Areg <= '1';
        sig_Ld_Breg <= '1';

     end case;
     end if;
  end process; --Signal assignment process
--Assign all temporary signal values to the output of the Control module
     countup <= sig_countup;
     countdown <= sig_countdown;
     dataLEaddressF <= sig_dataLEaddressF ;
     dataLEaddressB <= sig_dataLEaddressB ;
     parityaddressF <= sig_countup ;
     parityaddressB <= sig_countdown;
     gammaAddressF <= sig_countup;
     gammaAddressB <= sig_countdown;
     alphaAddress <= sig_alphaAddress;
     betaAddress <= sig_betaAddress;
     datamemenable <= sig_datamemenable;
     parity1enable <= sig_parity1enable;
     parity2enable <= sig_parity2enable ;
     LEmemenable <= sig_LEmemenable ;
     datawr <= sig_datawr ;
     parity1wr <= sig_parity1wr ;
     parity2wr <= sig_parity2wr ;
     LEwr <= sig_LEwr ;
```

```
    alphamemenable <= sig_alphamemenable ;
    betamemenable  <= sig_betamemenable ;
    gammamemenable <= sig_gammamemenable;
    alphawr      <= sig_alphawr ;
    betawr       <= sig_betawr ;
    gammawr      <= sig_gammawr ;
    alphaenable   <= sig_alphaenable ;
    betaenable    <= sig_betaenable ;
    gammaenable   <= sig_gammaenable ;
    LLRenable     <= sig_LLRenable  ;
    LEenable      <= sig_LEenable;
    Ld_Areg       <= sig_Ld_Areg  ;
    Ld_Breg       <= sig_Ld_Breg ;

    decoder       <= sig_decoder;
    decode        <= sig_decode;


end beha; --Behavior of the control module
```

```
--This is the module for complete decoder
--It consists all the module including the memory modules
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.AlphaType.all;
--Entity definition
entity TestGammaAlphaBeta is
 port (clk  : in std_logic;
     start : in std_logic;
    --for the test purposes, in actual implementation not required
     testcountup : out std_logic_vector(9 downto 0);        -- just for test purpose
     testcountdown : out std_logic_vector(9 downto 0);      -- just for test purpose
     testdecode  : out std_logic;                           -- just for test purpose
     testdecoder  : out std_logic;                          -- just for test purpose
     testllrenable : out std_logic;
     gammaFout : out Gamma;
     gammaBout : out Gamma;
     alphaout : out Alpha;
     betaout  : out Alpha;
   --Following two outputs are the actual output from the decoder
     DataoutB : out std_logic;      --decoded data from the forward modules
     DataoutF  : out std_logic      --decoded data from the backward modules

     );
```

```
end entity;

architecture beha of TestGammaAlphaBeta is
  --data memory generated by Quartus megafunction
 -- Dual port memory
component datamem
        PORT
        (
                address_a       : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                address_b       : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clock           : IN STD_LOGIC ;
                data_a          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                data_b          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                enable          : IN STD_LOGIC  := '1';
                wren_a          : IN STD_LOGIC  := '1';
                wren_b          : IN STD_LOGIC  := '1';
                q_a             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
                q_b             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;

--Memory for parity one, Dual port memory
component parity1mem
        PORT
        (
                address_a       : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                address_b       : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clock           : IN STD_LOGIC ;
                data_a          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                data_b          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                enable          : IN STD_LOGIC  := '1';
                wren_a          : IN STD_LOGIC  := '1';
                wren_b          : IN STD_LOGIC  := '1';
                q_a             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
                q_b             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
--Memory for parity2; Dual port memory
component parity2mem
        PORT
        (
                address_a       : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                address_b       : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clock           : IN STD_LOGIC ;
                data_a          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                data_b          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
```

```
                enable          : IN STD_LOGIC := '1';
                wren_a          : IN STD_LOGIC := '1';
                wren_b          : IN STD_LOGIC := '1';
                q_a             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
                q_b             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
--Memory for extrinsic values; Dual port memory
component LEmem
        PORT
        (
                address_a       : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                address_b       : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clock           : IN STD_LOGIC ;
                data_a          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                data_b          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                enable          : IN STD_LOGIC := '1';
                wren_a          : IN STD_LOGIC := '1';
                wren_b          : IN STD_LOGIC := '1';
                q_a               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
                q_b               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
--Memory for Alpha(0)
component alpha0
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                --rden          : IN STD_LOGIC ;
                wren            : IN STD_LOGIC ;
                q                 : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
--Memory for Alpha(1)
component alpha1
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
```

```
        );
end component;
--Memory for Alpha(2)
component alpha2
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
--Memory for Alpha(3)
component alpha3
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
--Memory for Alpha(4)
component alpha4
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
--Memory for Alpha(5)
component alpha5
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
```

```
		wren		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
	);
end component;
--Memory for Alpha(6)
component alpha6
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		clken		: IN STD_LOGIC ;
		clock		: IN STD_LOGIC ;
		data		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		wren		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
	);
end component;
component alpha7  --Memory for Alpha(7)
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		clken		: IN STD_LOGIC ;
		clock		: IN STD_LOGIC ;
		data		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		wren		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
	);
end component;
component alpha8  --Memory for Alpha(8)
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		clken		: IN STD_LOGIC ;
		clock		: IN STD_LOGIC ;
		data		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		wren		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
	);
end component;
component alpha9  --Memory for Alpha(9)
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		clken		: IN STD_LOGIC ;
		clock		: IN STD_LOGIC ;
		data		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		wren		: IN STD_LOGIC ;
```

```
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component alpha10  --Memory for Alpha(10)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component alpha11   --Memory for Alpha(11)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component alpha12      --Memory for Alpha(12)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component alpha13  --Memory for Alpha(13)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
```

```
end component;
component alpha14   --Memory for Alpha(14)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component alpha15   --Memory for Alpha(15)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;

component beta0   --Memory for Beta(0)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta1   --Memory for Beta(1)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
```

```
component beta2  --Memory for Beta(2)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta3  --Memory for Beta(3)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta4     --Memory for Beta(4)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta5     --Memory for Beta(5)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta6    --Memory for Beta(6)
        PORT
```

```
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta7     --Memory for Beta(7)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;

component beta8     --Memory for Beta(8)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta9     --Memory for Beta(9)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta10    --Memory for Beta(10)
        PORT
        (
```

```
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta11    --Memory for Beta(11)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;


component beta12    --Memory for Beta(12)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta13    --Memory for Beta(13)
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component beta14    --Memory for Beta(14)
        PORT
        (
```

```
		address		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		clken		: IN STD_LOGIC ;
		clock		: IN STD_LOGIC ;
		data		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		wren		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
	);
end component;
component beta15	--Memory for Beta(15)
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		clken		: IN STD_LOGIC ;
		clock		: IN STD_LOGIC ;
		data		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		wren		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
	);
end component;


component gamma0	--Memory for gamma00; Dual port memory
	PORT
	(
		address_a		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		address_b		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		data_a		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		data_b		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		enable		: IN STD_LOGIC := '1';
		wren_a		: IN STD_LOGIC := '1';
		wren_b		: IN STD_LOGIC := '1';
		q_a		: OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
		q_b		: OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
	);
end component;
component gamma1	--Memory for gamma01; Dual port memory
	PORT
	(
		address_a		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		address_b		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		data_a		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		data_b		: IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		enable		: IN STD_LOGIC := '1';
		wren_a		: IN STD_LOGIC := '1';
```

```
                wren_b          : IN STD_LOGIC  := '1';
                q_a             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
                q_b             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component gamma2    --Memory for gamma10; Dual port memory
        PORT
        (
                address_a               : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                address_b               : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clock           : IN STD_LOGIC ;
                data_a          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                data_b          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                enable          : IN STD_LOGIC  := '1';
                wren_a          : IN STD_LOGIC  := '1';
                wren_b          : IN STD_LOGIC  := '1';
                q_a             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
                q_b             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;

component gamma3    --Memory for gamma11; Dual port memory
        PORT
        (
                address_a               : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                address_b               : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clock           : IN STD_LOGIC ;
                data_a          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                data_b          : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                enable          : IN STD_LOGIC  := '1';
                wren_a          : IN STD_LOGIC  := '1';
                wren_b          : IN STD_LOGIC  := '1';
                q_a             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
                q_b             : OUT STD_LOGIC_VECTOR (9 DOWNTO 0)
        );
end component;
component FinalBeta is   --FinalBeta module
 port( enable  : in std_logic;
     betain  : in Alpha;
     gammain  : Gamma;
     Betaout : out Alpha
    );
end component;

component FinalAlpha is   --FinalAlpha module
 port( enable  : in std_logic;
```

```
      alphain  : in Alpha;
      gammain  : Gamma;
      Alphaout : out Alpha
    );
end component;

component gammaF is    --GammaF module
 port  ( enable  : in std_logic;
       datain   : in std_logic_vector(9 downto 0);
       parityin : in std_logic_vector(9 downto 0);
       L_eadd   : in std_logic_vector(9 downto 0);
       gammaout  : out Gamma

      );
end component;
component LLRCore is    --LLRCore module
 port ( enable  : in std_logic;
      alphain : in Alpha;
      betain  : in Alpha;
      gammain : in Gamma;
      LLRout  : out std_logic_vector(9 downto 0)
     );
end component;
component LEModule is   --LEModule definition
 port ( enable  : in std_logic;
      LLRin   : in std_logic_vector(9 downto 0);
      datain  : in std_logic_vector(9 downto 0);
      LEin    : in std_logic_vector(9 downto 0);
      LEout   : out std_logic_vector(9 downto 0)
      );
end component;
component control is    --control module definition
 port( clk       : in std_logic;
     start      : in std_logic;
     countup          : out std_logic_vector(9 downto 0);
     countdown        : out std_logic_vector(9 downto 0);
     dataLEaddressF   : out std_logic_vector(9 downto 0);
     dataLEaddressB   : out std_logic_vector(9 downto 0);
     parityaddressF   : out std_logic_vector(9 downto 0);
     parityaddressB   : out std_logic_vector(9 downto 0);
     alphaAddress     : out std_logic_vector(9 downto 0);
     betaAddress      : out std_logic_vector(9 downto 0);
     gammaAddressF      : out std_logic_vector(9 downto 0);
     gammaAddressB      : out std_logic_vector(9 downto 0);
     datamemenable : out std_logic;
     parity1enable : out std_logic;
```

```
    parity2enable : out std_logic;
    LEmemenable     : out std_logic;
    datawr      : out std_logic;
    parity1wr    : out std_logic;
    parity2wr    : out std_logic;
    LEwr       : out std_logic;
    alphamemenable :out std_logic;
    betamemenable  : out std_logic;
    gammamemenable : out std_logic;
    alphawr      : out std_logic;
    betawr       : out std_logic;
    gammawr      : out std_logic;
    alphaenable   : out std_logic;
    betaenable    : out std_logic;
    gammaenable   : out std_logic;
    LLRenable     : out std_logic;
    LEenable      : out std_logic;
    Ld_Areg       : out std_logic;
    Ld_Breg       : out std_logic;
    decode        : out std_logic;
    decoder       : out std_logic
  );
end component;

 --Intermediate signal definitions
signal sig_gammaF,sig_gammaB : Gamma;
signal tempalpha,tempbeta : Alpha;
signal iniAlpha : Alpha;
signal iniBeta  : Alpha ;
signal    sig_countup         : std_logic_vector(9 downto 0);
signal    sig_countdown       : std_logic_vector(9 downto 0);
signal    sig_dataLEaddressB   : std_logic_vector(9 downto 0);
signal    sig_dataLEaddressF   : std_logic_vector(9 downto 0);
signal    sig_parityaddressF    : std_logic_vector(9 downto 0);
signal    sig_parityaddressB   : std_logic_vector(9 downto 0);
signal    sig_alphaAddress     : std_logic_vector(9 downto 0);
signal    sig_betaAddress      : std_logic_vector(9 downto 0);
signal    sig_gammaAddressF    : std_logic_vector(9 downto 0);
signal    sig_gammaAddressB    : std_logic_vector(9 downto 0);
signal    sig_datamemenable :  std_logic;
signal    sig_parity1enable :  std_logic;
signal    sig_parity2enable :  std_logic;
signal    sig_LEmemenable     :  std_logic;
signal    sig_datawr      :  std_logic;
signal    sig_parity1wr    :  std_logic;
signal    sig_parity2wr    :  std_logic;
```

```
signal     sig_LEwr       : std_logic;
signal     sig_alphamemenable : std_logic;
signal     sig_betamemenable  : std_logic;
signal     sig_gammamemenable : std_logic;
signal     sig_alphawr      : std_logic;
signal     sig_betawr       : std_logic;
signal     sig_gammawr      : std_logic;
signal     sig_alphaenable   : std_logic;
signal     sig_betaenable    : std_logic;
signal     sig_gammaenable   : std_logic;
signal     sig_LLRenable     : std_logic;
signal     sig_Ld_Areg      : std_logic;
signal     sig_Ld_Breg      : std_logic;
signal     sig_decode       : std_logic;
signal     sig_LEenable     : std_logic;
signal     sig_dataoutF,sig_dataoutB      : std_logic;
--Alpha register is initialized to (1=>0, rest =>-32(Highest value for (10,4) signed representation)
signal Alpha_reg : Alpha := ("0000000000", "1000000000", "1000000000", "1000000000",
                             "1000000000", "1000000000", "1000000000", "1000000000",
                             "1000000000", "1000000000", "1000000000", "1000000000",
                             "1000000000", "1000000000", "1000000000", "1000000000" );
--Beta values initialized for decoder 1; encoder1  is terminated to state 0
signal Beta_reg0 : Alpha := ( "0000000000", "1000000000", "1000000000", "1000000000",
                              "1000000000", "1000000000", "1000000000", "1000000000",
                              "1000000000", "1000000000", "1000000000", "1000000000",
                              "1000000000", "1000000000", "1000000000", "1000000000" );
--Beta values initialized for decoder 2.  Encoder 2 is not terminated.
signal Beta_reg1 : Alpha := (others => "1111010100"); --log(1/16), all states are possible
signal LLRAlpha,LLRBeta  : Alpha;
signal gammaFadd,gammaBadd : std_logic_vector(9 downto 0);
signal tempgammaoutF,tempgammaoutB : Gamma;
signal tempgammaF,tempgammaB      : Gamma;
signal gammaFin,gammaBin          : Gamma;
signal sig_decoder    : std_logic ;
signal tempdataF,tempdataB : std_logic_vector(9 downto 0);
signal tempparity1F,tempparity1B : std_logic_vector(9 downto 0);
signal tempparity2F,tempparity2B : std_logic_vector(9 downto 0);
signal tempparityF,tempparityB  : std_logic_vector(9 downto 0);
signal tempLEF,tempLEB       : std_logic_vector(9 downto 0);
signal sig_LEoutF,sig_LEoutB,sig_LLRoutB,sig_LLRoutF : std_logic_vector(9 downto 0);
begin
--based on the decoder parity address are assigned
 tempparityF <= tempparity1F when (sig_parity1enable = '1') else
          tempparity2F;
 tempparityB <= tempparity1B when sig_parity1enable = '1' else
          tempparity2B;
```

```
--Decoding process for backward direction
DECODEDATAB : process(sig_LLRoutB,sig_decode)  --decode the data when decode is high
        begin
          if sig_decode = '1'  then
           if (sig_LLRoutB(9) = '1') then  --negative LLR
             sig_dataoutB <= '0';
           else                            --positive LLR
            sig_dataoutB <= '1';
           end if;
           end if;
        end process;              --End of decoding process
--Decoding process for forward direction
DECODEDATAF : process(sig_LLRoutF,sig_decode)
          begin
          if sig_decode = '1'  then
           if sig_LLRoutF(9) = '1' then
            sig_dataoutF <= '0';
           else
            sig_dataoutF <= '1';
           end if;
          end if;
         end process;
--Load the register based on the decoder and the frame position
LOADREGISTERA : process(sig_Ld_Areg,tempalpha)
          begin

          if (sig_countup = "0000000000" and sig_Ld_Areg = '1') then
            inialpha <= Alpha_reg;  --inialpha is loaded from the Alpha register(initialization)
           elsif(sig_countup > "0000000000" and sig_Ld_Areg = '1') then
            inialpha <= tempalpha;  --inialpha is loaded from the FinalAlpha module
           end if;


          if sig_countup = "0000000000" then
            if (sig_decoder = '1'  and sig_Ld_Breg = '1') then
             iniBeta <= beta_reg1; --inibeta is loaded for decoder 2 from the register beta_reg1
            elsif(sig_decoder = '0'  and sig_Ld_Breg = '1') then
             iniBeta <= beta_reg0;  --inibeta is loaded for decoder 1 from the register beta_reg0
            end if;
           elsif (sig_countup > "0000000000"  and sig_Ld_Breg = '1') then
            iniBeta <= tempbeta;  --inibeta is loaded from the FinalBeta
           end if;

         end process;  --end the load process
--Gamma values are either stored or read based on the frame position
SELECTGAMMAF : process(tempgammaF,tempgammaoutF)  --Forward direction
```

```
        begin

          if (sig_countup <= "0111111111") then
            gammaFin <= tempgammaF;  --from the gammaF module, needs to be stored

          else
           gammaFin <= tempgammaoutF; --from the memory module

          end if;

        end process;
SELECTGAMMAB : process(tempgammaB,tempgammaoutB) --backward direction
        begin

          if (sig_countup <= "0111111111") then
            gammaBin <= tempgammaB;   --from the gammaF module, to be stored
          else
            gammaBin <= tempgammaoutB; --From the gamma  memory
          end if;

        end process;

 --Instantiate the  control module
CONTROL_INST : control port map(
                              clk          =>clk,
                              start         =>start,
                              countup       => sig_countup,
                              countdown     => sig_countdown,
                              dataLEaddressF   => sig_dataLEaddressF,
                              dataLEaddressB   => sig_dataLEaddressB,
                              parityaddressF   => sig_parityaddressF,
                              parityaddressB   => sig_parityaddressB,
                              alphaAddress     => sig_alphaAddress,
                              betaAddress      => sig_betaAddress,
                              gammaAddressF    => sig_gammaAddressF,
                              gammaAddressB    => sig_gammaAddressB,
                              datamemenable    => sig_datamemenable,
                              parity1enable    => sig_parity1enable,
                              parity2enable    => sig_parity2enable,
                              LEmemenable      => sig_LEmemenable,
                              datawr        => sig_datawr,
                              parity1wr      => sig_parity1wr,
                              parity2wr      => sig_parity2wr,
                              LEwr          => sig_LEwr,
                              alphamemenable   => sig_alphamemenable,
                              betamemenable    => sig_betamemenable,
```

```
                        gammamemenable   => sig_gammamemenable,
                        alphawr          => sig_alphawr,
                        betawr           => sig_betawr,
                        gammawr          => sig_gammawr,
                        alphaenable      => sig_alphaenable,
                        betaenable       => sig_betaenable,
                        gammaenable      => sig_gammaenable,
                        LLRenable        => sig_LLRenable,
                        LEenable         => sig_LEenable,

                        Ld_Areg          => sig_Ld_Areg,
                        Ld_Breg          => sig_Ld_Breg,
                        decode           => sig_decode,
                        decoder          => sig_decoder
                       );
 --Instantiate the data memory module
--Memory is read only during the decoding process
datamem_inst : datamem PORT MAP (
                address_a       => sig_dataLEaddressF,
                address_b       => sig_dataLEaddressB,
                clock    => clk,
                data_a   => "0000000000",
                data_b   => "0000000000",
                enable   => sig_datamemenable,
                wren_a   => sig_datawr,
                wren_b   => sig_datawr,
                q_a      => tempdataF,
                q_b      => tempdataB
        );
--Instantiate the parity1 memory module
--Memory is read only during the decoding process
parit1ymem_inst : parity1mem PORT MAP (
                address_a       => sig_parityaddressF,
                address_b       => sig_parityaddressB,
                clock    => clk,
                data_a   => "0000000000",
                data_b   => "0000000000",
                enable   => sig_parity1enable,
                wren_a   => '0',
                wren_b   => '0',
                q_a      => tempparity1F,
                q_b      => tempparity1B
        );
--Instantiate the parity2 memory module
--Memory is read only during the decoding process
parity2mem_inst : parity2mem PORT MAP (
```

```
                address_a       => sig_parityaddressF,
                address_b       => sig_parityaddressB,
                clock   => clk,
                data_a  => "0000000000",
                data_b  => "0000000000",
                enable  => sig_parity2enable,
                wren_a  => '0',
                wren_b  => '0',
                q_a     => tempparity2F,
                q_b     => tempparity2B
        );
--Instantiate the gamma00 memory module
--Memory is read/written during the decoding process
gamma0_inst : gamma0 PORT MAP (
                address_a       => sig_gammaaddressF,
                address_b       => sig_gammaaddressB,
                clock   => clk,
                data_a  => tempgammaF(0),
                data_b  => tempgammaB(0),
                enable  => sig_gammamemenable,
                wren_a  => sig_gammawr,
                wren_b  => sig_gammawr,
                q_a     => tempgammaoutF(0),
                q_b     => tempgammaoutB(0)
        );
--Instantiate the gamma01 memory module
--Memory is read/written during the decoding process
gamma1_inst : gamma1 PORT MAP (
                address_a       => sig_gammaaddressF,
                address_b       => sig_gammaaddressB,
                clock   => clk,
                data_a  => tempgammaF(1),
                data_b  => tempgammaB(1),
                enable  => sig_gammamemenable,
                wren_a  => sig_gammawr,
                wren_b  => sig_gammawr,
                q_a     => tempgammaoutF(1),
                q_b     => tempgammaoutB(1)
        );
--Instantiate the gamma10 memory module
--Memory is read/written during the decoding process
gamma2_inst : gamma2 PORT MAP (
                address_a       => sig_gammaaddressF,
                address_b       => sig_gammaaddressB,
                clock   => clk,
                data_a  => tempgammaF(2),
```

```
                data_b  => tempgammaB(2),
                enable  => sig_gammamemenable,
                wren_a  => sig_gammawr,
                wren_b  => sig_gammawr,
                q_a     => tempgammaoutF(2),
                q_b     => tempgammaoutB(2)
        );
--Instantiate the gamma11 memory module
--Memory is read/written during the decoding process
gamma3_inst : gamma3 PORT MAP (
                address_a       => sig_gammaaddressF,
                address_b       => sig_gammaaddressB,
                clock   => clk,
                data_a  => tempgammaF(3),
                data_b  => tempgammaB(3),
                enable  => sig_gammamemenable,
                wren_a  => sig_gammawr,
                wren_b  => sig_gammawr,
                q_a     => tempgammaoutF(3),
                q_b     => tempgammaoutB(3)
        );
--Instantiate the LEmem memory module
--Memory is read/written during the decoding process
LEmem_inst : LEmem PORT MAP (
                address_a       => sig_dataLEaddressF,
                address_b       => sig_dataLEaddressB,
                clock   => clk,
                data_a  => sig_LEoutF,
                data_b  => sig_LEoutB,
                enable  => sig_LEmemenable,
                wren_a  => sig_LEwr,
                wren_b  => sig_LEwr,
                q_a     => tempLEF,
                q_b     => tempLEB
        );
--Instantiate the alpha(0) memory
alpha0_inst : alpha0 PORT MAP (
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(0),
                --rden    => '0',
                wren    => sig_alphawr,
                q           => LLRAlpha(0)
        );
--Instantiate the alpha(1) memory
```

```
alpha1_inst : alpha1 PORT MAP (
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(1),
                wren    => sig_alphawr,
                q       => LLRAlpha(1)
        );
alpha2_inst : alpha2 PORT MAP (                 --Instantiate the alpha(2) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(2),
                wren    => sig_alphawr,
                q       => LLRAlpha(2)
        );
alpha3_inst : alpha3 PORT MAP (                 --Instantiate the alpha(3) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(3),
                wren    => sig_alphawr,
                q       => LLRAlpha(3)
        );
alpha4_inst : alpha4 PORT MAP (                 --Instantiate the alpha(4) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(4),
                wren    => sig_alphawr,
                q       => LLRAlpha(4)
        );
alpha5_inst : alpha5 PORT MAP (                 --Instantiate the alpha(5) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(5),
                wren    => sig_alphawr,
                q       => LLRAlpha(5)
        );
alpha6_inst : alpha6 PORT MAP (         --Instantiate the alpha(6) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(6),
                wren    => sig_alphawr,
```

```
                q       => LLRAlpha(6)
        );
alpha7_inst : alpha7 PORT MAP (     --Instantiate the alpha(7) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(7),
                wren    => sig_alphawr,
                q       => LLRAlpha(7)
        );
alpha8_inst : alpha8 PORT MAP (     --Instantiate the alpha(8) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(8),
                wren    => sig_alphawr,
                q       => LLRAlpha(8)
        );
alpha9_inst : alpha9 PORT MAP (     --Instantiate the alpha(9) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(9),
                wren    => sig_alphawr,
                q       => LLRAlpha(9)
        );
alpha10_inst : alpha10 PORT MAP (   --Instantiate the alpha(10) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(10),
                wren    => sig_alphawr,
                q       => LLRAlpha(10)
        );
alpha11_inst : alpha11 PORT MAP (   --Instantiate the alpha(11) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(11),
                wren    => sig_alphawr,
                q       => LLRAlpha(11)
        );
alpha12_inst : alpha12 PORT MAP (   --Instantiate the alpha(12) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
```

```
                data    => iniAlpha(12),
                wren    => sig_alphawr,
                q       => LLRAlpha(12)
        );
alpha13_inst : alpha13 PORT MAP (       --Instantiate the alpha(13) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(13),
                wren    => sig_alphawr,
                q       => LLRAlpha(13)
        );
alpha14_inst : alpha14 PORT MAP (       --Instantiate the alpha(14) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(14),
                wren    => sig_alphawr,
                q       => LLRAlpha(14)
        );
alpha15_inst : alpha15 PORT MAP (       --Instantiate the alpha(15) memory
                address => sig_alphaAddress,
                clken   => sig_alphamemenable,
                clock   => clk,
                data    => iniAlpha(15),
                wren    => sig_alphawr,
                q       => LLRAlpha(15)
        );

beta0_inst : beta0 PORT MAP (     --Instantiate the beta(0) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(0),
                wren    => sig_betawr,
                q       => LLRBeta(0)
        );
beta1_inst : beta1 PORT MAP (       --Instantiate the beta(1) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(1),
                wren    => sig_betawr,
                q       => LLRBeta(1)
        );
beta2_inst : beta2 PORT MAP (     --Instantiate the beta(2) memory
```

```
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(2),
                wren    => sig_betawr,
                q       => LLRbeta(2)
        );
beta3_inst : beta3 PORT MAP (       --Instantiate the beta(3) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(3),
                wren    => sig_betawr,
                q       => LLRBeta(3)
        );
beta4_inst : beta4 PORT MAP (       --Instantiate the beta(4) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(4),
                wren    => sig_betawr,
                q       => LLRBeta(4)
        );
beta5_inst : beta5 PORT MAP (       --Instantiate the beta(5) memory
                address =>sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(5),
                wren    => sig_betawr,
                q       => LLRBeta(5)
        );
beta6_inst : beta6 PORT MAP (       --Instantiate the beta(6) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(6),
                wren    => sig_betawr,
                q       => LLRBeta(6)
        );
beta7_inst : beta7 PORT MAP (       --Instantiate the beta(7) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(7),
                wren    => sig_betawr,
                q       => LLRBeta(7)
```

```
        );
beta8_inst : beta8 PORT MAP (         --Instantiate the beta(8) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(8),
                wren    => sig_betawr,
                q       => LLRBeta(8)
        );
beta9_inst : beta9 PORT MAP (         --Instantiate the beta(9) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(9),
                wren    => sig_betawr,
                q       => LLRBeta(9)
        );
beta10_inst : beta10 PORT MAP (       --Instantiate the beta(10) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(10),
                wren    => sig_betawr,
                q       => LLRBeta(10)
        );
beta11_inst : beta11 PORT MAP (       --Instantiate the beta(11) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(11),
                wren    => sig_betawr,
                q       => LLRBeta(11)
        );
beta12_inst : beta12 PORT MAP (       --Instantiate the beta(12) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(12),
                wren    => sig_betawr,
                q       => LLRBeta(12)
        );
beta13_inst : beta13 PORT MAP (       --Instantiate the beta(13) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(13),
```

```
                wren    => sig_betawr,
                q       => LLRBeta(13)
        );
beta14_inst : beta14 PORT MAP (     --Instantiate the beta(14) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(14),
                wren    => sig_betawr,
                q       => LLRBeta(14)
        );
beta15_inst : beta15 PORT MAP (     --Instantiate the beta(15) memory
                address => sig_betaAddress,
                clken   => sig_betamemenable,
                clock   => clk,
                data    => iniBeta(15),
                wren    => sig_betawr,
                q       => LLRBeta(15)
        );


--Instantiate GammaF module for forward direction
CALCULATEGAMMAF : gammaF port map
          (sig_gammaenable,tempdataF,tempparityF,tempLEF,tempgammaF);
--Instantiate GammaF module for backware direction
CALCULATEGAMMAB : gammaF port map
         (sig_gammaenable,tempdataB,tempparityB,tempLEB,tempgammaB);
--Instantiate the FinalAlpha module
CALCULATEALPHA  : FinalAlpha port map(sig_alphaenable,inialpha,gammaFin,tempalpha);
--Instantiate the FinalBeta module
CALCULATEBETA   : FinalBeta port map(sig_betaenable,iniBeta,gammaBin,tempbeta);
--Instantiate LLRCore module for forward direction
CALCULATELLRF   : LLRCore PORT MAP( enable  =>sig_LLRenable,
                                    alphain =>tempalpha,
                                    betain  => LLRBeta,
                                    gammain => gammaFin,
                                    LLRout  => sig_LLRoutF
                                     );
--Instantiate LLRCore module for backward direction
CALCULATELLRB   : LLRCore PORT MAP( enable  =>sig_LLRenable,
                                    alphain =>LLRAlpha,
                                    betain  =>tempbeta,
                                    gammain => gammaBin,
                                    LLRout  => sig_LLRoutB
                                     );
--Instantiate LE module for forward direction
```

```
CALCULATELEF : LEmodule port map(
            enable  => sig_leenable,
            LLRin    => sig_LLRoutF,
            datain   => tempdataF,
            LEin     => tempLEF,
            LEout    => sig_LEoutF
            );
--Instantiate LE module for backward direction
CALCULATELEB : LEmodule port map(
            enable  => sig_leenable,
            LLRin    => sig_LLRoutB,
            datain   => tempdataB,
            LEin     => tempLEB,
            LEout    => sig_LEoutB
            );

    gammaFout <= gammaFin;      --for test purpose
    gammaBout <= gammaBin;      --for test purpose
    alphaout <= tempalpha;      --for test purpose
    betaout  <= tempbeta;       --for test purpose
    testcountup <= sig_countup;  --for test purpose
    testcountdown <= sig_countdown;   --for test purpose
    testdecode   <= sig_decode;       --for test purpose
    testdecoder  <= sig_decoder;      --for test purpose
    dataoutF    <= sig_dataoutF;      --Assign the decoded data to dataoutF
    dataoutB    <= sig_dataoutB;      --Assign the decoded data to dataoutB
    testllrenable <= sig_llrenable;   --for test purpose
end beha;      --End of the behavior of the complete decoder
```

```
--This is the test bench for the decoder module

library ieee;
use ieee.std_logic_1164.all;

package AlphaType is
type Alpha is array(0 to 15) of std_logic_vector(9 downto 0);
type Gamma is array(0 to 3) of std_logic_vector(9 downto 0);
end package;
library ieee,std;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use ieee.numeric_std.all;
use std.textio.all;   --for file handling
use work.AlphaType.all;
```

```
entity testbenchGALBETA is
end entity;

architecture beha of testbenchGALBETA is
signal clk : std_logic := '0';
signal start : std_logic := '1' ;
signal test_countup : std_logic_vector(9 downto 0);
signal test_countdown : std_logic_vector(9 downto 0);
signal test_decode   : std_logic;
signal test_decoder  : std_logic;
signal test_dataoutF  : std_logic;
signal test_dataoutB  : std_logic;
signal test_llrenable : std_logic;
--file for writing the decoded data in the forward direction
file out_fileF :text open WRITE_MODE IS "/home/vlsi/thanga2/dataoutF_file";
--file for writing the decoded data in the backware direction
file out_fileB : text open WRITE_MODE IS "/home/vlsi/thanga2/dataoutB_file";
component TestGammaAlphaBeta is     --component under test
 port (clk  : in std_logic;
      start : in std_logic;
      testcountup : out std_logic_vector(9 downto 0);      -- just for test purpose
      testcoutdown : out std_logic_vector(9 downto 0);    -- just for test purpose
      testdecode   : out std_logic;                        -- just for test purpose
      testdecoder  : out std_logic;                        -- just for test purpose
      testllrenable : out std_logic;
      gammaFout : out Gamma;
      gammaBout : out Gamma;
      alphaout : out Alpha;
      betaout  : out Alpha;
      DataoutB : out std_logic;
      DataoutF   : out std_logic

      );
end component;

 begin
  start <= '0' after 20 ns;   --Start signal is active high for 20 ns
 process(clk)  --clock signal is generated, period is 10ns
 begin
  clk <= not clk after 5 ns;
 end process;
  process(test_llrenable)
  --Variable type to hold the decoded data in line type
  variable out_lineF,out_lineB   : line;  --defined in texio package
begin
```

```
  if(test_decode = '1' and test_llrenable = '0') then
   write(out_lineF,test_dataoutF); --Format the dataoutF
    writeline(out_fileF,out_lineF);  --write the data into the file
    write(out_lineB,test_dataoutB);  --Format the dataoutB
    writeline(out_fileB,out_lineB);    --write the data into the file
  end if;
  end process;
--Instantiate the unit under test
C : TestGammaAlphaBeta port map(clk => clk,
                                start => start,
                                testdecode => test_decode,
                                testdecoder => test_decoder,
                                testllrenable => test_llrenable,
                                dataoutB => test_dataoutB,
                                dataoutF => test_dataoutF);

 end beha;  --End of the test bench
```

VITA AUCTORIS

Krishnamohan Thangarajah was born in Sri Lanka. He completed his degree in engineering titled B.A.Sc. Eng. (Hons.) in Communication Engineering from University of Windsor, Canada in 2008. At the time of writing this thesis Krishnamohan is a candidate for the degree of M. A. Sc. in Electrical and Computer Engineering, at the University of Windsor (Ontario, Canada).

His research interests include error correction coding and communications systems, digital signal processing on FPGAs for such systems and hardware realizations, and wireless communication systems.