DOUBLE BINARY TURBO CODES ANALYSIS AND DECODER IMPLEMENTATION

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULLFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Özlem Yılmaz
September 2008

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Abdullah Atalar (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Erdal Arıkan

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. İbrahim Körpeoğlu

Approved for the Institute of Engineering and Sciences:

Prof. Dr. Mehmet B. Baray
Director of Institute of Engineering and Sciences

ABSTRACT

DOUBLE BINARY TURBO CODE ANALYSIS AND DECODER IMPLEMENTATION

Özlem Yılmaz
M.S. in Electrical and Electronics Engineering
Supervisor: Prof. Dr. Abdullah Atalar

September 2008

Classical Turbo Code presented in 1993 by Berrau et al. received great attention due to its near Shannon Limit decoding performance. Double Binary Circular Turbo Code is an improvement on Classical Turbo Code and widely used in today's communication standards, such as IEEE 802.16 (WIMAX) and DVB-RSC. Compared to Classical Turbo Codes, DB-CTC has better error-correcting capability but more computational complexity for the decoder scheme. In this work, various methods, offered to decrease the computational complexity and memory requirements of DB-CTC decoder in the literature, are analyzed to find the optimum solution for the FPGA implementation of the decoder. IEEE 802.16 standard is taken into account for all simulations presented in this work and different simulations are performed according to the specifications given in the standard. An efficient DB-CTC decoder is implemented on an FPGA board and compared with other implementations in the literature.

Keywords: Double Binary Turbo Codes, IEEE 802.16, FPGA, decoder.

ÖZET

ÇİFT İKİLİ TURBO KOD ANALİZİ ve KOD ÇÖZÜCÜ UYGULAMASI

Özlem Yılmaz
Elektrik ve Elektronik Mühendisliği Bölümü Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Abdullah Atalar
Eylül 2008

İlk olarak 1993 senesinde Berrou tarafından tariflenen klasik Turbo kodlar, Shannon sınırına yakın kod çözücü performansları sayesinde büyük ilgi toplamıştır. Çift ikili dönel Turbo kodları, klasik Turbo kodların daha da gelişmiş halidir ve IEEE 802.16 (WIMAX) and DVB-RSC gibi bugünün haberleşme standartlarında yaygın olarak kullanılmaktadır. Bu kodlar, klasik Turbo kodlara kıyasla daha iyi hata düzeltme yeteneğine sahip olmakla birlikte çözücü açısından daha fazla hesapsal karmaşa içermektedir. Bu çalışmada, çift ikili turbo kod çözücünün alan programlanır kapı dizilerinde en verimli şekilde uygulanması için, literatürde hesaplama karmaşıklığını ve gerekli hafiza alanını azaltmaya yönelik yapılmış çalışmalar araştırılmıştır. Çalışmada IEEE 802.16 standardı baz alınmıştır ve burada verilen belirtimlere uygun olarak simülasyonlar yapılmıştır. Yapılan araştırmaya göre, alan programlanır kapı dizilerinde verimli bir çift ikili turbo kod çözücü uygulaması geliştirilmiştir ve daha önce alan programlanır kapı dizilerinde uygulanan kod çözücülerle karşılaştırılmıştır.

Anahtar Kelimeler: Çift İkili Turbo Kodlar, IEEE 802.16, Alan Programlanır Kapı Dizileri, kod çözücü

Acknowledgements

I would like to express my gratitude to Prof. Abdullah Atalar for his guidance and supervision throughout the development of this thesis; I would also like to gratefully thank Prof. Erdal Arıkan for suggesting, leading the project and providing FPGA board.

I would like to thank my committee member Assist. Prof. İbrahim Körpeoğlu for reading and commenting on this thesis.

I would like to express my thanks to Cahit Uğur Ungan and Oğuzhan Atak for sharing their knowledge with me.

Special thanks to Erdem Ersagun for reviewing my thesis and all his support and understanding throughout the development of this thesis.

I would also like to express my thanks to Duygu Ceylan, Soner Çınar and other colleagues in Aselsan for their support and understanding during my studies.

Last but not the least I would like to thank my family for their endless support, encouragement and love throughout my life.

Table of Contents

List of Figures

List of Tables

To My Family…

Chapter 1

Introduction

In wireless communication systems, received data from the transmitter is corrupted due to the imperfectness of the channel. Error correcting codes are used to reduce the error rate in the received data avoiding increase of transmission power. There are two types of error correcting. In ARQ (Automatic Repeat reQuest) case, receiver sends an acknowledge message to the transmitter upon the reception of a data without error. If transmitter can not receive an acknowledge message in a predetermined time interval, it resends the previously sent data. On the other hand, “forward error correction” (FEC), which is another type of error correcting, uses the redundant bits sent by the transmitter. It avoids retransmission at the cost of high bandwidth requirement and preferred when retransmission is more costly or even impossible. Hybrid ARQ enables using FEC and ARQ together.

FEC is divided into two types: convolutional codes and block codes. Block codes processes on fixed length channel code while convolutional codes work on bits of arbitrary length. Non-recursive convolutional codes are not systematic, meaning that actual bits are not sent through the channel. In this case, output is a linear combination of input bit and delayed input bits. Another type of convolutional code namely recursive convolutional code is systematic and parity output is a function of input bits, delayed input bits and previous input bits. Turbo code is a modified form of convolutional codes in which two

recursive systematic convolutional codes are concatenated in parallel separated by an interleaver.

Turbo coding, first introduced in 1993, aroused great attention due to its near Shannon Limit performance [1]. It allows maximum information transfer over a limited bandwidth. They are widely used in cellular communication systems and specifications for WCDMA (UMTS) and cdma2000 [2]. Non-binary turbo codes introduced in [3] perform better than classical Turbo codes as explained in [4]. Popular radio systems such as DVB-RSC (Digital Video Broadcasting – Return Channel via Satellite) and IEEE 802.16 (WIMAX –Worldwide Interoperability for Microwave Access) [5] standards include double binary turbo codes. On the other hand, compared to classical turbo decoder, double binary turbo decoder is more complex in hardware implementation. Researchers are working on double binary turbo codes to find an efficient way such that the trade off between performance and computational complexity is optimized. First of all, Log-MAP algorithm -the biggest effect on computational complexity- is reduced by using Max-Log-MAP algorithm in the decoders. The performance of the algorithm is improved by using a scaling factor for the calculation of extrinsic information [6]. Another issue causing complexity is the estimation of the initial trellis state at the decoder side. By using feedback method in [6] instead of pre-decoder method, this problem can be solved. Although there are some implementations of the double binary turbo decoder, most of them are based on application specific integrated circuits (ASIC) and not flexible.

In this thesis, investigations improving the performance of the double binary turbo codes are analyzed using MATLAB simulations. Based on the results obtained, double binary turbo decoder is implemented on a field programmable gate array (FPGA). Finally the performance of the decoder is compared to other FPGA implementations in the literature.

Basic information about turbo codes is given and double binary turbo codes are explained in detail together with improvements suggested by other investigators in Chapter 2. MATLAB simulations performed are presented in Chapter 3. Architecture, results of the hardware implementation and the comparison with other implementations are given in Chapter 4. Thesis is concluded in Chapter 5.

Chapter 2

Turbo Code

2.1 Classical Turbo Code

Classical turbo code encoder consists of two rate 1/2 binary recursive systematic convolutional codes concatenated in parallel and separated by a random interleaver as shown in Figure 2.1.

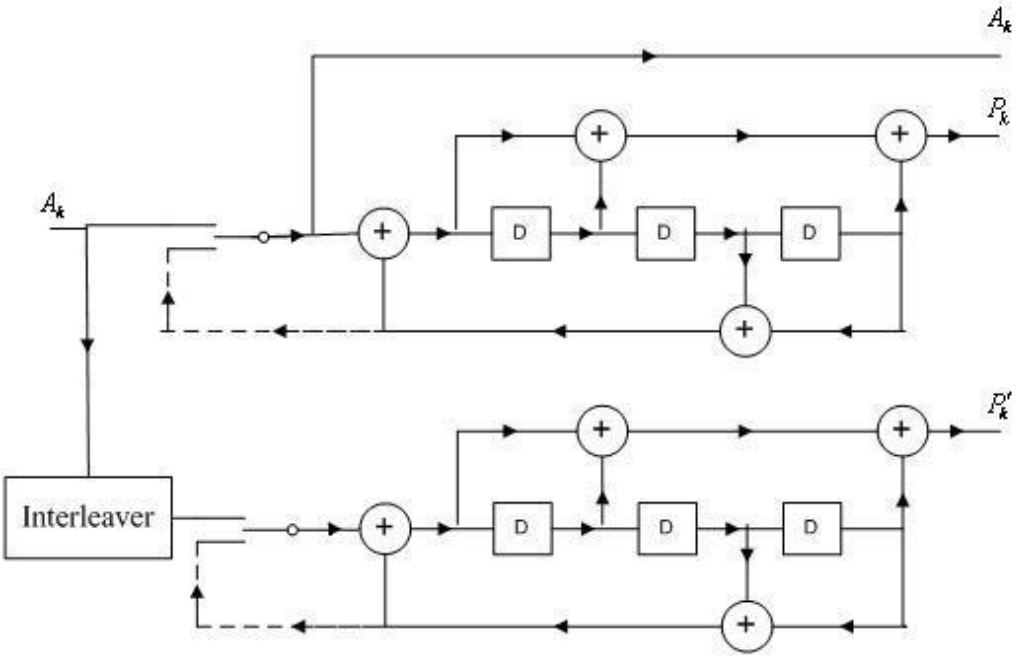

Figure 2.1 Turbo Encoder

In Figure 2.1, upper encoder encodes the data in natural order and lower encoder encodes the interleaved data. Interleaver structure has a big importance on the performance of the turbo codes because it provides the systematic and parity bits sent through the channel are uncorrelated. The data bits A_k and parity

bits P_k, P_k' are transmitted together, thus the overall code rate of the encoder is 1/3. After encoding all data bits, tailing bits are encoded and transmitted to force the trellises of the two encoders to all zero state. It is possible to terminate conventional convolutional codes by transmitting a tail of zeros. However, in the case of recursive convolutional codes, separately calculated tail bits are needed for the encoders [2]. These tail bits are generated by turning the switches in Figure 2.1 on the down position [2].

The turbo decoder is an iterative serial concatenation of two soft output Viterbi or BCJR algorithm decoders as shown in Figure 2.2.

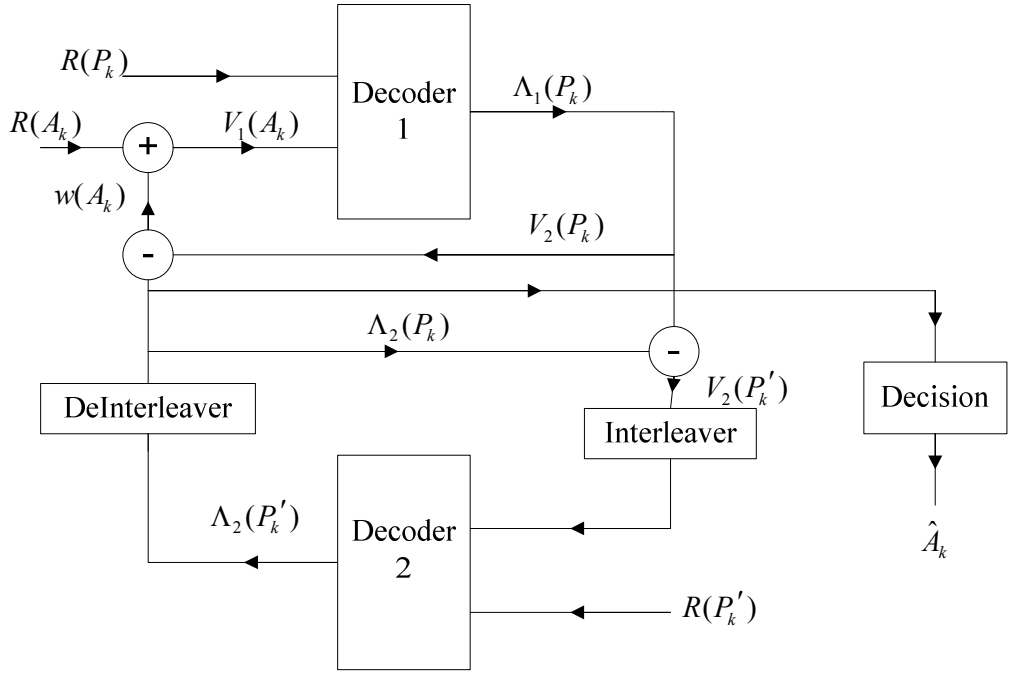

Figure 2.2 Turbo Decoder

Each iteration consists of two half iterations. RSC Decoder 1 works in the first iteration while RSC Decoder 2 works in the second iteration. Decoder 1 uses the received LLR (Log Likelihood Ratios) corresponding to the systematic bits and LLR for the parity bits produced by the first encoder –the encoder which encodes the data in natural order- to produce extrinsic information to be

used by the second decoder. Decoder 2 produces extrinsic information by using the interleaved extrinsic information from the first decoder and LLR of parity bits produced by the second encoder –the encoder which encodes the interleaved data. After de-interleaving process, the extrinsic information is introduced to the first decoder. The progress continues until a reasonable BER or iteration number is reached [2]. This process includes only the actual bits; tail bits are not decoded.

2.2 Double Binary Turbo Code

Recursive Systematic Convolutional codes used in turbo codes are based on single-input linear feedback shift registers (LFSRs). Several information bits can be encoded and decoded at the same time by making use of multiple input LFSRs [3]. It has been shown in [3] that m-input binary turbo codes combined with a two-level permutation performs better than classical turbo codes especially at low SNR and high coding rate. The advantages of m input turbo codes are better convergence of the iterative decoding, large minimum distances, less sensitivity to puncturing patterns, reduced latency, robustness for the weaknesses of the Max-Log-MAP algorithm which is generally preferred as decoding algorithm[4]. Turbo codes with m=2 are called “Double Binary Turbo Codes” and 8 state double binary turbo codes have been widely used in today’s mobile radio systems such as DVB-RCS and IEEE 802.16(WIMAX) standards[5]. Figure 2.3 shows an overall picture of double binary turbo codes including the modulation and demodulation processes. An eight-state Double Binary Turbo Code encoder, interleaver, subblock interleaver, puncturing and decoder structures are explained in the following sections.

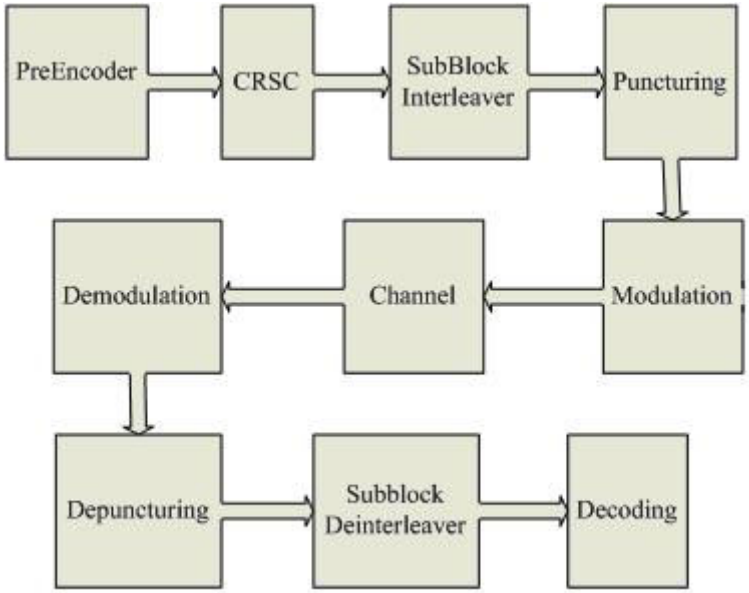

Figure 2.3 Overall picture for Double-Binary CTC

2.2.1 Double Binary Turbo Encoder

Double binary turbo encoder consists of two double binary RSC codes concatenated in parallel as shown in Figure 2.4.

Two data streams A and B are fed to the encoders in natural and interleaved orders. The encoder output consists of systematic bits A and B, parity bits produced by the upper encoder and lower encoder Y1, W1 and Y2, W2 respectively, causing a 2/6 coding rate. In circular double binary Turbo codes, it is ensured that the ending trellis state is equal to the initial trellis state which is called circular state Sc [6]. When compared to classical turbo codes which uses redundant tail bits to force the encoder to all zero state, tail biting technique in double binary turbo codes brings an advantage due to the increase in spectral efficiency. However, in order to provide the circular behavior of the code and to determine the initial state for the given data stream, a pre-encoding procedure is

necessary. This causes the encoder scheme of the double binary codes to be more complex than the encoder scheme of the classical turbo codes.

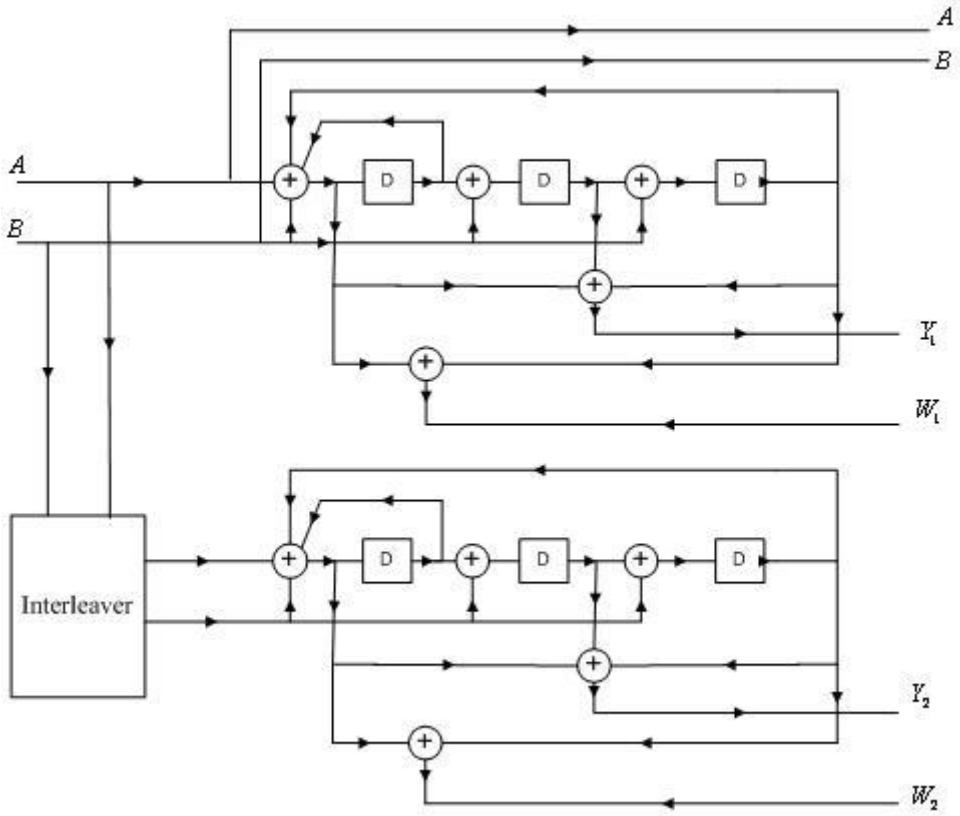

Figure 2.4 Double Binary Turbo Encoder

2.2.2 Interleaver Structure

In turbo codes, interleaver structure has a big effect on the noise performance of the code. In [3] it is stated that two level permutation-inter symbol and intra symbol permutation- helps obtaining large minimum distances and better error correction performance compared to classical turbo codes. Two steps followed in the interleaving procedure are:

1. for j = 0,1,2 ....(N-1)

If (j mod 2 = 1)

Then, (B,A) = (A,B) (switch the couple)

2. For j = 0,1,2.....(N-1)

Switch j mod 4:

Case 0 : $P(j) = (P0 \times j + 1)_{\text{mod}\, N}$

Case 1 : $P(j) = (P0 \times j + 1 + N/2 + P1)_{\text{mod}\, N}$

Case 2 : $P(j) = (P0 \times j + 1 + P2)_{\text{mod}\, N}$

Case 3 : $P(j) = (P0 \times j + 1 + N/2 + P3)_{\text{mod}\, N}$

where Interleaved Vector (j) = Original Vector (P(j)) and N is the block size, P0, P1, P2, P3 are the parameters defined in standards for the different block sizes [7]. In this thesis Double Binary Turbo Codes are implemented according to the IEEE 802.16 standard, so P0, P1, P2, P3 are picked according to the table given in the standard [5].

2.2.3 Sub-block Interleaver Structure

Sub-block interleaving process takes place on systematic bits A, B and Parity bits Y1, W1, Y2, W2 which are the outputs of the encoder. Addresses of the bits are calculated according to the formula given as:

$$T_k = 2^m (k \bmod j) + BRO_m(\lfloor k / j \rfloor)$$

where T_k is the output address, m and j are standard and block size dependent parameters[7].

2.2.4 Puncturing

After sub-block interleaving, puncturing is performed on parity bits according to a given formulation defined in the standards. Puncturing enables increasing the code rate from 1/3 to other rates defined in the standards. Table 2.1 is the puncturing pattern defined in IEEE 802.16 to obtain code rates 1/2, 2/3 and 3/4.

Rate $R_n/(R_n+1)$	Y					
	0	1	2	3	4	5
1/2	1	1				
2/3	1	0	1	0		
3/4	1	0	0	1	0	0

Table 2.1 Double Binary Turbo Code Puncturing Patterns

In each case, systematic bits are sent without deleting any information. For example, to obtain a code rate of 1/2; A, B together with Y1, Y2 blocks are modulated and sent through the channel. For a code rate of 2/3, bits with odd indexes are removed from Y1 and Y2.

De-puncturing is the reverse operation of puncturing and takes place after demodulation. In this case, according to the code rate specified, the received data is padded with zeros to obtain the natural code rate 1/3 which will be used by the iterative decoder.

2.2.5 Double Binary Turbo Decoder

The decoder involves two Soft Input Soft Output (SISO) decoders working iteratively as shown in Figure 2.5.

Decoder 1 calculates extrinsic information denoted as $\Lambda_1(A_k, B_k)$ by making use of LLR of systematic bits, LLR of parity bits and de-interleaved extrinsic information produced by Decoder 2. Decoder 2 uses interleaved LLR of systematic bits, LLR of parity bits and interleaved extrinsic output of Decoder 1. Iteration number generally changes from 2 to 8 depending on the required BER and speed. During iterations, inputs to the decoders (LLR of A, B, Y1, W1, Y2 and W2) are kept constant and only extrinsic information is passed between the decoders. As it will be shown in the Chapter 3, increasing the number of

iterations results in better BER performance at the cost of longer decoding time causing the decoding rate to decrease. In order to obtain a reasonable BER while keeping the decoding time as low as possible, a stopping criterion should be defined.

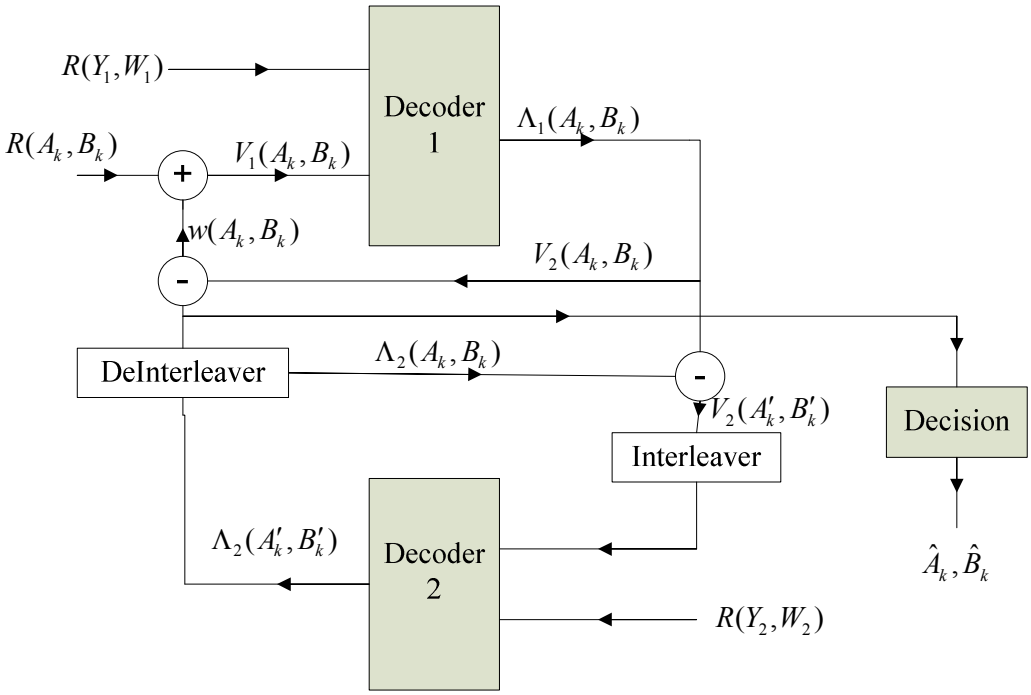

Figure 2.5 Double Binary Turbo Decoder

2.2.6 Decoder Algorithm

Considering one LLR value of a posteriori probabilities as in the case of classical turbo codes is not enough for double binary turbo codes. Instead a modified MAP algorithm or BCJR algorithm, in which three LLRs $L_1 = \ln\left(\frac{P(u_k = (01) \mid y)}{P(u_k = (00) \mid y)}\right)$, $L_2 = \ln\left(\frac{P(u_k = (10) \mid y)}{P(u_k = (00) \mid y)}\right)$, $L_3 = \ln\left(\frac{P(u_k = (11) \mid y)}{P(u_k = (00) \mid y)}\right)$ are calculated, is introduced [10]. This increases the computational complexity of the decoder. However there is no need to compute LLR values, finding four posterior probabilities $P(u_k = (0,0) \mid y)$, $P(u_k = (0,1) \mid y)$, $P(u_k = (1,0) \mid y)$,

$P(u_k = (1,1) \mid y)$ and picking up the maximum of the four values is enough for MAP algorithm [6]. Posteriori possibility of each data pair in log domain is defined as:

$$\ln P(u_k \mid y) = \ln(\sum \exp(\beta_k(s) + \gamma_k(s', s) + \alpha_{k-1}(s')))$$

$$where \begin{cases} \alpha_k(s) = \ln(\sum_{all\, s'} \exp(\gamma_k(s', s) + \alpha_{k-1}(s')) \\ \beta_{k-1}(s') = \ln(\sum_{all\, s} \exp(\gamma_k(s', s) + \beta_k(s) \\ \gamma_i(s', s) = \left(\sum_{l=1}^{m+n} x_{kl} \cdot y_{kl} \right) + \ln P(u_k) \end{cases}$$

where m is the length of systematic bits and n is the length of parity bits. x_{kl} and y_{kl} stands for the received LLR from the demodulator [6].

After MAP decoder operation,

$$\ln P_{out}^{ex}(u_k \mid y) = \ln P(u_k \mid y) - \sum_{l=1}^{m} x_{kl} \cdot y_{kl} - \ln P(u_k)$$

representing 4 log domain extrinsic information is sent to the other decoder.

It is not an easy to implement Log-MAP algorithm in hardware. To simplify the algorithm further and to calculate $MAX(x, y) = \ln(e^x + e^y)$ in an easier way, three main techniques are offered:

Constant Log-MAP:

$$MAX(x, y) = \begin{cases} \max(x, y) + 0, if \mid y - x \mid > T \\ \max(x, y) + C, if \mid y - x \mid \leq T \end{cases}$$

According to [7], this technique gives the best results when C = 0.5 and T = 1.5.

Linear Log-MAP

$$MAX(x,y)=\begin{cases}\max(x,y)+0, if\ |\ y-x\ |>T\\ \max(x,y)+a(|\ y-x\ |-T), if\ |\ y-x\ |\le T\end{cases}$$

The optimum "a" is found to be -0.24904 and "T" to be 2.5068 in [7]. Linear Log-MAP algorithm gives more reliable results however include more computational complexity.

Max-Log-MAP Algorithm

$$MAX(x,y)=\max(x,y)$$

Max-Log-MAP algorithm gives less accurate results when compared to the Log-MAP algorithm itself. However, due to its decreased computational complexity; it is the most preferred algorithm for hardware implementations. In [12], a modified Max-Log-MAP algorithm called Enhanced Max-Log-MAP algorithm is introduced. In this algorithm, by multiplying the extrinsic information with a coefficient smaller than 1, performance of Max-Log-MAP is improved. In [6], it has been shown that Enhanced Max-Log-MAP algorithm achieves the best trade off between performance and computational complexity which is recommended in hardware implementations. In this thesis, Max-Log-MAP is chosen for hardware implementation so this algorithm is explained in further detail.

2.2.7 Max-Log-MAP Algorithm

Max-Log-MAP algorithm includes sweeping the trellis in the forward and backward directions. Each sweep uses a modified version of the Viterbi algorithm in which Add Compare Select operations are carried out by MAX* operator [2]. Performing the forward sweep or backward sweep first does not matter. In either case, metrics calculated in the first sweep are stored in memory and metrics calculated in the second sweep are directly used together with the metrics stored in the first sweep to find final extrinsic information. In [2]

performing the backward sweep first is recommended, because in this case, LLR estimates of the data are produced in the forward sweep and is output in the correct ordering.

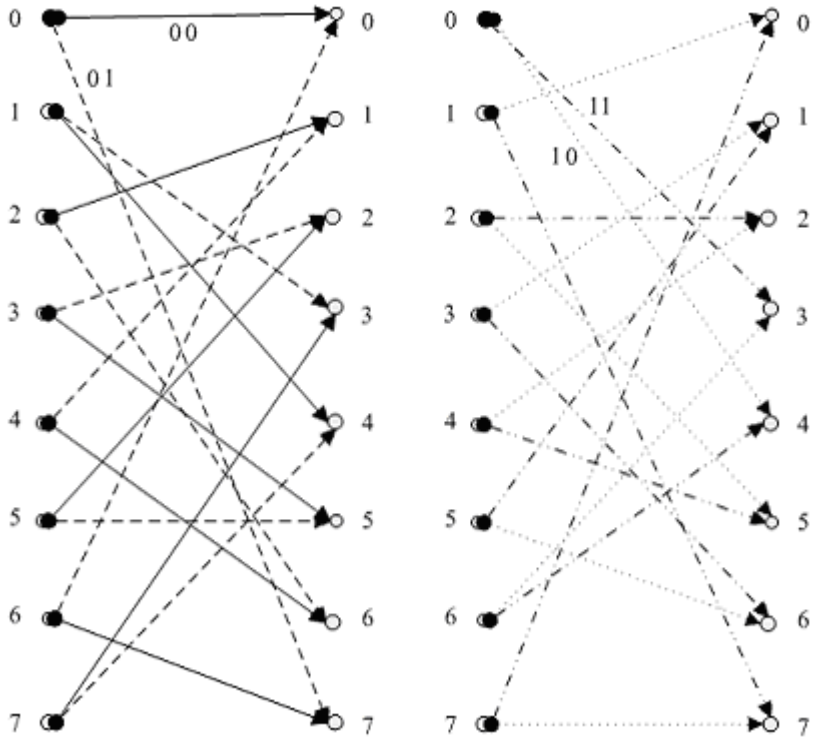

Figure 2.6 Trellises for input AB=00, 01, 10 and 11

During backward recursion, beta metrics are calculated and stored in the memory. Beta metrics represent the probability for different states when considering all the data after time instance k [13] and are calculated according to the following expression:

$$\beta_k(s_k) \cong \max_{s_{k+1} \in B}[\beta_{k+1}(s_{k+1}) + \gamma_{k+1}(s_k \to s_{k+1})]$$

where B is the set of states at time $k+1$ connected to state s_k.

Branch metrics denoted as γ are calculated as:

$$\gamma_k(s_k \to s_{k+1}) = \ln[P(y_k \mid x_k).P(u_k = z)] = \frac{L_c}{2}(x_k^{s_1} y_k^{s_1} + x_k^{s_2} y_k^{s_2} + x_k^{p_1} y_k^{p_1} + x_k^{p_2} y_k^{p_2}) + L_{e,IN}^{(z)}$$

where $z \in \varphi = \{00, 01, 10, 11\}$; u_k is the input symbol consisting of two bits, $P(u_k)$ is a priori probability of u_k, x_k and y_k are transmitted and received codeword associated with u_k [14]. s and p stands for systematic and parity bits respectively. $L_{e,IN}^{(z)}$ is the extrinsic information received from the other SISO decoder. L_c is equal to $2/\sigma^2$ where σ^2 stands for the noise variance of the AWGN channel and generally set to a constant value since turbo decoding based on the Max-Log-MAP algorithm is independent of SNR[14].

During forward recursion, alpha metrics are calculated and without storing in the memory, they are used together with beta metrics to produce extrinsic information for the other SISO decoder. Alpha metrics are calculated as:

$$\alpha_k(s_k) \cong \max_{s_{k-1} \in A}[\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1} \to s_k)]$$

where A is the set of states at time $k-1$ connected to state s_k.

LLR(extrinsic information) calculations are:

$$\Lambda_k^{(z)} \cong \max_{(s_k \to s_{k+1}, z)}[\alpha_k(s_k) + \gamma_{k+1}(s_k \to s_{k+1}) + \beta_{k+1}(s_{k+1})] - \max_{(s_k \to s_{k+1}, 00)}[\alpha_k(s_k) + \gamma_{k+1}(s_k \to s_{k+1}) + \beta_{k+1}(s_{k+1})]$$

where $z \in \varphi = \{01, 10, 11\}$, s_k is the state of the encoder at time k.

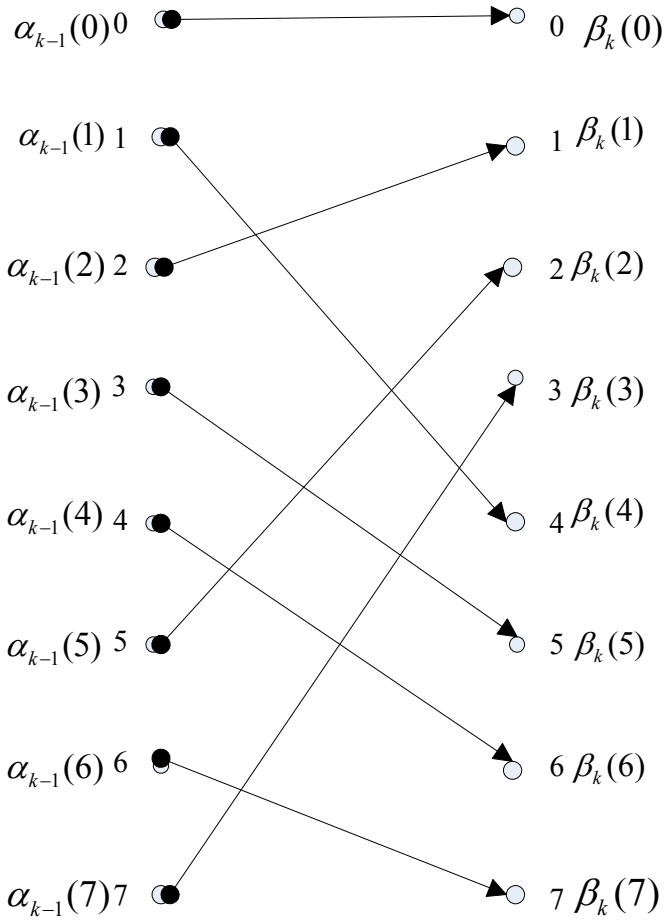

Figure 2.7 Trellis for calculation of extrinsic information when AB=00

It is possible to implement Max-Log-MAP algorithm in two ways; MAP decoding is applied to the whole data block or a large block is split into several windows and MAP decoding is applied to each window separately which is called sliding window technique [15]. The sliding window technique is effective on reducing the memory size required to store metrics [14]. In this technique, forward metrics are calculated before backward metrics are ready, so a dummy calculation is performed for the backward metrics to obtain reliable initial values. Since the dummy calculations do not reflect the actual backward metric values, the performance of the decoder is degraded. The performance gets better as the window size is increased. Instead of dummy calculation another method is introduced in [14] based on border metric encoding. In this method, for each

window, the final backward metric is stored in the border metric memory and used in the next iteration as initial values for the new metrics [14]. There is performance degradation when compared to dummy calculation method, but degradation disappears as the number of iterations increases. By applying the energy efficient turbo decoding method based on border metric encoding, the size of branch memory is reduced by half and the dummy calculation causing computational complexity is removed. [14].

Initiation of forward and backward metrics has a big effect on the performance of the turbo codes. As explained in Section 2.2.1, in the classical turbo code, trellis starts and ends at all zero state so forward and backward metrics can be initiated as follows [6] :

$A_0(S_0 = 0) = 0$
$B_N(S_N = 0) = 0$
$A_0(S_0 = s) = -\infty$
$B_N(S_N = s) = -\infty$ for all s $\neq 0$

For double binary turbo codes, since circular RSC codes are used, the decoder doesn't have any information about the initial and final state of the trellis. Several methods are offered to solve this problem. One solution is to include a pre-decoder to estimate the initial state of the trellis and use through the remaining iterations [8][9]. However this method increases both the computational complexity and latency [6]. In [6] a new method called "feedback" is introduced. Forward and backward metrics are initially set to zero but final metric values are stored to be used as initial values for the metrics in the following iterations. This method does not add any computational complexity or latency but requires the final metric values to be memorized [6]. This method is shown in Figure 2.8.

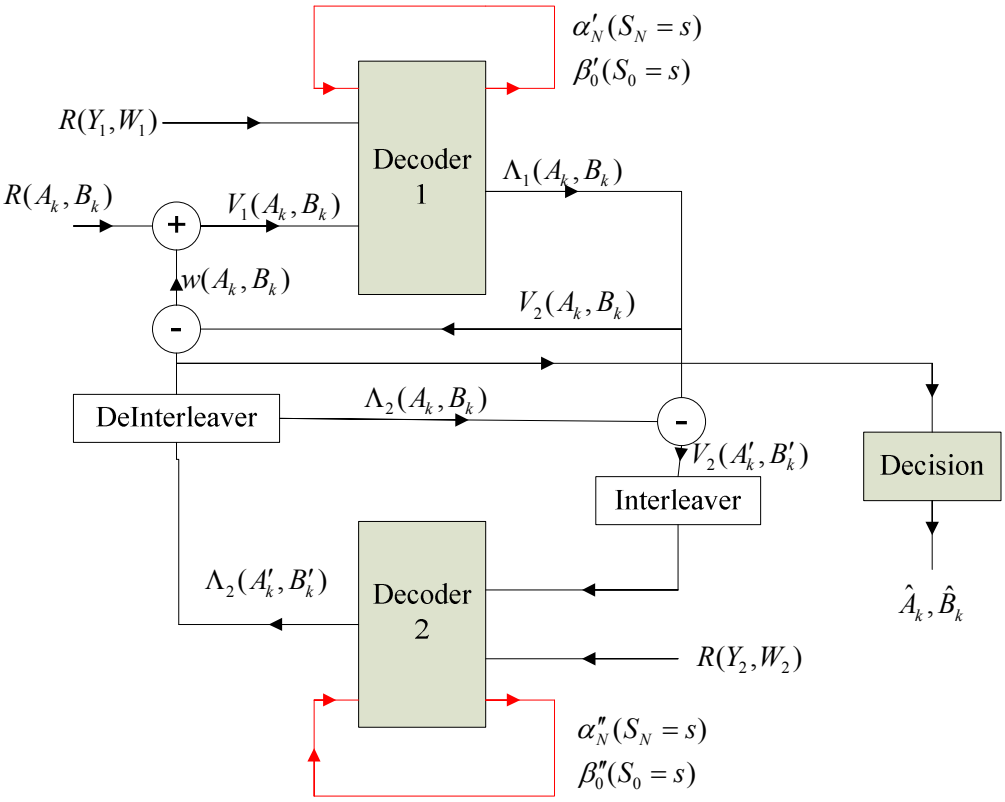

Figure 2.8 Double Binary Turbo Decoder with Feedback

In this case

$$\alpha_0(S_0 = s) = \alpha'_N(S_N = s)$$
$$\beta_N(S_N = s) = \beta'_0(S_0 = s)$$

For the first few iterations, the performance of the algorithm is worse when compared to pre-decoder method but it gets better as the number of iterations increases [6].

Chapter 3

Simulation Results for Double-Binary Turbo Codes

Using MATLAB environment, different simulations are carried out in order to analyze the performance of the Double Binary Turbo Code under different circumstances explained in Chapter 2. Using MATLAB function “rand”, random data is generated for different block sizes. Two random data streams are then encoded by using “encode” function written according to the information given in Chapter 2. Interleaving process is applied by using the parameters given in [5]. “SubBlockInterleaver” function is also written according to the specifications in [5] and applied to the output of “encode” function. Code rate higher than 1/3 is obtained by making use of the puncturing pattern defined in [5]. IEEE defines parameters for three types of modulations: QPSK, 16-QAM and 64 QAM. For modulation, “pskmod” function of MATLAB are used with relevant parameters and passed through the channel defined by “awgn” function of MATLAB which accepts SNR values as parameter. The output of AWGN function is fed to “pskdemod” function of which parameters are “pi/4” for phase offset, “binary” for symbol order, “bit” for decision type and “llr” for decision type. After passing through de-puncturing and sub-block de-interleaving functions, data is given as input to the decoders implemented in “SISO” functions. Output of one SISO function is passed to the other SISO function as input and progress continues until specified iteration number is reached. For each simulation, program runs until the number of decoded bits is 960000.

3.1 Effect of Block Size

Simulations are carried out for block size values 240, 480, 960 and 1920. For each simulation code rate is 1/3(no puncturing), modulation type is QPSK and iteration number is 6.

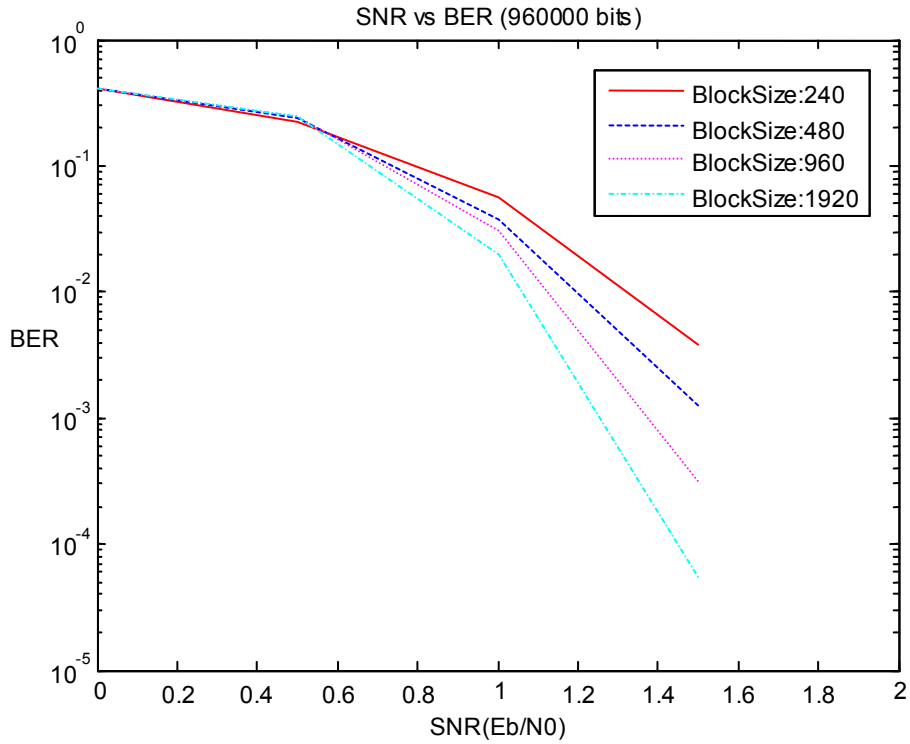

Figure 3.1 Effect of Block Size on the performance of the Turbo code

According to the simulation results, BER performance gets better upon increasing the block size. Using blocks consisting of 1920 bits brings about 0.5 dB performance gain at 0.007 BER. However, large blocks require more memory in hardware, so a block size optimizing the performance and memory requirement can be preferred.

3.2 Effect of Iteration Number

Iteration number has a significant effect on the performance of the decoder. As explained in Chapter 2, pre-decoder method or feedback method is used for estimation of the circular state. According to [6], when feedback method is used, the number of iteration becomes more important. To observe the effect of iteration number for both "feedback" case and "pre-decoder" case, two different simulations are carried out. For the simulation in Figure 3.2, code rate is 1/3 (no puncturing), modulation type is QPSK and block size is 480.

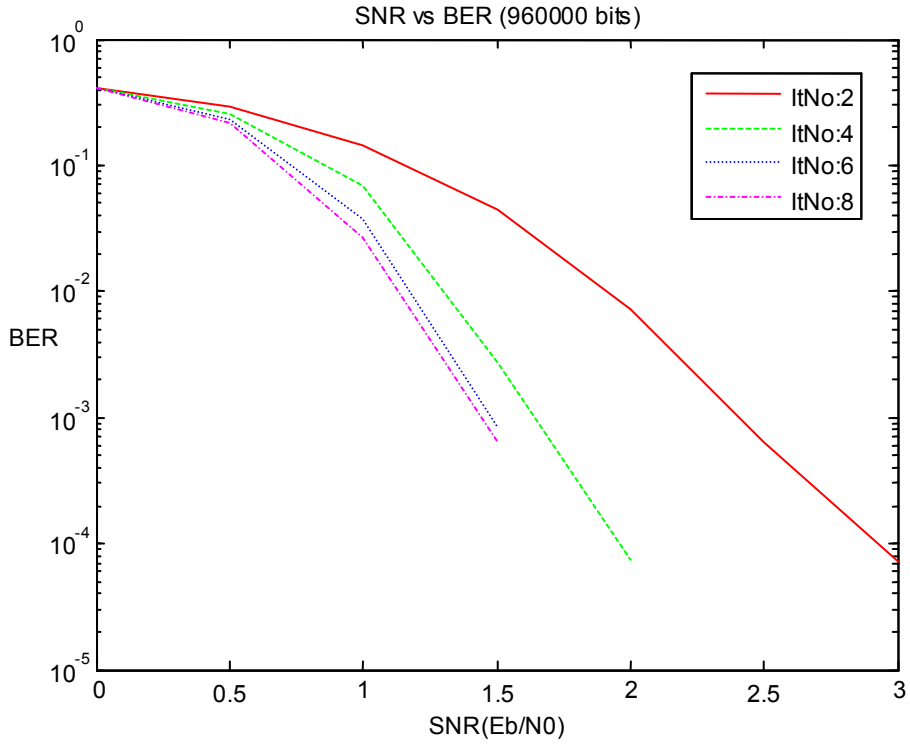

Figure 3.2 Effect of iteration numbers when pre-decoder method is used

When pre-decoder is used for initial metric estimation, iterating 6 or 8 times instead of 2 brings about 1 dB gain at BER of 10^{-3}. As it is seen in Figure 3.2, at BER of 10^{-3}, the difference between 2 iterations and 4 iterations is about 0.8 dB. On the other hand the difference between 4 iterations and 6 iterations is about 0.2 dB at BER of 10^{-3}. This indicates that number of iterations does not

affect the performance linearly. As the number of iterations increase, the improvement on BER performance decreases.

Simulation results when feedback method is used for initial metric estimation are shown in Figure 3.3. For the simulation in Figure 3.3, code rate is 1/3 (no puncturing), modulation type is QPSK and block size is 480.

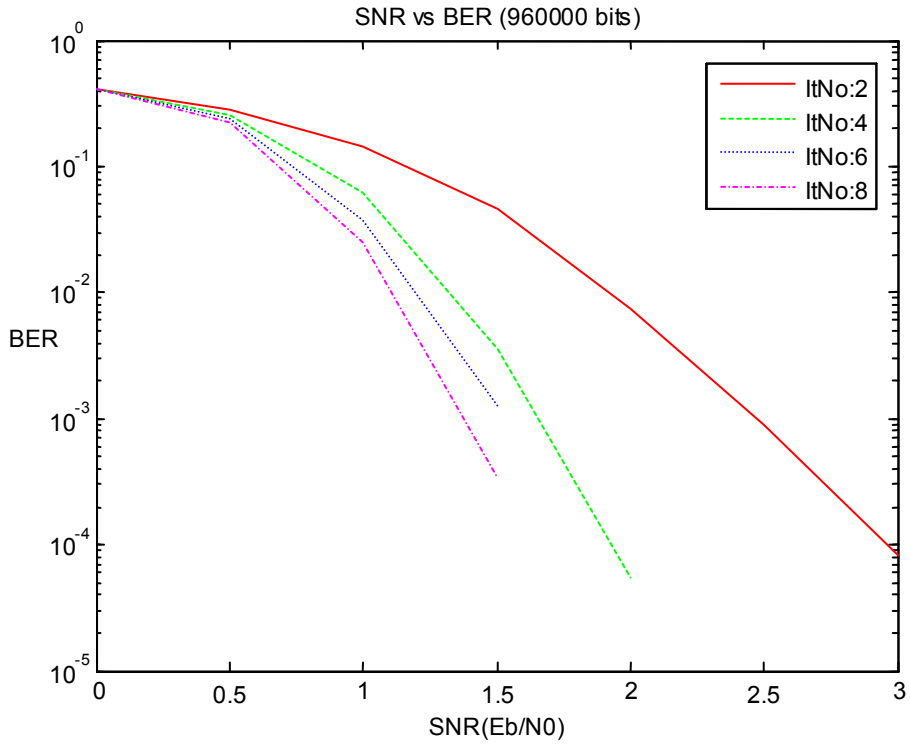

Figure 3.3 Effect of iteration number when feedback method is used

When feedback method is used for initial metric estimation, iterating 6 or 8 times instead of 2 brings more than 1 dB gain at BER of 10^{-3}. At 10^{-3} BER, performance difference between 2 iterations and 4 iterations is about 0.9 dB. On the other hand performance difference between 4 iterations and 6 iterations or 6 iterations and 8 iterations is about 0.1 dB at 10^{-3} BER. In the case of feedback method, increasing the number of iterations improves the performance more when compared to pre-decoder case.

Another simulation is carried out to observe the effect of iteration number when code rate is different than 1/3. For the simulation in Figure 3.3, code rate is 1/2, modulation type is QPSK and block size is 480.

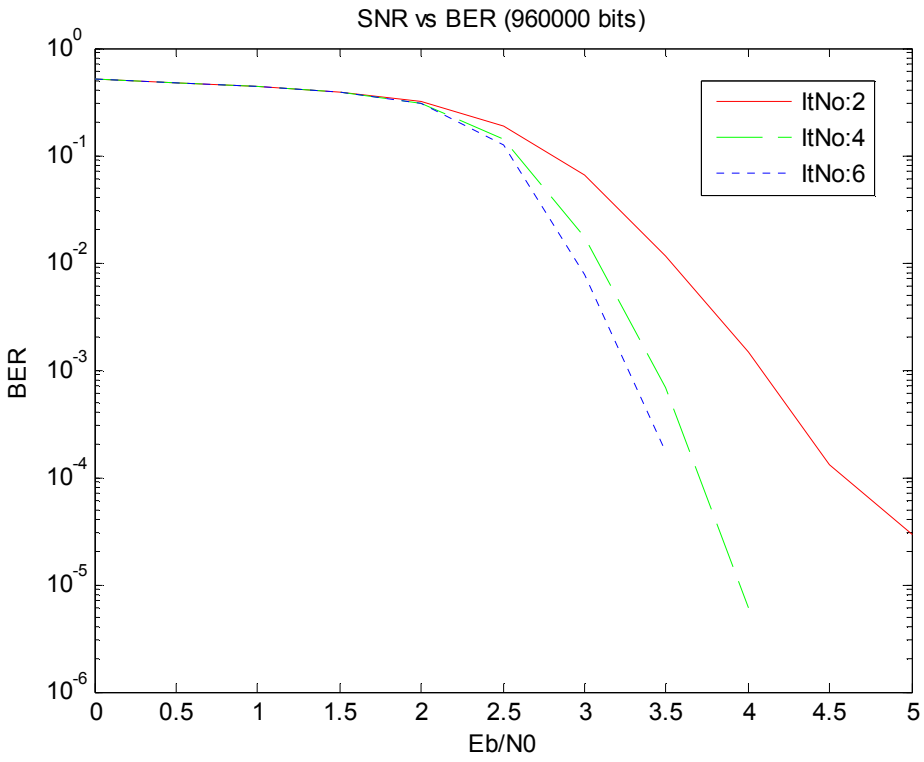

Figure 3.4 Effect of iteration number when code rate is 1/2

Comparing Figure 3.3 and 3.4, it can be concluded that effect of iteration number is similar for a code rate of 1/3 (no puncturing) and 1/2.

Simulation results indicate that increasing iteration number improves the BER performance. However high number of iteration means latency and results in low decoding rate. Keeping in mind that the amount of improvement on the BER performance decreases after 4 iterations; ideal number for iteration can be chosen as 6 or 8 depending on the BER requirement of the application.

3.3 Effect of Pre-Decoder and Feedback Methods

Number of iterations plays a significant role on the performance of the Pre-Decoder and Feedback methods. For this reason, pre-decoder and feedback methods are compared for iteration number 2 and iteration number 6. Code rate is 1/3(no puncturing), modulation type is QPSK and block size is 480 for this simulation.

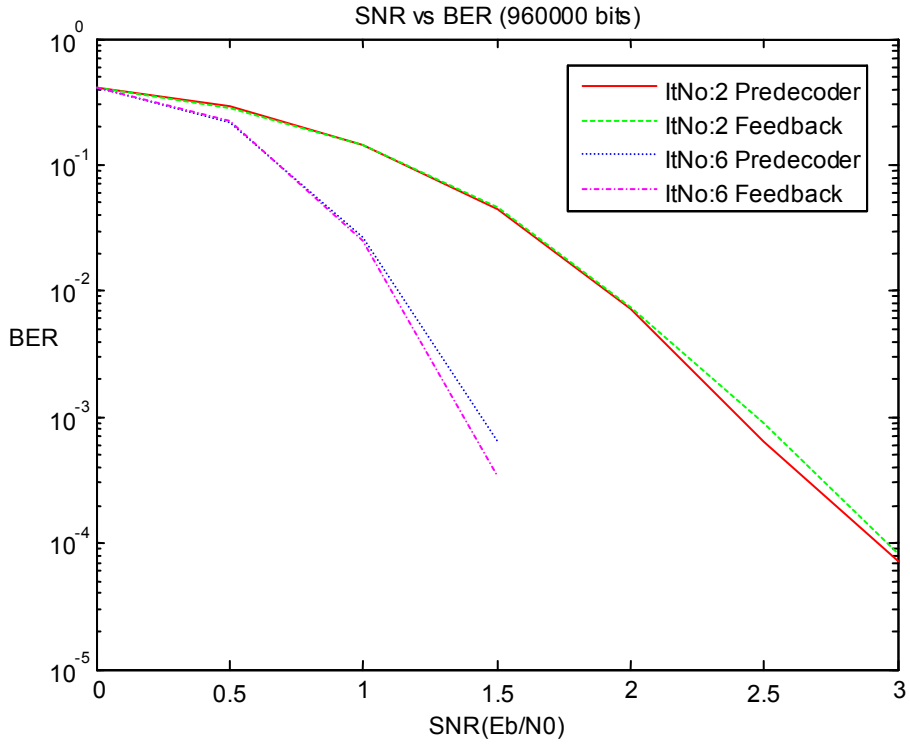

Figure 3.5 Effect of using feedback techniques and pre-decoder techniques

As it is seen in Figure 3.5 when iteration number is 2, pre-decoder method performs slightly better but when iteration number is 6, performance of feedback method supersedes.

Effect of pre-decoder and feedback methods are compared when code rate is different than 1/3 (no puncturing). For the simulation in Figure 3.6, code rate is 1/2, modulation type is QPSK and block size is 480. Simulation is carried out for 2 iterations and 6 iterations.

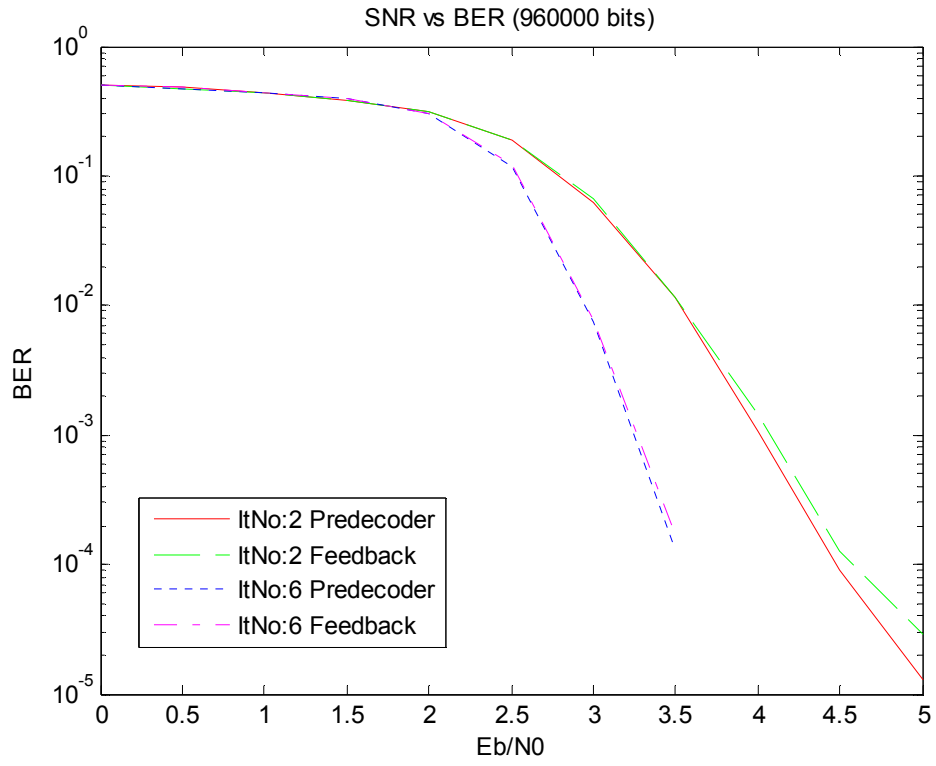

Figure 3.6 Effect of using feedback techniques and pre-decoder techniques when code rate is 1/2

As it is seen in Figure 3.6, for iteration number 6, performance of pre-decoder and feedback methods are nearly the same while in Figure 3.5 the difference is more significant. Except this slight difference between Figure 3.5 and Figure 3.6 it can be concluded that effect of using pre-decoder and feedback methods is similar for code rates 1/3 and 1/2.

According to simulation results there is not a big performance difference between pre-decoder and feedback methods especially at high number of

iterations. Besides, feedback method brings advantage in terms of computational complexity and decoding rate of the decoder.

3.4 Effect of Enhanced Max-Log-MAP

To observe the effect of scaling the extrinsic information by a constant, namely Enhanced Max-Log-MAP algorithm, simulation is carried out for which code rate is 1/3, modulation type is QPSK, iteration number is 6 and block size is 480. Scaling constant is 0.75 for the simulations.

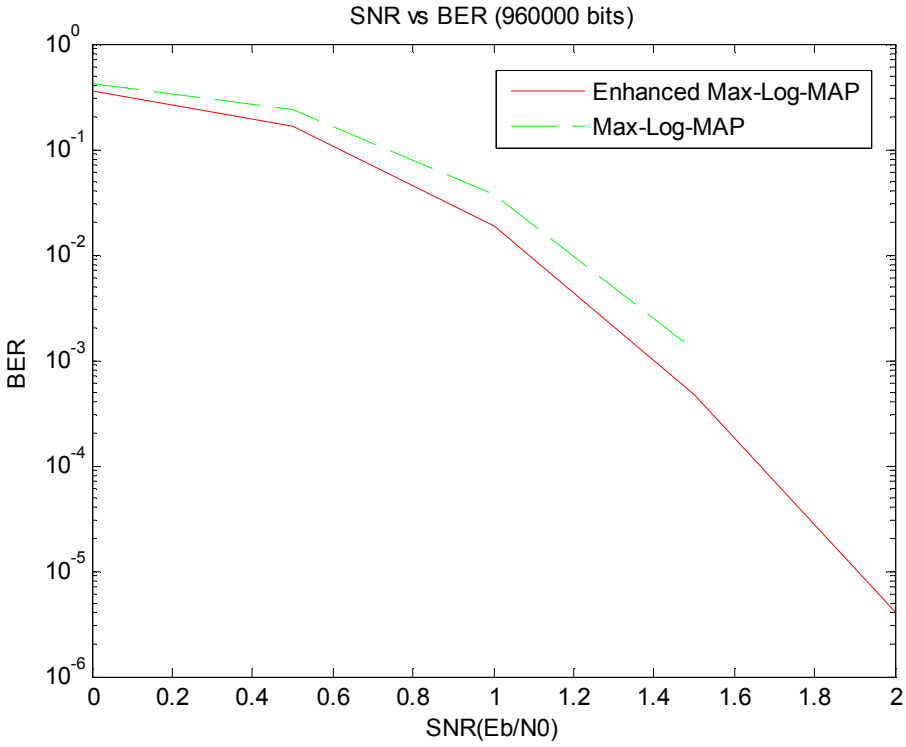

Figure 3.7 Effect of Using Enhanced Max-Log-MAP algorithm

Figure 3.7 indicates that Enhanced Max-Log MAP algorithm improves the BER performance about 0.1 dB at BER of 10^{-3}. This method does not increase the computational complexity much because it requires the multiplication of the extrinsic values by 0.75 which can easily be implemented in hardware.

Chapter 4

Hardware Implementation of Turbo Decoder

Based on the results obtained through MATLAB simulations, a double binary turbo decoder supporting feedback method and Max-Log-MAP algorithm is implemented on an FPGA board. The code is written in VHDL and XC4VFX12-FF668-10 Virtex4 FPGA on Xilinx ML 403 board is used as target system. The code is developed by making use of Xilinx 9.2i ISE, and XST is chosen as synthesis tool.

De-puncturing and sub-block de-interleaving parts are not included in the implementation. These processes are assumed to be performed in software on the processor.

4.1 Architecture

Main modules in the hardware implementation are Controller module, Data selector module, Beta module, Alpha&LLR module and Serial Channel module. Figure 4.1 depicts the interaction of these modules.

LLR values of received systematic bits and parity bits to be processed are assumed to be loaded to the block RAMs both in the natural order and in interleaved order. These values are the de-punctured and de-interleaved soft outputs of the demodulator block. In other words, this implementation

corresponds to the “Decoding” part in Figure 2.3 and changes in the code rate does not affect the implementation.

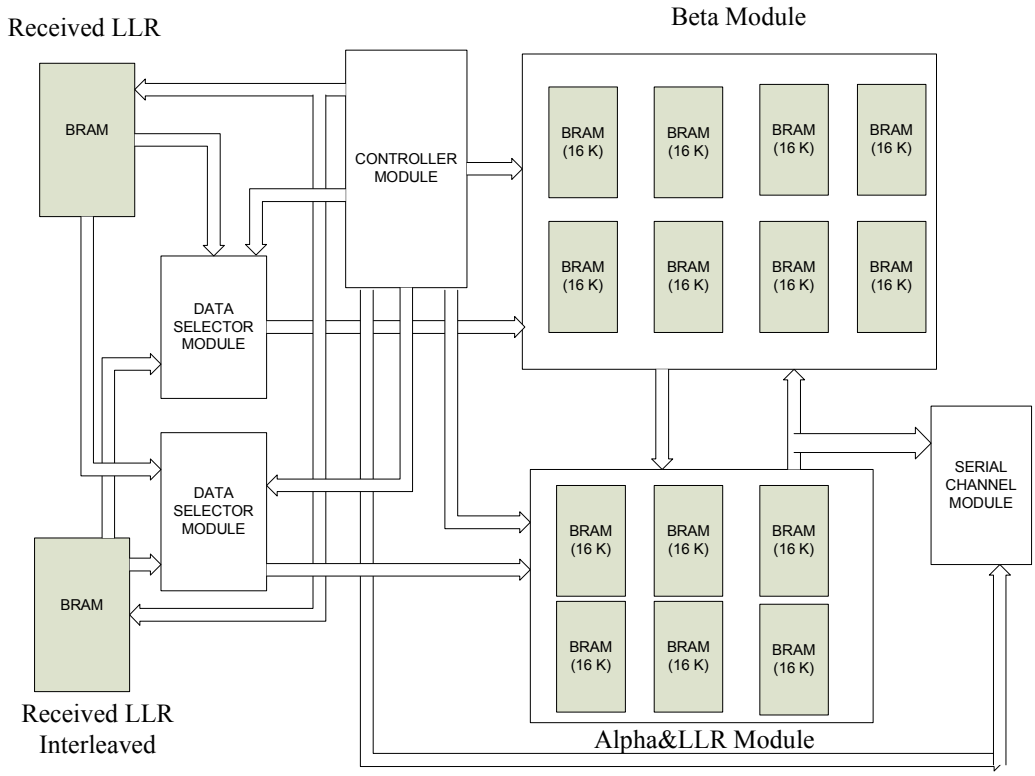

Figure 4.1 Overall architecture of the implemented Turbo Decoder

Controller module has interaction with all other modules in the architecture as it is seen in Figure 4.1. Controller module is responsible for managing other modules by determining the inputs and outputs of the modules according to the state of the decoder. Number of iterations and block size is also set in the controller module.

Beta and Alpha&LLR modules correspond to backward sweep and forward sweep respectively. One backward sweep and one forward sweep together with LLR calculation represents a half-cycle of the turbo decoder. In this implementation, backward sweep of trellis is performed first and followed by forward sweep together with LLR calculation in parallel. Due to the iterative nature of turbo decoder, second half iteration has to wait for the first half

iteration to be completed. This means that single Beta and single Alpha&LLR module is enough for a turbo decoder implementation if proper input is supplied to the modules. Using single modules for forward and backward metric calculation, the area required to implement a turbo decoder is minimized.

Beta module's main task is to calculate backward metrics using the input data fed from its own data selector module and Alpha&LLR module. Calculated metrics are stored in the addresses specified by the controller module. The outputs of the Beta module are connected to the Alpha&LLR module and updated according to the addresses specified by the controller module.

Alpha&LLR module calculates forward metrics employing the input data fed from its own data selector module and extrinsic information produced by itself in the previous half iteration. Forward metrics are not stored in the memory and included directly to the calculations of current extrinsic information together with the metrics produced by the Beta module. Extrinsic information is stored in the addresses determined according to the state and address information supplied by the controller module. In other words, for the first half iteration, Alpha&LLR uses the addresses specified by the controller module directly but in the second half iteration, uses those addresses to calculate the interleaved addresses. The outputs of Alpha&LLR module are utilized by both itself and by the Beta module. This module updates its output according to the addresses and state information fed by the controller module. For example, if the decoder runs for the second half iteration, data stored in the interleaved addresses is supplied to the Beta module although the addresses fed by the controller are in natural order.

Due to the data dependency between Alpha and Beta modules, they have to work sequentially. Controller decides on which module to run at each state. However, to improve the decoding speed, Alpha and Beta modules should work in parallel. This is achieved by providing another data block to the decoder and saving the metrics belonging to the new data block to a different location in the

memory of each module. In this scheme, while Beta module processes for the first data block, Alpha&LLR Module processes for the second data block and vice versa. For each module, Controller module decides on the data blocks to be processed at each state and specifies the addresses to be used. Parallel processing of two different data blocks doubles the decoding speed at the cost of larger memory requirement. Since we focus primarily on the speed of the decoder, memory disadvantage of parallel processing is ignored.

4.2 Modules in Detail

Inputs and outputs of the modules in the decoder architecture are explained in detail in this section. All modules operates at the rising edge of the clock and fixed point operations are carried out for the data, metrics and extrinsic information in which fractional part is 3 bit width.

4.2.1 Data Selector Module

Main task of Data Selector module is to selectively direct the proper data to the module connected to its outputs. Inputs and outputs of the module are shown in Figure 4.2.

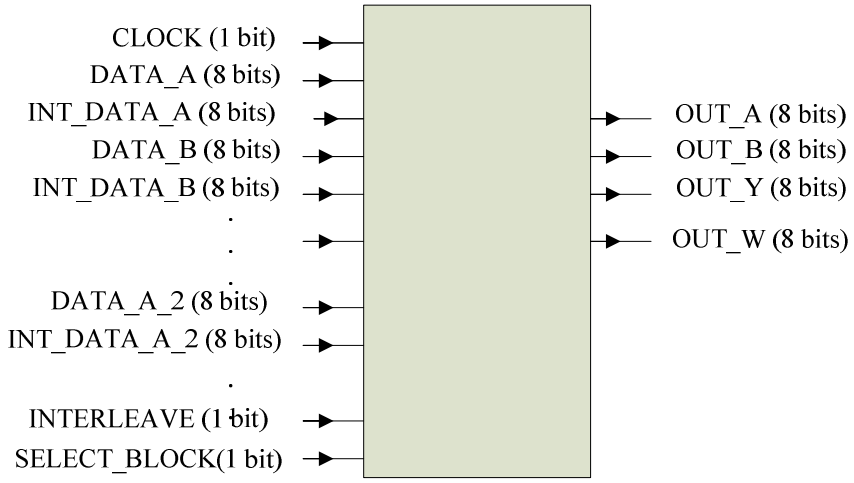

Figure 4.2 Data Selector module inputs and outputs

Beta and Alpha&LLR modules work on different data blocks at the same time hence there are two separate Data Selector modules for each module.

Inputs of the Data Selector module are connected directly to the outputs of the block RAMs in which LLR values of received systematic bits and parity bits are stored. Proposed turbo decoder processes two different data blocks in parallel hence data related to two different blocks are fed to the module doubling the number of inputs. Inputs denoted as DATA_A, INT_DATA_A, DATA_B, INT_DATA_B, DATA_Y, INT_DATA_Y, DATA_W, INT_DATA_W corresponds to the received LLR of the systematic and parity bits of the first data block and DATA_A_2, INT_DATA_A_2, DATA_B_2, INT_DATA_B_2, DATA_Y_2, INT_DATA_Y_2, DATA_W_2, INT_DATA_W_2 are the received LLR of the systematic bits and parity bits of the second data block. INT_DATA_Y and INT_DATA_W are the received LLR values of the bits encoded by the lower encoder in Figure 2.4 and transmitted through the channel while INT_DATA_A and INT_DATA_B are the interleaved DATA_A and DATA_B respectively at the decoder side. Inputs INTERLEAVE and SELECT_BLOCK of the module are set by the Controller module according to the state of the decoder. For example if a module will process for the first data block and in the second half iteration, then INTERLEAVE signal is set to high and SELECT_BLOCK signal is set to low. In this case, the input signals will be directed as follows:

INT_DATA_A => OUT_A
INT_DATA_B => OUT_B
INT_DATA_Y => OUT_Y
INT_DATA_W => OUT_W

4.2.2 Beta Module

Beta module's responsibility is to calculate, store and emit backward metrics. Inputs and outputs of the module are shown in Figure 4.3.

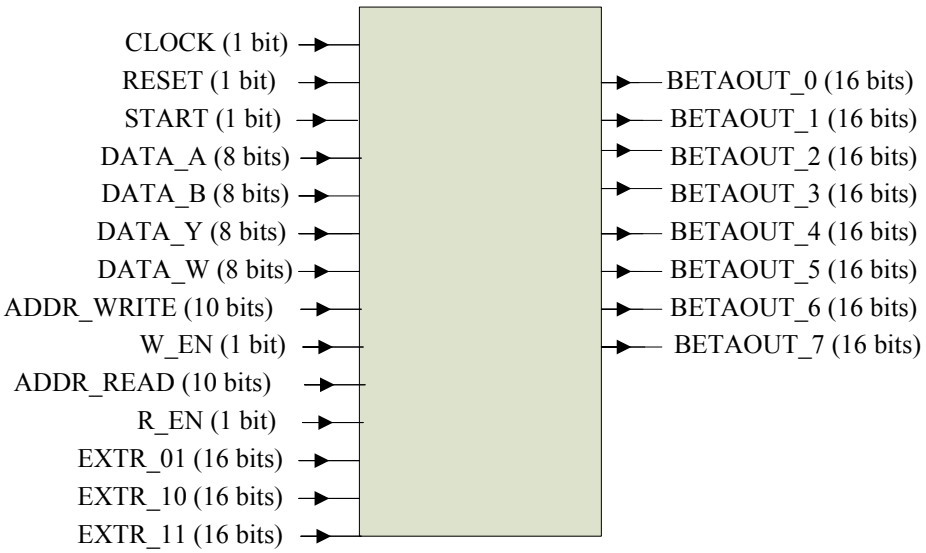

Figure 4.3 Beta Module inputs and outputs

Inputs DATA_A, DATA_B, DATA_Y and DATA_W are directly connected to the outputs of the Data Selector module which is reserved for the use of Beta Module. Other inputs are determined by the Controller module. When the input denoted as START is set to high by the controller module, Beta module begins to calculate the backward metrics using DATA_A, DATA_B, DATA_Y and DATA_W together with EXTR_01, EXTR_10 and EXTR_11 which are the extrinsic values calculated by the Alpha&LLR module in the previous half iteration. For each different time instance in other words for each different data pair, 8 beta metrics are calculated corresponding to 8 different states in the Trellis. A normalization operation is performed before saving the metrics in the memory. This is done by subtracting the first Beta metric –metric for state 0- from other Beta metrics. Each metric is stored in a separate dual port block RAM hence there are 8 block RAMs inside the module. Actually 7 block RAMs

are needed since Beta metrics for state 0 will always be zero because of the normalization, however 8 block RAMs are used to obtain a flexible design. WEN and REN signals enable writing and reading to the RAMs. Last half of each block RAM is reserved for saving the metrics of the second data block. Calculated metrics are stored in the memory locations specified by ADDR_WRITE signal which is set by the Controller module. Outputs of the module which are connected to the inputs of the Alpha&LLR module are beta metrics saved in the memory locations determined by ADDR_READ signal.

Due to the parallel processing of Alpha and Beta modules, block RAMs are written and read at the same time. When Beta module is calculating and writing to the block RAMs, Alpha&LLR module is reading the metrics of the other data block stored in the previous half iteration. Dual port block RAMs in the module enables concurrent read and write operations. For each RAM, one port is assigned for reading and one port is assigned for writing.

Feedback method explained in Chapter 2 is implemented for the initialization of the Beta metrics. Final Beta metrics for a data block are kept to be used as initial values for the same half iteration of the data block.

It takes two cycles for the Beta module to calculate and store the metrics to the RAM when clock frequency is 100 MHz.

4.2.3 Alpha&LLR Module

Main task of Alpha&LLR module is to calculate forward metrics and produce extrinsic information by making use of backward metrics and forward metrics. Inputs and outputs of the module are shown in Figure 4.4. This module is the most complex module and occupies the largest area on the FPGA.

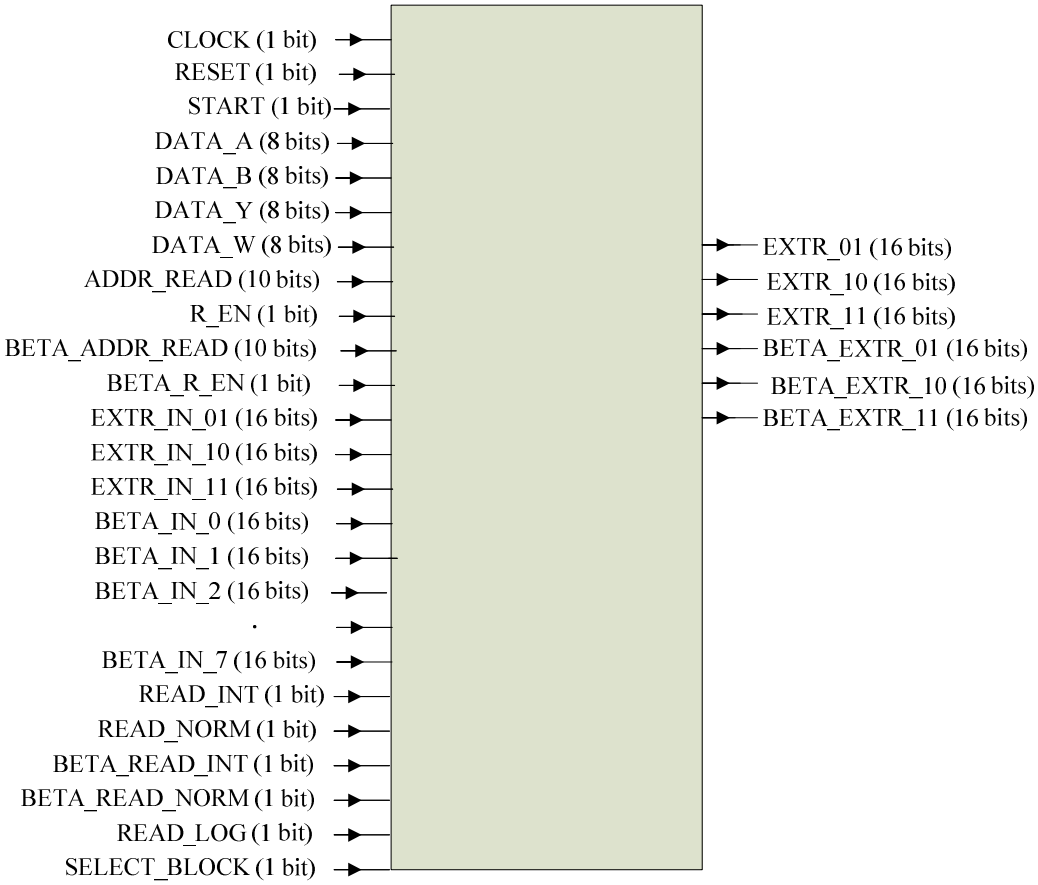

Figure 4.4 Alpha&LLR module inputs and outputs

Inputs DATA_A, DATA_B, DATA_Y, DATA_W are directly connected to the outputs of the Data Selector module assigned for the Alpha&LLR module. Beta metrics are taken from the Beta module through the inputs BETA_IN_0, BETA_IN_1 ..... BETA_IN_7. EXTR_IN_01, EXTR_IN_10, EXTR_IN_11 are connected to EXTR_01, EXTR_10, EXTR_11 respectively which are the outputs of the module. Other inputs are set by the Controller Module.

When START signal is set to high by the Controller module, it begins calculating forward metrics using the DATA_A, DATA_B, DATA_Y, DATA_W signals and extrinsic information(EXTR_IN_01, EXTR_IN_10, EXTR_IN_11) calculated in the previous half iteration of the related data block.

Forward metrics are not stored in the memory and together with BETA_IN_0, BETA_IN_1 ..... BETA_IN_7, included to the calculations carried out to produce extrinsic information. Extrinsic information is saved in the memory locations of which addresses are calculated by the module itself. The module calculates the addresses to write according to the inputs SELECT_BLOCK, READ_INT and READ_NORM. If READ_INT is set to high, this means that the module is operating in the second half iteration of the data block specified by SELECT_BLOCK. In this case extrinsic information is stored in de-interleaved addresses for the Beta module to be able to read them in normal order in the following half iteration.

In the second half iteration of decoding process of a data block, extrinsic information produced in the first iteration should be read in interleaved order. If READ_INT is high, it reads the extrinsic information produced in the prior half iteration of the block specified by SELECT_BLOCK in the interleaved order. In this module, an interleaver is designed to calculate the interleaved addresses for different block sizes. However, this design is not used during tests since the block size is kept constant and the corresponding interleaver addresses are embedded in the code. Using the ADDR_READ signal and the table embedded in the code, interleaved addresses are found and output is updated accordingly.

SELECT_BLOCK is set to high or low to indicate the module whether it is working on the first data block or second data block respectively. Extrinsic information belonging to the first data block and second data block are stored in the first half and second half of the block RAMs respectively.

Number of block RAMs in the module is 6 although there are 3 types of extrinsic information. The reason is that when Alpha module is in progress it both writes and reads the extrinsic information from the RAMs so two ports of each RAM is occupied by the Alpha module. However, Beta module is also in progress on the other data block. Hence RAM number is doubled and 3 RAMs

are reserved for the usage of Beta Modules. As it is seen in Figure 4.4, there are six outputs of the module: EXTR_01, EXTR_10, and EXTR_11 stands for the extrinsic information to be used by the module itself and BETA_EXTR_01, BETA_EXTR_10, BETA_EXTR_11 stands for the extrinsic information to be used by the Beta module. BETA_READ_INT and BETA_READ_NORM control the read address of the extrinsic information to be used by the Beta module.

Feedback method explained in Chapter 2 is implemented for the initialization of the Alpha metrics. Final Alpha metrics for a data block are kept to be used as initial values for the same half iteration of the data block.

Calculation and storage of extrinsic information is performed in 2 clock cycles at 100 MHz operating frequency.

4.2.4 Serial Channel Module

A serial channel module operating at a baud rate of 115200 is implemented for test purpose. Input and outputs of the module are shown in Figure 4.5

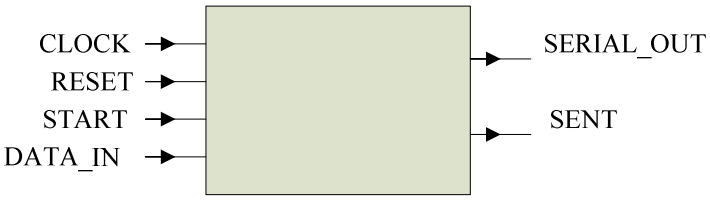

Figure 4.5 Serial Channel Module Inputs and Outputs

Input DATA_IN which is 8 bits width is updated by the Controller module. When START is set to high, module begins to send bits of DATA_IN through SERIAL_OUT at the desired baud rate. Output SENT is set to high upon transmission of one byte to indicate the readiness of the serial channel. Extrinsic

information stored in the Alpha&LLR module is sent through the serial channel at the end of each iteration or at the end of all iterations.

4.3 Test Procedure

Data generated by the MATLAB model is loaded to the Block RAMs manually and the decoding process starts. After specified number of iterations is completed, final extrinsic values are transmitted to the PC through serial channel with a baud rate of 115200. An application developed using Microsoft Visual Studio 6.0 running on the PC collects the data received from the ML403 board into a file and converts the data to a suitable format. File is compared with MATLAB output. Test is carried out for different number of iterations configured in the code and it is observed that hardware results and software results are the same.

4.4 Results

In this section, hardware implementation is evaluated in terms of the resources used on the FPGA and the decoding rate. Results are compared with another FPGA implementation in the literature [13].

4.4.1 FPGA Device Utilization Report

As it is stated at the beginning of Chapter 4, XC4VFX12-FF668-10 Virtex4 FPGA on Xilinx ML 403 board is used as target system. The code is developed by using Xilinx 9.2 ISE and XST is preferred as the synthesis tool. The amount of the resources used for the implementation is depicted in Table 4.1.

	Used	Available
Number of Slice Flip Flops	2992	10944
Number of 4 input LUTs used as logic	7734	
Number of 4 input LUTs used as shift registers	242	10944
Number of Occupied Slices	4866	5472
Number of DCM	1	4
Number of BRAM	22	36

Table 4.1 Device Utilization Report

In this table, BRAMs used to store the data blocks should be excluded since they are not a part of the decoding process. Thus actual number of BRAM is 14; 8 for Beta module and 6 for Alpha&LLR module.

4.4.2 Decoding Rate

The decoder proposed works for a block size of 480; however it can easily be configured to another number less than 480 defined in the IEEE.802.16 standard. For the block size of K, a complete iteration for two different data blocks takes $(4\times K+5)\times 2+(2\times K+3)$ cycles and each cycle takes 10 ns since the operating frequency is 100 MHz. For N iterations, this formula becomes

$$(4\times K+5)\times 2\times N+(2\times K+3).$$

At the end of iterations, $4xK$ bits are decoded; then the decoded data rate per clock cycle is:

$$\frac{4\times K}{(4\times K+5)\times 2\times N+(2\times K+3)}$$

The formula is evaluated for different block size values and the results in Table 4.2 are obtained.

Now assume that a data stream including 2P blocks (P blocks for each stream) are available at the input of the decoder and the blocks are sent to the decoder in such a way that when the decoding of a block is over, immediately new block, to be decoded, is ready. Then the formula becomes

$$\frac{P\times 4\times K}{P\times(4\times K+5)\times 2\times N+(2\times K+3)}$$

and for P >> K the results becomes

$$\frac{P\times 4\times K}{P\times(4\times K+5)\times 2\times N}$$

and the decoding rate becomes as indicated in Table 4.3.

Block Size (K)	2 iterations (Mb/sec)	4 iterations (Mb/sec)	6 iterations (Mb/sec)	8 iterations (Mb/sec)
480	22,16	11,73	8,00	6,00
240	22,10	11,70	7,96	6,00
216	22,09	11,70	7,95	6,00
192	22,07	11,69	7,95	6,00
180	22,06	11,68	7,95	6,00
144	22,03	11,66	7,93	6,00
120	21,99	11,64	7,92	6,00
108	21,96	11,63	7,90	5,99
96	21,93	11,62	7,89	5,98
72	21,83	11,56	7,86	5,96
48	21,65	11,46	7,79	5,90
36	21,46	11,36	7,73	5,86
24	21,10	11,17	7,60	5,76

Table 4.2 Decoding Rate for different block sizes for 2 data blocks

Block Size(K)	2 iterations (Mb/sec)	4 iterations (Mb/sec)	6 iterations (Mb/sec)	8 iterations (Mb/sec)
480	24,95	12,47	8,31	6,24
240	24,87	12,44	8,29	6,22
216	24,86	12,43	8,28	6,21
192	24,84	12,42	8,28	6,20
180	24,83	12,41	8,28	6,20
144	24,78	12,40	8,26	6,20
120	24,74	12,37	8,24	6,18
108	24,71	12,36	8,24	6,18
96	24,68	12,34	8,23	6,17
72	24,57	12,29	8,19	6,14
48	24,36	12,18	8,12	6,09
36	24,16	12,08	8,05	6,04
24	23,76	11,88	7,92	5,94

Table 4.3 Decoding Rate for different block sizes for very large number of data blocks

4.4.3 Comparison

A number of previous researchers implemented Double Binary Turbo Decoder. In most of them, an ASIC has been designed and analyzed. Comparison of a dedicated ASIC for turbo decoding and an FPGA implementation is not suitable both in terms of decoding rate and in terms of area occupied. Another FPGA implementation is performed by the authors of [13] from Linköping University and our implementation is compared with [13].

In [13], an Altera Stratix II FPGA is used and Synplify Pro is used as synthesis tool. In Table 4.4, resource utilizations of two implementations are given.

	Proposed Decoder	Decoder in [13]
Number of Slice Flip Flops	2992	2869
Number of Occupied Slices	4866	7146
Memory	14 BRAM (16Kb each)	57600 bits

Table 4.4 Comparison of the proposed decoder to the decoder in [13]

As Table 4.4 reveals, our implementation occupies less logic cells but more memory on the FPGA. One reason of larger memory requirement is that block size of 480 is also supported in our implementation. In [13], block sizes up to 240 are supported only. Parallel decoding of two different data blocks using only one decoder, which is not available in [13] also doubles the memory required to save metrics.

Table 4.5 are the decoding rates in [13] for different block sizes and when four decoders are working on different data blocks in parallel, at 100 MHz clock frequency.

Blocksize (N)	2 iters, Mb/s	4 iters, Mb/s	6 iters, Mb/s	8 iters, Mb/s
24	93,2	46,6	31,1	23,3
36	95,4	47,7	31,8	23,8
48	96,5	48,2	32,2	24,1
72	97,6	48,8	32,5	24,4
96	98,2	49,1	32,7	24,6
108	98,4	49,2	32,8	24,6
120	98,6	49,3	32,9	24,6
144	98,8	49,4	32,9	24,7
180	99,0	49,5	33,0	24,8
192	99,1	49,5	33,0	24,8
216	99,2	49,6	33,1	24,8
240	99,3	49,6	33,1	24,8

Table 4.5 Decoded Data Rate for four decoders with frequency 100 MHz

Decoding rates in Table 4.5 are nearly 4 times greater than the decoding rates of the proposed turbo decoder given in Table 4.3. In [13] it is stated that decoding rate is linearly dependent to the number of decoders working in parallel; this means that the decoding rate of a single decoder in [13] is nearly equal to the decoding rate of our decoder.

Chapter 5

Conclusions and Future Work

Double Binary Turbo codes which are widely used in today's communication standards such as DVB-RSC and IEEE 802.16 are explored and an efficient double binary Turbo decoder is implemented on an FPGA. The implementation is compared with the previous implementations in the literature.

Double Binary Turbo encoder is parallel concatenation of two double binary RSC codes. The encoder has a circular nature which means that the initial state of the trellis is equal to the final state of the trellis. This brings the advantage of spectral efficiency at the expense of an extra pre-encoder process.

Double Binary Turbo decoder consists of two SISO decoders working iteratively and exchanging the extrinsic information in between. MAP algorithm used in SISO decoders is very important to achieve the best trade-off between performance and computational complexity for an efficient hardware implementation. Different studies are investigated and a MATLAB code is developed to apply the recommendations. According to the results, the best solution is Enhanced Max-Log-MAP algorithm. Another important issue for the decoder is initializing the forward and backward metrics in the algorithm. Due to the circular nature of the encoder, the initial hence the final state of the trellis can not be estimated by the decoder. Two techniques- using a pre-decoder and feedback- to overcome this problem are discussed. Pre-decoder technique provides good performance even in the initial iterations but brings an important computational complexity and decreases the decoding rate. Simulations show

that feedback technique is as good as pre-decoder technique especially when iteration number increases and does not bring much computational complexity. Border metric encoding which is introduced to reduce the memory size and power consumption of the decoder, is also investigated.

A turbo decoder configurable up to a data block size of 480 is implemented in hardware. One SISO decoder together with a dedicated controller is designed. The modules calculating backward metrics, forward metrics and LLR values are used as efficient as possible. Two data blocks are decoded in parallel using a single decoder and a decoding rate of 6.3 Mb/s is achieved for 8 iterations at 100 MHz operating frequency.

As future work, de-puncturing process supporting different code rates changing dynamically should be included to the hardware implementation. Border metric encoding introduced in [14] should be applied in order to decrease the memory used. Although the implementation supports block sizes up to 480 with a proper configuration in the VHDL code, it should be tested whether it works properly when block size changes dynamically. The decoder should be fed with continuous data through Ethernet or etc. to observe the performance of the decoder.

Appendix A

MATLAB Simulation Codes

A.1 Double Binary Turbo Code

```
function [Number,DemodError] =
DuoBinaryTurboCode(Length,ItNo,Noise,ModType,PunctRate)

%Random data is generated
A = round(rand(Length,1));
B = round(rand(Length,1));

%Interleaving
[AI,BI]=interleaver(A,B);

%Encoding
[Y1,W1]=encode(A,B);
[Y2,W2]=encode(AI,BI);

%SubBlockInterleaver
TempDataToSend=SubBlockInterleaver(A,B,Y1,Y2,W1,W2);

%puncturing is performed
DataToSend = Puncture(PunctRate,TempDataToSend);

%Modulation, Noise addition and Demodulation
if ModType==1
    m = modem.pskmod('M', 4, 'PhaseOffset', pi/4,    'SymbolOrder','binary',
'InputType', 'bit');
    Modulated = modulate(m,DataToSend);
    Channel = awgn(Modulated,Noise,'measured');
    h = modem.pskdemod('M', 4, 'PhaseOffset', pi/4,'SymbolOrder', 'binary',
'OutputType', 'bit','DecisionType', 'llr');
    Demodulated = demodulate(h,Channel);
elseif ModType==2
    m = modem.qammod('M', 16, 'PhaseOffset', pi/4, 'SymbolOrder','binary',
'InputType', 'bit');
    Modulated = modulate(m,DataToSend);
    Channel = awgn(Modulated,Noise,'measured');
    h = modem.qamdemod('M', 16, 'PhaseOffset', pi/4,'SymbolOrder', 'binary',
'OutputType', 'bit','DecisionType', 'llr');
    Demodulated = demodulate(h,Channel);
elseif ModType==3
    m = modem.qammod('M', 64, 'PhaseOffset', pi/4, 'SymbolOrder','binary',
'InputType', 'bit');
    Modulated = modulate(m,DataToSend);
    Channel = awgn(Modulated,Noise,'measured');
    h = modem.qamdemod('M', 64, 'PhaseOffset', pi/4,'SymbolOrder', 'binary',
'OutputType', 'bit','DecisionType', 'llr');
    Demodulated = demodulate(h,Channel);
end
Demodulated = Demodulated * (-1);
```

```
DepuncturedData = Depuncture(PunctRate,Demodulated);
[Ar,Br,Y1r,W1r,Y2r,W2r]=SubBlockDeInterleaver(DepuncturedData);
DemodOut = [Ar;Br];
ActualData = [A;B];
[DemodError,R]=biterr((DemodOut>0)+0,ActualData);
%Interleave received LLR of A and B
[ArI,BrI]=interleaver(Ar,Br);
Extrinsic=zeros(3,Length);
%Final alpha and beta metrics for each decoder
AlphaI = zeros(8,1);
BetaI = zeros(8,1);
AlphaO = zeros(8,1);
BetaO = zeros(8,1);
%Iterative decoding
for k=1:ItNo
    %First decoder processes data in natural order
    [Extrinsic1,AlphaI,BetaI]=SISO(Ar,Br,Y1r,W1r,Extrinsic,AlphaI,BetaI);
    ExtrinsicInt=Interleaver_Ext(Extrinsic1);
    %Second decoder processes data in interleaved order
    [Extrinsic2,AlphaO,BetaO]=SISO(ArI,BrI,Y2r,W2r,ExtrinsicInt,AlphaO,BetaO);
    Extrinsic = DeInterleaver_Ext(Extrinsic2);
    %After each full iteration, decision is carried out
    [Out,Number]= Decision(A,B,Extrinsic);
end
```

A.2 Interleaver

```
function [AI,BI] = interleaver(A,B)
% This function interleaves data streams given as A and B using the
% parameters specified in IEEE 802.16 standard

%T holds the block sizes defined in the standard
T = [24 36 48 72 96 108 120 144 180 192 216 240 480 960 1440 1920
2400];
%P holds parameters P0,P1,P2,P3 specified for different block sizes
P=zeros(17,4);
P(1,:) = [5 0 0 0];
P(2,:) = [11 18 0 18];
P(3,:) = [13 24 0 24];
P(4,:) = [11 6 0 6];
P(5,:) = [7 48 24 72];
P(6,:) = [11 54 56 2];
P(7,:) = [13 60 0 60];
P(8,:) = [17 74 72 2];
P(9,:) = [11 90 0 90];
P(10,:) = [11 96 48 144];
P(11,:) = [13 108 0 108];
P(12,:) = [13 120 60 180];
P(13,:) = [53 62 12 2];
P(14,:) = [43 64 300 824];
P(15,:) = [43 720 360 540];
P(16,:) = [31 8 24 16];
P(17,:) = [53 66 24 2];
```

```
%Parameter set corresponding to the block size of A and B
index = 0;
[length,temp]=size(A);
for j=1:17
    if (T(j)==length)
        index=j;
    end
end

AI = A;
BI = B;
t = 0;
%STEP 1, intrasymbol permutation
for k=1:length
    if rem(k,2)==0
        temp=A(k,1);
        A(k,1)=B(k,1);
        B(k,1)=temp;
    end
end
%STEP 2, intersymbol permutation
for m=0:(length-1)
    if rem(m,4)==0
        t = 0;     %P=0
    elseif rem(m,4)==1
        t = length/2 + P(index,2); %P=N/2+P1
    elseif rem(m,4)==2
        t = P(index,3);  %P=P2
    elseif rem(m,4)==3
        t = length/2 + P(index,4); %P=N/2+P3
    end
    AI(m+1,1)=A(mod(((P(index,1)*m)+t+1),length)+1);
    BI(m+1,1)=B(mod(((P(index,1)*m)+t+1),length)+1);
end
```

A.3 Encode

```
function [Y1,W1] = encode(A,B)
% This function corresponds to an 8 state double binary turbo encoder
% Two streams A and B are encoded
% Y1 and W1 are encoded A and B respectively

[length,temp]=size(A);   %size of A and B are equal
Y1 = zeros(length,1);
W1 = zeros(length,1);

Si = [ 0     % Si is the trellis state
       0     % Pre-encoder part assumes that
       0 ];  % trellis is in all zero state initially
R1 = [ 1 1 0 ];
R2 = [ 1 0 0 ];
G = [ 1 0 1
      1 0 0
      0 1 0 ];
C = [ 1 1
      0 1
      0 1 ];
```

```
for k = 1 : length
    di = [A(k)   % input to the encoder
          B(k)];
    Ti = C*di;
    Y1(k,1) = mod((sum(di) + R1*Si),2) ;
    W1(k,1) = mod((sum(di) + R2*Si),2) ;
    % Next state of the trellis is calculated
    Si = G*Si+Ti;
    Si = rem(Si,2);
end

% Final trellis state should be equal to the initial trellis state
% Matrix_Sc holds circular states and the result of Pre-encoder part is
used to find the initial state of the encoder
Matrix_Sc = [0 6 4 2 7 1 3 5
             0 3 7 4 5 6 2 1
             0 5 3 6 2 7 1 4
             0 4 1 5 6 2 7 3
             0 2 5 7 1 3 4 6
             0 7 6 1 3 4 5 2];
Sc = Matrix_Sc(mod(length,7),(Si(1,1)*4+Si(2,1)*2+Si(3,1)*1)+1);
% Initial state of the encoder
Si(3,1) = rem (Sc,2);
Si(2,1) = rem (fix(Sc./2),2);
Si(1,1) = fix(Sc./4);
%Actual Encoding
Y1 = zeros(length,1);
W1 = zeros(length,1);
for k = 1 : length
    di = [A(k)
          B(k)];
    Ti = C*di;
    Y1(k,1) = mod((sum(di) + R1*Si),2) ;
    W1(k,1) = mod((sum(di) + R2*Si),2) ;
    % Next state of the trellis is calculated
    Si = G*Si+Ti;
    Si = rem(Si,2);
end
```

A.4 SubBlock Interleaver

```
function Out = SubBlockInterleaver(u1,u2,u3,u4,u5,u6)
% This function performs sub block interleaving using the parameters
% defined in IEEE 802.16 standard
% u1,u2 are systematic bits
% u3,u4 are encoded bits
% u5,u6 are encoded bits of interleaved data

% T holds block sizes defined
T = [24 36 48 72 96 108 120 144 180 192 216 240 480 960 1440 1920
2400];
```

```
% P holds parameters m and j defined for different block sizes
P=zeros(17,2);
P(1,:) = [3 3];
P(2,:) = [4 3];
P(3,:) = [4 3];
P(4,:) = [5 3];
P(5,:) = [5 3];
P(6,:) = [5 4];
P(7,:) = [6 2];
P(8,:) = [6 3];
P(9,:) = [6 3];
P(10,:) = [6 3];
P(11,:) = [6 4];
P(12,:) = [7 2];
P(13,:) = [8 2];
P(14,:) = [9 2];
P(15,:) = [9 3];
P(16,:) = [10 2];
P(17,:) = [10 3];

index = 1;
[length,temp]=size(u1);
for j=1:17
    if (T(j)==length)
        index=j;
    end
end
% Parameters corresponding to the block size of inputs are found by
%making use of index
m=P(index,1);
J=P(index,2);
y1 = zeros(length,1);
y2 = zeros(length,1);
y3 = zeros(length,1);
y4 = zeros(length,1);
y5 = zeros(length,1);
y6 = zeros(length,1);
k = 0 ;
i = 0 ;
while i<length
    Tk = (2^m)*mod(k,J)+BitReverseOrder(floor(k./J),m);
    if Tk <length
        y1(i+1)=u1(Tk+1);
        y2(i+1)=u2(Tk+1);
        y3(i+1)=u3(Tk+1);
        y4(i+1)=u4(Tk+1);
        y5(i+1)=u5(Tk+1);
        y6(i+1)=u6(Tk+1);
        i=i+1;
    end
    k=k+1;
end
for j=1:length
    if mod(j,2)==0
        temp = y3(j);
        y3(j)= y4(j);
        y4(j)=temp;
        temp = y5(j);
        y5(j)= y6(j);
        y6(j)=temp;
    end
end;
```

```
Out = [y1;
       y2;
       y3;
       y4;
       y5;
       y6];
```

A.5 Puncturing

```
function Out = Puncture(Rate,In)
% This function punctures the data given as In to obtain the desired
% coding rate specified as "Rate"

[length,temp]=size(In);
DataSize = length/6;
if Rate == 1/2
    Out(1:DataSize*4,1) = In(1:DataSize*4,1);
elseif Rate == 2/3
    Out(1:DataSize*2,1) = In(1:DataSize*2,1);
    Out(DataSize*2+1:DataSize*3,1) = In(DataSize*2+1:2:DataSize*4,1) ;
elseif Rate == 3/4
    Out(1:DataSize*2,1) = In(1:DataSize*2,1);
    Out(DataSize*2+1:DataSize*2+DataSize*2/3,1) =In(DataSize*2+1:3:DataSize*4,1);
elseif Rate == 1/3 % no puncturing
    Out = In;
end;
```

A.6 De-puncturing

```
function Out = Depuncture(Rate,In)
% This function depunctures the data given as In to obtain the natural
% coding rate 1/3

[length,temp]=size(In);
DataSize = length*Rate/2;
Out = zeros (DataSize*6,1);
if Rate == 1/2
    Out(1:DataSize*4,1) = In(1:DataSize*4,1);
    Out(DataSize*4+1:DataSize*6,1) = zeros(DataSize*2,1);
elseif Rate == 2/3
    Out(1:DataSize*2,1) = In(1:DataSize*2,1);
    Out(DataSize*2+1:2:DataSize*4,1) = In(DataSize*2+1:DataSize*3,1) ;
elseif Rate == 3/4
    Out(1:DataSize*2,1) = In(1:DataSize*2,1);
    Out(DataSize*2+1:3:DataSize*4,1) =In(DataSize*2+1:DataSize*2+DataSize*2/3,1);
elseif Rate == 1/3  % no puncturing
    Out=In;
end;
```

A.7 Sub Block De-interleaving

```
function [A,B,Y1,W1,Y2,W2]= SubBlockDeInterleaver(In)
% This function performs subblock deinterleaving
% Input in is deinterleaved and A,B,Y1,W1,Y2,W2 are formed

[length,temp]=size(In);
BlockNo = 6 ;
A = zeros (length/BlockNo,1);
B = zeros (length/BlockNo,1);
Y1 = zeros (length/BlockNo,1);
Y2 = zeros (length/BlockNo,1);
W1 = zeros (length/BlockNo,1);
W2 = zeros (length/BlockNo,1);

K = reshape(In,[(length/BlockNo),BlockNo]);
At = K (:,1);
Bt = K (:,2);
Y1t = K (:,3);
Y2t = K (:,4);
W1t = K (:,5);
W2t = K (:,6);
for j=1:length/BlockNo
    if mod(j,2)==0
        temp = Y1t(j);
        Y1t(j)= Y2t(j);
        Y2t(j)=temp;
        temp = W1t(j);
        W1t(j)= W2t(j);
        W2t(j)=temp;
    end
end;
% T holds block sizes defined
T = [24 36 48 72 96 108 120 144 180 192 216 240 480 960 1440 1920 2400];
% P holds parameters m and j defined for different block sizes
P=zeros(17,2);
P(1,:) = [3 3];
P(2,:) = [4 3];
P(3,:) = [4 3];
P(4,:) = [5 3];
P(5,:) = [5 3];
P(6,:) = [5 4];
P(7,:) = [6 2];
P(8,:) = [6 3];
P(9,:) = [6 3];
P(10,:) = [6 3];
P(11,:) = [6 4];
P(12,:) = [7 2];
P(13,:) = [8 2];
P(14,:) = [9 2];
P(15,:) = [9 3];
P(16,:) = [10 2];
P(17,:) = [10 3];
% Parameters corresponding to the block size of inputs are found by making
% use of index
index = 1;
for j=1:17
    if T(j)==(length/BlockNo)
        index=j;
    end
end
```

```
m=P(index,1);
J=P(index,2);
y = zeros(length/BlockNo,1);
k = 0 ;
i = 0 ;
Tk=0;
while i<(length/BlockNo)
    Tk = (2^m)*mod(k,J)+BitReverseOrder(floor(k./J),m);
    if Tk <(length/BlockNo)
        A(Tk+1)=At(i+1);
        B(Tk+1)=Bt(i+1);
        Y1(Tk+1)=Y1t(i+1);
        Y2(Tk+1)=Y2t(i+1);
        W1(Tk+1)=W1t(i+1);
        W2(Tk+1)=W2t(i+1);
        i=i+1;
    end
    k=k+1;
end
```

A.8 Soft Input Soft Output Decoding

```
function [Extrinsic,AlphaOut,BetaOut]  = SISO(Ai,Bi,Y1i,W1i,ExtIn,AlphaIn,BetaIn)
% This function calculates LLR values for given inputs.
% Ai,Bi are the received LLR of systematic bits
% Y1i and W1i are the received LLR of parity bits
% ExtIn is extrinsic information for inputs 01,10 and 11 calculated in the
% previous half iteration
% AlphaIn, BetaIn are final metrics calculated in the previous half iteration
% Extrinsic is extrinsic information for inputs 01,10 and 11 calculated in
% this function
% AlphaOut, BetaOut are final metrics calculated in this function

TRELLIS_END_STATE = 1;
TRELLIS_OUT = 2;
TRELLIS_SIZE=32;
INPUT_NO=2;
M = 4;
MAX_STATE_NO=8;
TRELLIS = zeros(32,2);
TRELLIS(1,:)  = [0 0];
TRELLIS(2,:)  = [7 3];
TRELLIS(3,:)  = [4 3];
TRELLIS(4,:)  = [3 0];
TRELLIS(5,:)  = [4 0];
TRELLIS(6,:)  = [3 3];
TRELLIS(7,:)  = [0 3];
TRELLIS(8,:)  = [7 0];
TRELLIS(9,:)  = [1 2];
TRELLIS(10,:) = [6 1];
TRELLIS(11,:) = [5 1];
TRELLIS(12,:) = [2 2];
TRELLIS(13,:) = [5 2];
TRELLIS(14,:) = [2 1];
TRELLIS(15,:) = [1 1];
TRELLIS(16,:) = [6 2];
TRELLIS(17,:) = [6 3];
TRELLIS(18,:) = [1 0];
TRELLIS(19,:) = [2 0];
```

```
TRELLIS(20,:) = [5 3];
TRELLIS(21,:) = [2 3];
TRELLIS(22,:) = [5 0];
TRELLIS(23,:) = [6 0];
TRELLIS(24,:) = [1 3];
TRELLIS(25,:) = [7 1];
TRELLIS(26,:) = [0 2];
TRELLIS(27,:) = [3 2];
TRELLIS(28,:) = [4 1];
TRELLIS(29,:) = [3 1];
TRELLIS(30,:) = [4 2];
TRELLIS(31,:) = [7 2];
TRELLIS(32,:) = [0 1];

%All parameters are initialized
AlphaOut = zeros (MAX_STATE_NO,1);
BetaOut = zeros (MAX_STATE_NO,1);
[length,N]=size(Ai);
Extrinsic = zeros(3,length);
Alpha = zeros (MAX_STATE_NO,length+1);
Beta = zeros (MAX_STATE_NO,length+1);
Gamma = zeros (TRELLIS_SIZE,1);
MAXLOG = 1e7;
tempab = zeros(MAX_STATE_NO,1);

A =  Ai;
B =  Bi;
Y1=  Y1i;
W1=  W1i;
% Alpha and Beta metrics are initialized by making use of inputs AlphaIn,BetaIn
for i=1:MAX_STATE_NO
      Alpha(i,1)=AlphaIn(i,1);
      Beta(i,length+1)=BetaIn(i,1);
end

%Calculation of beta metrics
for i=length:-1:1
    for j=1:TRELLIS_SIZE
        temp_input = mod((j-1),M);
        temp_output = TRELLIS(j,TRELLIS_OUT);
        %Calculate Branch Metrics
        if temp_input == 0
            Gamma(j,1) = 0;
        elseif temp_input == 1
            Gamma(j,1) = B(i,1) + ExtIn(1,i);
        elseif temp_input == 2
            Gamma(j,1) = A(i,1) + ExtIn(2,i);
        else
            Gamma(j,1) = A(i,1)+B(i,1)+ ExtIn(3,i);
        end
        if temp_output == 0
            Gamma(j,1) = Gamma(j,1) + 0;
        elseif temp_output == 1
            Gamma(j,1) = Gamma(j,1) + W1(i,1);
        elseif temp_output == 2
            Gamma(j,1) = Gamma(j,1) + Y1(i,1);
        else
            Gamma(j,1) = Gamma(j,1) + Y1(i,1) + W1(i,1);
        end
        Gamma(j,1) = Gamma(j,1) + Beta (TRELLIS(j,TRELLIS_END_STATE)+1,i+1);
    End
```

```
    for j=1:MAX_STATE_NO
        tempab(j,1) = -MAXLOG;
    end
    % find the maximum
    for j=1:TRELLIS_SIZE
        if tempab((floor((j-1)./M)+1),1) < Gamma(j,1)
            tempab((floor((j-1)./M)+1),1) = Gamma(j,1);
        end
    end
    for j=2:MAX_STATE_NO
        tempab(j,1) = tempab(j,1)-tempab(1,1); %normalize with respect to the
first metric
        Beta(j,i)=tempab(j,1);
    end
    Beta(1,i)=0;
end
for j=1:MAX_STATE_NO
    BetaOut(j,1)=Beta(j,1); % save the final beta metric
end


%Calculation of alpha metrics
for i=1:length
    for j=1:TRELLIS_SIZE
        temp_input = mod((j-1),M);
        temp_output = TRELLIS(j,TRELLIS_OUT);
         %Calculate Branch Metrics
        if temp_input == 0
            Gamma(j,1) = 0;
        elseif temp_input == 1
            Gamma(j,1) = B(i,1)+ExtIn(1,i);
        elseif temp_input == 2
            Gamma(j,1) = A(i,1)+ExtIn(2,i);
        else
            Gamma(j,1) = A(i,1)+B(i,1)+ExtIn(3,i);
        end
        if temp_output == 0
            Gamma(j,1) = Gamma(j,1) + 0;
        elseif temp_output == 1
            Gamma(j,1) = Gamma(j,1) + W1(i,1);
        elseif temp_output == 2
            Gamma(j,1) = Gamma(j,1) + Y1(i,1);
        else
            Gamma(j,1) = Gamma(j,1) + Y1(i,1) + W1(i,1);
        end
        Gamma(j,1) = Gamma(j,1) + Alpha(floor((j-1)./M)+1,i);
    end
    for j=1:MAX_STATE_NO
        tempab(j,1) = -MAXLOG;
    end
    % find the maximum
    for j=1:TRELLIS_SIZE
        if tempab(TRELLIS(j,TRELLIS_END_STATE)+1,1) < Gamma(j,1)
            tempab(TRELLIS(j,TRELLIS_END_STATE)+1,1) = Gamma(j,1);
        end
    end
    for j=2:MAX_STATE_NO
        tempab(j,1) = tempab(j,1)-tempab(1,1);%normalize with respect to the
first metric
        Alpha(j,i+1)=tempab(j,1);
    end
    Alpha(1,i+1)=0;
end
```

```
for j=1:MAX_STATE_NO
    AlphaOut(j,1) = Alpha(j,length+1); %save the final alpha metric
end

temp_llrout = zeros(4,1);
Extrinsic=zeros(3,length);
%LLR Calculation
for i=1:length
    for j=1:TRELLIS_SIZE
            temp_input = mod((j-1),M);
            temp_output = TRELLIS(j,TRELLIS_OUT);
             %Calculate Branch Metrics
            if temp_input == 0
                Gamma(j,1) = 0;
            elseif temp_input == 1
                Gamma(j,1) = B(i,1) + ExtIn(1,i);
            elseif temp_input == 2
                Gamma(j,1) = A(i,1)  + ExtIn(2,i);
            else
                Gamma(j,1) = A(i,1)+B(i,1) + ExtIn(3,i);
            end
            if temp_output == 0
                Gamma(j,1) = Gamma(j,1) + 0;
            elseif temp_output == 1
                Gamma(j,1) = Gamma(j,1) + W1(i,1) ;
            elseif temp_output == 2
                Gamma(j,1) = Gamma(j,1) + Y1(i,1) ;
            else
                Gamma(j,1) = Gamma(j,1) + Y1(i,1) + W1(i,1) ;
            end
            Gamma(j,1) = Gamma(j,1) + Alpha(floor((j-1)./M)+1,i) + Beta
(TRELLIS(j,TRELLIS_END_STATE)+1,i+1);
    end
    for j=1:M
        temp_llrout(j,1) = -MAXLOG;
    end
    % Find the maximum
    for j=1:TRELLIS_SIZE
        if temp_llrout((mod((j-1),M))+1,1)<Gamma(j,1)
            temp_llrout((mod((j-1),M))+1,1) = Gamma(j,1);
        end
    end
    for j=2:M
        Extrinsic((j-1),i) = temp_llrout(j,1)-temp_llrout(1,1); %Normalize with
respect to LLR of input 00
    end
end
Extrinsic = Extrinsic - ExtIn ;
```

A.9 Interleaving Extrinsic Information

```
function LLR_Int = Interleaver_Ext(Ext)
% This function interleaves extrinsic information

P=zeros(2400,4);
P(24,:) = [5 0 0 0];
P(36,:) = [11 18 0 18];
P(48,:) = [13 24 0 24];
P(72,:) = [11 6 0 6];
P(96,:) = [7 48 24 72];
P(108,:) = [11 54 56 2];
P(120,:) = [13 60 0 60];
P(144,:) = [17 74 72 2];
P(180,:) = [11 90 0 90];
P(192,:) = [11 96 48 144];
P(216,:) = [13 108 0 108];
P(240,:) = [13 120 60 180];
P(480,:) = [53 62 12 2];
P(960,:) = [43 64 300 824];
P(1440,:) = [43 720 360 540];
P(1920,:) = [31 8 24 16];
P(2400,:) = [53 66 24 2];

[temp,length]=size(Ext);

C = zeros(2,length);
C(1:2*length) = 1:2*length;
D = zeros(2,length);
t = 0;
interleaver = zeros(3,length);
%STEP 1
for k=1:length
    if rem(k,2)==0
        C(1,k)=2*k;
        C(2,k)=2*k-1;
    end
end

%STEP 2
for m=0:(length-1)
    if rem(m,4)==0
        t = 0;     %P=0
    elseif rem(m,4)==1
        t = length/2 + P(length,2); %P=N/2+P1
    elseif rem(m,4)==2
        t = P(length,3);  %P=P2
    elseif rem(m,4)==3
        t = length/2 + P(length,4); %P=N/2+P3
    end
    D(:,m+1)=C(:,(mod(((P(length,1)*m)+t+1),length)+1));
end

Inter_M = reshape(D,1,2*length);

couple_index = ceil(Inter_M(1:2:2*length)/2);
interleaver(1,:) = (couple_index-1 + Inter_M(1:2:2*length))';
interleaver(2,:) = (couple_index-1 + Inter_M(2:2:2*length))';
interleaver(3,:) = (3*couple_index)';
LLR_Int = Ext(interleaver);
```

A.9 De-interleaving Extrinsic Information

```
function LLR = DeInterleaver_Ext(Ext)
% This function deinterleaves extrinsic information

P=zeros(2400,4);
P(24,:) = [5 0 0 0];
P(36,:) = [11 18 0 18];
P(48,:) = [13 24 0 24];
P(72,:) = [11 6 0 6];
P(96,:) = [7 48 24 72];
P(108,:) = [11 54 56 2];
P(120,:) = [13 60 0 60];
P(144,:) = [17 74 72 2];
P(180,:) = [11 90 0 90];
P(192,:) = [11 96 48 144];
P(216,:) = [13 108 0 108];
P(240,:) = [13 120 60 180];
P(480,:) = [53 62 12 2];
P(960,:) = [43 64 300 824];
P(1440,:) = [43 720 360 540];
P(1920,:) = [31 8 24 16];
P(2400,:) = [53 66 24 2];

[temp,length]=size(Ext);

C = zeros(2,length);
C(1:2*length) = 1:2*length;
D = zeros(2,length);
t = 0;
interleaver = zeros(1,3*length);
LLR = zeros(3,length);
%STEP 1
for k=1:length
    if rem(k,2)==0
        C(1,k)=2*k;
        C(2,k)=2*k-1;
    end
end
%STEP 2
for m=0:(length-1)
    if rem(m,4)==0
        t = 0;     %P=0
    elseif rem(m,4)==1
        t = length/2 + P(length,2); %P=N/2+P1
    elseif rem(m,4)==2
        t = P(length,3);  %P=P2
    elseif rem(m,4)==3
        t = length/2 + P(length,4); %P=N/2+P3
    end
    D(:,m+1)=C(:,(mod(((P(length,1)*m)+t+1),length)+1));
end

Inter_M = reshape(D,1,2*length);
couple_index = ceil(Inter_M(1:2:2*length)/2);
interleaver(1:3:3*length) = (couple_index-1 + Inter_M(1:2:2*length))';
interleaver(2:3:3*length) = (couple_index-1 + Inter_M(2:2:2*length))';
interleaver(3:3:3*length) = (3*couple_index)';
LLR(interleaver) = Ext;
```

A.10 Decision

```
function [Out,Number]=Decision(A,B,In)
% This function decides on the received bits by making use of extrinsic
% information given as In
% This unction also calculates bit error rate by using actual data sent by
% the transmitter
% Output "Number" is the number of bits with error

[temp,length] = size(In);
temp_llrout = zeros(4,1);
Detected = zeros(2*length,1);
for i=1:length
    temp_llrout(1,1) = 0 ;
    temp_llrout(2,1) = In(1,i);
    temp_llrout(3,1) = In(2,i);
    temp_llrout(4,1) = In(3,i);
    if(temp_llrout(4,1)>temp_llrout(3,1))
        term1 = temp_llrout(4,1);
    else
        term1 = temp_llrout(3,1);
    end
    if(temp_llrout(1,1)>temp_llrout(2,1))
        term2 = temp_llrout(1,1);
    else
        term2 = temp_llrout(2,1);
    end
    if(temp_llrout(4,1)>temp_llrout(2,1))
        term3 = temp_llrout(4,1);
    else
        term3 = temp_llrout(2,1);
    end
    if(temp_llrout(1,1)>temp_llrout(3,1))
        term4 = temp_llrout(1,1);
    else
        term4 = temp_llrout(3,1);
    end
    Detected(i,1)=term1-term2;
    Detected(i+length,1)=term3-term4;
end

Out=(Detected>0)+0;
Data = [ A ; B];
[Number,Ratio] = biterr(Out,Data);
```

BIBLIOGRAPHY

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

[2] M.C. Valenti and J. Sun, "The UMTS Turbo Code an Efficient Decoder Implementation Suitable for Software Defined Radios," *International Journal of Wireless Information Networks*, 2001

[3] C. Douillard, C. Berrou, "Turbo Codes with Rate –m/(m+1) Constituent Convolutional Codes", *IEEE Transactions on Communications*, 2005

[4] C. Berrou, M. Jezequel, C. Douillard and S. Kerouedan, "The Advantages of Non-Binary Turbo Codes", *IEEE Information Theory Workshop*, 2001, pp. 61-63

[5] IEEE Std 802.16 Part 16: Air Interface for Fixed Broad Band Wireless Access Systems, 2004

[6] C. Zhan, T. Aslan, A. T. Erdogan and S. MacDougall, "An Efficient Decoder Scheme for Double Binary Circular Turbo Codes", *IEEE international conference on Acoustics, Speech and Signal Processing*, 2006, *ICASSP 2006 proceedings Volume 4*, 2006, pp. IV229-IV232

[7] LB Communications, **"**Method for Hardware implementation of a Convolutional Turbo Code Interleaver and a Sub-block Interleaver"

[8] D. Giancristofaro, A. Bartolazzi, "Novel DVB-RSC Standard Turbo Code: Details and Performances of a Decoding Algorithm", *ESA conference*, 2001

[9] C. Douillard, M. Jezequel and C. Berrou, "The Turbo Code Standard for DVB-RSC", *2nd International Symposium on Turbo Codes & Related Topics*, Brest, France, 2000, pp. 535-538

[10] L. R. Bahl, J. Cocke, F. Jelinek and J. Raviv, "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate", *IEEE Transactions on Information Theory*, 1974

[11] J. F. Cheng and T. Ottosson, "Linearly Aprroximated log-MAP Algorithms for Turbo Coding", *Proc. Of IEEE VTC*, 2000

[12] J. Vogt, A. Finger, "Improving the Max-Log-MAP Turbo Decoder", *Electronics letters, Vol.36 No:23*, 2000

[13] J. Bjarmark, M. Strandberg, "Hardware Accelerator for Duo Binary CTC decoding : Algorithm Selection, HW/SW Partitioning and FPGA Implementation", *MS Thesis*, 2006

[14] J. H. Kim, I. C. Park, "Energy Efficient Double Binary Tail Biting Turbo Decoder Based on Border Metric Encoding,", *Proc. IEEE Int. Symp. On Circuits and Systems,* 2007, pp. 1325-1328

[15] A. J. Viterbi, "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes", *IEEE. J. Sel Areas Commun. Vol. 16, no:2*, 1998, pp. 260-264