LOW DENSITY PARITY CHECK DECODER IMPLEMENTATIONS

by

Rukiye Güldalı

BS, Electronics and Telecommunication Engineering, İstanbul Technical University,1998

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the reqirements for degree of
Master of Science

Graduate Program in System and Control Engineering
Boğaziçi University

2007

LOW DENSITY PARITY CHECK DECODER IMPLEMENTATIONS

APPROVED BY:

Prof. Günhan Dündar .....
(Thesis Supervisor)

Assist. Prof. Mutlu Koca .....

Assist. Prof. Şenol Mutlu .....

DATE OF APPROVAL:04.05.2007

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Prof. Günhan Dündar for his patience, understanding and invaluable guidance during the preparation of my thesis. I also would like to thank to my thesis committee members for their help and guidance.

I would like to thank my colleagues in ST Microelectronics who were always eager to help me. Finally, I would like to thank my family who supported me a lot during the preparation of this thesis.

ABSTRACT

LOW DENSITY PARITY CHECK DECODER IMPLEMENTATIONS

Low density parity check (LDPC) codes are linear block codes used for error correction mostly in high speed digital communication systems like digital broadcasting, optical fiber communications and wireless local area networks. LDPC codes have been subject to extensive research because of their significant performance in error correction. LDPC codes are mainly decoded using an iterative algorithm called sum product algorithm. This algorithm can be implemented in both probability and log domains. Since it is more suitable for hardware, sum product algorithm is commonly implemented in log-domain.

The work done in this MS thesis is hardware implementation of LDPC decoders for variations of sum product algorithm in log-domain. Irregular LDPC codes which were found to be better in error correction were used in all implementations. Decoders were designed configurable for usage of different parity check matrices. All decoders were implemented using parallel architecture and one of the variations of the algorithm was also implemented using serial and semi-parallel architectures. Decoders were implemented in VHDL (VHSIC Hardware Description Language). Functional verification was made by running simulations, using Cadence NCSIM simulator, a top-level VHDL testbench and input stimuli generated using MATLAB. As the result of the simulations, bit error rate (BER) values for different signal to noise ratio (SNR) values were found for each decoder implementation. The implementations were synthesized to logic gates in 65 nm technology. Area reports were generated using the synthesis tool, Synopsys Design Compiler. Finally, the power estimation was done for each decoder implementation using Synopsys Power Compiler tool. As the result of the analysis, the decoder implementations are compared according to their BER performance, area and power consumption.

ÖZET

“LOW DENSITY PARITY CHECK DECODER” UYGULAMALARI

Low Density Parity Check (LDPC) kodlar, gürültülü kanallarda sayısal veri iletiminde oluşabilecek hataların düzeltilmesinde kullanılan lineer blok kodlardır. Sayısal yayın, fiber optik iletişim sistemleri, kablosuz iletişim ağları gibi yüksek hızlı sayısal veri iletim sistemlerinde kullanılan LDPC kodları, çok yüksek hata düzeltme oranları sağlamaktadır. Bu özelliklerinden dolayı pek çok araştırmaya konu olan LDPC kodlarının çözülmesi için toplam çarpım algoritması adında yinelemeli bir algoritma kullanılır. Bu algoritma hem olasılık tanım alanında hem logaritma tanım alanında kullanılabilir fakat donanım tasarımına uygunluğu nedeniyle logaritma tanım alanındaki kullanımı yaygındır.

Bu master tezinin içeriği logaritma tanım alanında toplam çarpım algoritmasının değişik varyasyonları donanım yapılarının tasarlanmasıdır. Hata düzeltmede daha iyi oldukları bulunduğundan, tasarımlarda düzgün olmayan eşlik kontrol matris yapısı kullanılmıştır. Matris parametreleri programlanabilen LDPC kod çözücüler, paralel mimari kullanılarak tasarlanmıştır. Ayrıca toplam çarpım algoritmasının bir varyasyonu için yarı paralel ve seri mimariler de kullanılarak tasarımlar yapılmıştır. Tasarımlar VHDL adı verilen donanım tasarım dili kullanılarak oluşturulmuştur. Daha sonra tasarımlar, VHDL testbenchler ve MATLAB kullanılarak oluşturulmuş giriş verisi kullanılarak oluşturulan doğrulama ortamında, Cadence NCSIM lojik simülatörle simule edilmiştir. Simülasyonların sonucunda her bir tasarım için değişik işaret gürültü oranlarında bit hata oranı değerleri elde edilmiştir. Ayrıca tasarımlar Synopsys Design Compiler devre sentez programı kullanılarak 65 nm teknolojisinde sentezlenmiş ve her tasarım için alan bilgileri elde edilmiştir. Son olarak devrelerin tahmini güç değerleri Synopsys Power Compiler programı kullanılarak bulunmuştur. Sonuç olarak, kod çözücü tasarımları, bit hata oranı performansı, alan ve güç harcaması özelliklerine göre karşılaştırılmıştır.

TABLE OF CONTENTS

ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES x
1. INTRODUCTION 1
2. LOW DENSITY PARITY CHECK CODES 3
2.1. Introduction 3
2.2. Linear Block Codes 3
2.3. Low Density Parity Check Codes 5
2.4. Low Density Parity Check Encoding 9
2.5. Low Density Parity Check Decoding Algorithms 9
2.5.1. Sum Product Algorithm in Probability Domain 12
2.5.2. Sum Product Algorithm in Log Domain 15
2.5.2.1. Min-sum approximation method 18
2.5.2.2. Look-up table approximation method 19
2.5.2.3. Piecewise linear approximation method 20
2.5.2.4. Linear approximation method 21
3. IMPLEMENTATIONS OF LDPC DECODER 22
3.1. Introduction 22
3.2. Implementation of Sum Product Algorithm in Probability Domain 23
3.2.1. Create_Input Submodule 26
3.2.2. Initialization Module 27
3.2.3. Allchecknodes Module 28
3.2.4. Allbitnodes Module 30
3.2.5. ComputebigQ Module 31
3.2.6. DecodebigQ Module 31
3.2.7. Computesyndrome Module 32
3.2.8. Controller Module 33
3.3. Implementations of Sum Product Algorithm in Log Domain 33
3.3.1. Common Modules of Implementations 34

3.3.1.1. Create_Input Module 34
3.3.1.2. Initialization Module 35
3.3.1.3. Allchecknodes Module 35
3.3.1.4. Allbitnodes Module 37
3.3.1.5. ComputebigQ Module 38
3.3.1.6. DecodebigQ Module 38
3.3.1.7. Computesyndrome Module 39
3.3.1.8. Controller Module 39
3.3.2. Min-Sum Approximation Method Implementation 39
3.3.2.1. Parallel Architecture Implementation 39
3.3.2.2. Semi-Parallel Architecture Implementation 40
3.3.2.3. Serial Architecture Implementation 41
3.3.3. Look-up Table Approximation Method Implementation 42
3.3.4. Piecewise Linear Approximation Method Implementation 43
3.3.5. Optimum Linear Approximation Method Implementation 44
4. VERIFICATION AND BIT ERROR RATE ANALYSIS 45
5. SYNTHESIS AND POWER ANALYSIS 48
6. SUMMARY OF FINDINGS 51
REFERENCES 54

LIST OF FIGURES

Figure 2.1. Tanner graph of a parity check matrix H with a cycle of length four ……. 7

Figure 2.2. Message passing structure of an LDPC decoder ……... 10

Figure 2.3. General structure of LDPC encoder and iterative decoder structure …… 11

Figure 2.4. $y(x) = \log(1 + e^{-|x|})$ function …….…. 19

Figure 3.1. Top level diagram of LDPC decoder implementations ……..…….. 24

Figure 3.2. Create_input module block diagram …….....…..... 27

Figure 3.3. Initialization module block diagram …….....…. 27

Figure 3.4. Initialization of the message matrix q_init using pin input vector …….. 28

Figure 3.5. Check node sub-module r_{ji} calculation …….....……... 29

Figure 3.6. Bit node sub-module q_{ij} calculation ……..... 30

Figure 3.7. Calculation of jth bit of syndrome vector in computesyndrome module .. 32

Figure 3.8. Sign values part of $L(r_{ji})$ calculation in check node sub-module ……... 36

Figure 3.9. Minimum finding part of $L(r_{ji})$ calculation in check node sub-module 36

Figure 3.10. Bit node sub-module structure in min-sum approximation algorithm …. 38

Figure 3.11. Semi-parallel architecture of allchecknodes module ……..... 41

Figure 3.12. Check node module in serial architecture implementation ……..…….. 42

Figure 3.13. Check node sub-module structure for look-up table approximation implementation …….....…..... 43

Figure 3.14. $L(r_{ji})$ calculation in piecewise linear approximation method ……..... 44

Figure 3.15. $L(r_{ji})$ calculation in optimum linear approximation method …….. 44

Figure 6.1. BER performances of LDPC decoders for different SNR values …….... 51

LIST OF TABLES

Table 2.1. Look-up table for $\delta(x)$ implementation ……..... 20

Table 2.2. Linear equations for piecewise linear approximation method ……..…….. 20

Table 3.1. Top level entity of probability domain sum product algorithm implementation …….....…..... 25

Table 4.1. Decoding latency values for decoder implementations ……..... 46

Table 5.1. Area values of LDPC decoder implementations ……... 48

Table 5.2. Power estimation values for LDPC decoder implementations ……..... 50

1. INTRODUCTION

The recent improvements in high-speed communication increased the importance of efficiency and reliability in digital communication systems. Since error correction is one of the most important subjects for designing a reliable digital communication system, a lot of research activity has been done in the last years on this subject. One of the most significant outputs of these activities is the rediscovery of Low Density Parity Check (LDPC) codes which were developed by Gallager [1] in the 1960's.

LDPC codes remained unnoticed for nearly thirty years because they were too complex to implement using the hardware implementation techniques of those days. Later in 1990's they were rediscovered by MacKay and Neal [2] and they were found to have very good error correcting capability. Extensive research activities following this rediscovery showed that it is possible to have a channel capacity which is very near to Shannon limit using LDPC codes as error correction method in a digital communication system.

An LDPC code is linear block code specified by a sparse parity check matrix which determines the error correcting performance. Encoding of LDPC codes is made using a generating matrix derived from the parity check matrix. Decoding of LDPC codes is generally made using derivations of "sum product algorithm" that can also be found with the name "message passing algorithm" in the literature. Sum product algorithm was presented by MacKay [12]. This algorithm can be classified as a belief propagation algorithm which uses a bipartite graph composed of bit nodes and check nodes and belief messages are sent between these bit nodes and check nodes. Different derivations of sum product algorithm are used in different application areas of LDPC codes and each has its own advantages and disadvantages according to its application area.

Sum product algorithm can be implemented in two different domains like probability domain and log domain. The aim of this thesis is to research the algorithm in both domains

and make hardware implementations of different derivations of the algorithm in order to compare them in terms of suitability for hardware implementation.

LDPC decoders implemented in this thesis are written in VHDL (VHSIC Hardware Description Language) in RTL (Register Transfer Level). They are functionally verified using a logic simulator (NCSIM from Cadence) using VHDL testbenches and input stimuli generated using MATLAB. As the result of this verification, bit error rate (BER) values are found for each implementation for different signal to noise ratio (SNR) values. In addition, decoding latency is found for each implementation. After verification, the decoder implementations are synthesized to logic gates of 65 nm technology using Synopsys Design Compiler tool and area reports are generated for each implementation. Finally, the estimated power consumption for each implementation was found using Synopsys Power Compiler tool. Using the result of this analysis, a comparison is made in terms of hardware design suitability among the decoder implementations to find the implementation type which is more convenient for hardware implementation.

2. LOW DENSITY PARITY CHECK CODES

2.1. Introduction

In this chapter, the necessary background for the research of Low Density Parity Check (LDPC) decoder implementations is presented. Since LDPC codes are a special case of linear block codes, main properties of linear codes are discussed first. Then LDPC codes and their general properties are explained. Finally LDPC decoding is explained and two main types of LDPC decoding algorithms are presented.

2.2. Linear Block Codes

Linear block coding is a subtype of block coding that is made by dividing the information sequence into message blocks. Linear block codes have a linear algebraic structure that provides a reduction in the encoding and decoding complexity compared to arbitrary block codes.

Definition 2.1. An (n, k) linear block code ζ with message word length k and codeword length n over the finite field F2 = ({0, 1}, +, ·) is a k dimensional subspace of the vector space V_n(F2), of n-tuples with elements from F2. There are 2k message words $u = [u_0, u_1, \ldots, u_{k-1}]$ and 2k corresponding code words $c = [c_0, c_1, \ldots, c_{n-1}]$ in the code ζ. Thus a linear code of length n is a subspace of V_n which is spanned by k linearly independent vectors $g_0, g_1, \ldots, g_{k-1}$ of V_n. With the k linearly independent vectors $g_0, g_1, \ldots, g_{k-1}$ of V_n given above, any codeword X can be written as a linear combination of these vectors as follows:

$$X = \sum_{i=1}^{k} m_i g_i \qquad (2.1)$$

Different code words are obtained for different combinations of the coefficients of m_i. Also the codeword X can be represented by matrix multiplication as X=mG where m is a 1 by k matrix (vector) which is essentially the message word to be encoded and G is a k

by n matrix whose rows constitute the k linearly independent vectors g_i's. G is called the generating matrix of ζ. From the above discussion, it is easy to see that G has rank k, hence it can be reduced to the form G = [I_k | P] where I_k is a k by k identity matrix. The reduction of G to that form may need some column swapping which permutes the order of the bits in the code words.

In addition, using G matrix, if a message word m is encoded to a codeword c, then the first k bits of c are exactly equal to m. This results an easy extraction of original message sent after decoding a received word. The null space $\zeta^{\perp}$ of the subspace ζ has dimension n-k and is spanned by n-k linearly independent vectors h_0, h_1, …, h_{n-k-1}. Since each hi belongs to $\zeta^{\perp}$, for any c in ζ, $c.h_i^T = 0$ for all i. Furthermore, if x is any binary block of length n but x does not belong to ζ, then $x.h_i^T \neq 0$ for all i. These n-k linearly independent vectors h_i constitute the rows of a matrix called Parity Check Matrix so that $c.H^T = 0$, if and only if c belongs to ζ. Before continuing the explanations about linear block codes, the following definitions should be given:

Definition 2.2. The Hamming weight of a codeword is the number of non zero bits of the word.

Definition 2.3. The Hamming distance between two code words of same length is defined as the number of places where the digits differ between the two words. Hamming distance is a metric on the set of all code words, ζ. Also, the weight of a codeword U is simply d(U, 0) where 0 is a codeword of same length as U and consists of all zero's.

Definition 2.4. The minimum distance of the code ζ is the minimum distance between any two vectors in ζ.

The generating matrix G is chosen in such a way that the minimum distance of ζ is d. If a received word is found to be at a shorter distance from any other codeword in ζ, it indicates that an error has occurred and it should to be corrected to a codeword that is closest to it in the sense of the Hamming distance. For example, in case of a channel that

will create at most one error per codeword, minimum distance between the code words should be two. By this way, if an error occurs, the received word which is at a distance of one from at least one of the code words in ζ implies that there was an error. But it might be impossible to correct the error if the received word is at the same distance from two or more code words in ζ. Linear block codes is that in order to detect t errors, the minimum distance of ζ must be at least t+1, and in order to correct t errors, the minimum distance must be at least 2t+1 [3].

Detecting an error in a received codeword by comparing it to all possible code words in ζ and correcting the error by replacing it with the valid codeword which is at the least distance from the received codeword is not practical in case a large generating matrix is used. In such cases, syndrome decoding is used.

Definition 2.5. The syndrome of a codeword x is defined as the product of x with the transpose of the parity check matrix H like, $S = x \cdot H^T = 0$. Thus upon arrival, a received word is valid if and only if its syndrome is zero. A generating matrix G in the form of $G = [I_k \mid A]$ so that the first k bits of any codeword x are exactly equal to the message word it encodes and the parity check matrix is $H = [A^T \mid I_{n-k}]$. Syndrome decoding is used in LDPC decoding algorithms when deciding if the decoded codeword is correct or not.

2.3. Low Density Parity Check Codes

Low Density Parity Check (LDPC) codes were developed by Gallager [1] in the 1960s but for the next few decades, they remained unnoticed. The reason that they were neglected by researchers was it was not practical to implement LDPC encoder and decoder architectures in hardware in those years. They were rediscovered by MacKay and Neal [2] in 1990s and since the VLSI technologies and IC design techniques had improved drastically throughout these years, LDPC codes became an important alternative for error correction coding. There has been a lot of research activity done on LDPC codes since their rediscovery which showed their very good error correcting performance that can reach near Shannon limit [6].

LDPC codes are linear block codes specified by a sparse parity check matrix. This means the number of 1's per column (column weight) is very small compared to the column length of parity check matrix and the number of 1's per row (row weight) is very small compared to the row length of parity check matrix [20].

LDPC codes are classified into two groups like regular LDPC codes and irregular LPDC codes according to the row and column weight properties of parity check matrix. In regular LDPC codes, the parity check matrix has uniform column weight and row weight. On the contrary, in irregular LDPC codes the parity check matrix has non-uniform column weight and row weight. As the result of extensive research done on regular and irregular LDPC codes, it is found that irregular LDPC codes have a better error correcting performance than regular LDPC codes [7],[18]. On the other hand, regular LPDC codes have the advantage of regularity which brings them a big advantage like they can be implemented much easier compared to irregular LDPC codes. LDPC decoder implementations presented in this thesis have irregular LDPC code structure.

Besides the parity check matrix representation, LDPC codes can be represented by a bipartite graph called Tanner graph [1], [35]. A bipartite graph is a graph whose nodes may be separated into two classes, and where edges may only be connecting two nodes not residing in the same class. The two classes of nodes in a Tanner graph are bit nodes and check nodes. The Tanner graph of a code is drawn according to the following rule: Check node f_j ; j = 1,...,.N - K is connected to bit node x_i; i = 1,...,N whenever element h_{ji} in H (parity check matrix) is a one. Edges of the Tanner graph act as information path between bit nodes and check nodes for decoding process. Figure 2.1 shows a Tanner graph made for a simple parity check matrix H. In this graph each bit node is connected to two check nodes and each check node is connected to four bit nodes.

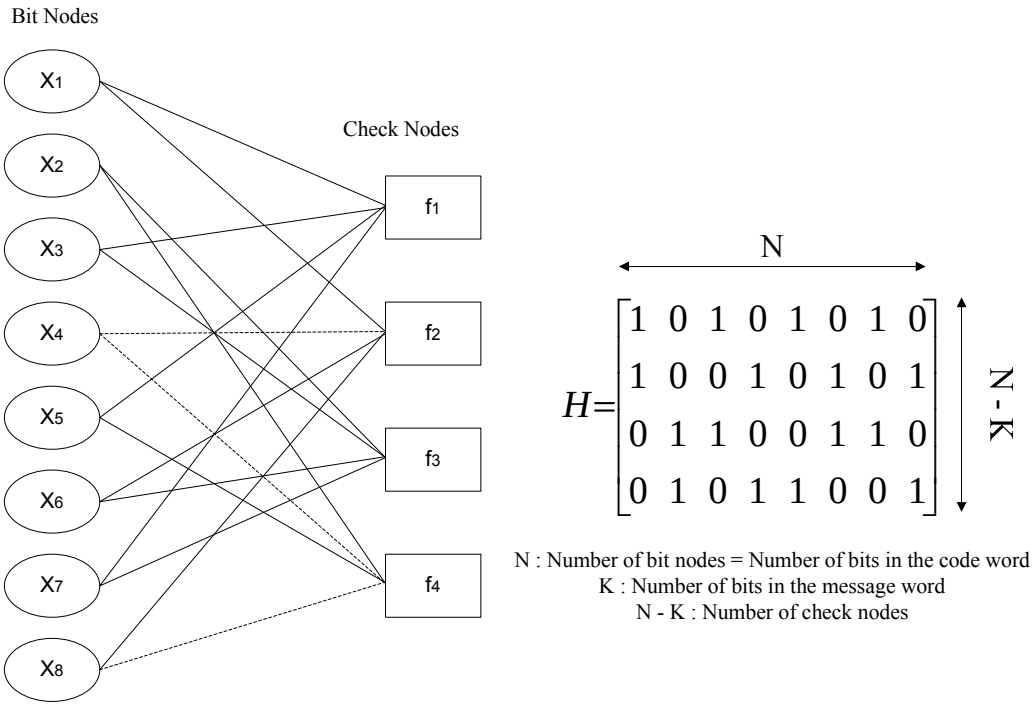

Figure 2.1. Tanner graph of a parity check matrix H with a cycle of length four

Definition 2.6. A cycle of length l in a Tanner graph is a path comprised of l edges which closes back on itself. The Tanner graph in Figure 2.1 has a cycle of length four which has been shown by dashed lines.

Definition 2.7. The girth of a Tanner graph is the minimum cycle length of the graph.
The shortest possible cycle in a bipartite graph is clearly a cycle of length four. These cycles are observed in the H matrix as four 1's that lie on the corners of a sub-matrix of H. The cycles in a Tanner graph is important because they have negative impact on the decoding algorithms for LDPC codes if they are short cycles.

LDPC codes are constructed by defining the parity check matrix H. If the parity check matrix A has N columns and M rows, any codeword generated for this LDPC code consists of N bits which satisfy M parity checks, where the location of a 1 in the parity check matrix indicates that a bit is involved in a parity check. The total length of the codeword is N bits, the number of message bits is K = N - M, and the rate of the code is R = K / N, assuming that the matrix is full rank.

The error correcting performance of LDPC codes is determined by the properties of H matrix. The most important properties that H matrix should have for a good performance LDPC code is like below:

- H matrix should be sparse. Sparseness increases the minimum distance of the code words generated for this LDPC code which results good error correction capability.[1]
- The girth of the Tanner graph derived from H matrix should be high. Short cycles in Tanner graph decreases the error correcting capability of LDPC code.[1]

Some ways to generate good performance LDPC codes and the most commonly used ways of generating H matrix are listed below:

1. Start from all zero matrix of the size (N - K) x N and randomly invert some elements in the matrix to reach the resulting degrees for different nodes.
2. Generate H by randomly creating weight W_c columns.
3. Generate H with weight W_c columns and uniform row weights of Wr.
4. Generate H with weight Wc columns and uniform row weights of Wr with no two columns have overlap of more than one. This condition removes all the length-four cycles which results in better performance.
5. Generating H like (4) and avoiding other short cycles.
6. Generate the parity check matrix in a structured manner. For example a structure that is used in hardware design is to generate this matrix using a combination of the shifted blocks of identity matrices.
7. Generate the parity check matrix using a polynomial.

Each of the above ways has advantages and disadvantages, depending on the application. Sixth way is more suitable for the hardware design. After designing the parity check matrix H, it can be put in the form $H = [P^T| I]$ by Gaussian elimination and the generator matrix G can be derived by solving $GH^T = 0$ like $G = [I | P]$.

2.4. Low Density Parity Check Encoding

LDPC encoding is more complex than it appears for LDPC codes of big codeword lengths due to the computational intensity of matrix multiplication of generating matrix G and message word. The encoding methods are out of the scope of this thesis but there is extensive research done on low complexity encoding techniques based on the H matrix and efficient methods for LDPC encoding can be found in the literature [4].Besides low complexity, it is also important that the encoding process should be suitable for different channels. Since the decoder implementations are made for AWGN channels in this thesis, encoding for AWGN channel is described briefly below.

Given a message word m, a corresponding codeword c such that $c = m \cdot G$ is generated. This codeword is then converted to integer numbers {-1, +1} word x according to the following rule: $x_i = (-1)^{ci}$. This integer codeword is then sent through the channel and white Gaussian noise $n \sim N(0, \sigma_2)$ is added to it. The resulting word has same length but the bits can have any real values that are caused due to the noise. Once decoding is done, the codeword sent is recovered via the inverse relation $c_i = 0$ if $y_i = +1$ and $c_i = 1$ if $y_i = -1$.

2.5. Low Density Parity Check Decoding Algorithms

LDPC decoding algorithms for AWGN channels are based on Gallager's [1] iterative decoding method. After reworking Gallager's method, MacKay [12] came up with the sum product algorithm. Sum product algorithm is classified as a belief propagation algorithm. Belief propagation algorithms are presented as messages update equations on a factor graph [17], [19].

Definition 2.8 : A factor graph is a bipartite graph that is composed of two kinds of nodes like variable nodes for variables and factor nodes for local functions. A variable node is connected to a factor node by an edge if the variable is an argument of the local function.

Sum product algorithm uses the Tanner graph created from the parity check matrix H, as factor graph and sends belief messages between bit nodes (variable nodes for LDPC Tanner graph) and check nodes (factor nodes for LDPC Tanner graph) [40]. By this way,

sum product algorithm determines the a posteriori probabilities for bit values based on a priori information, improving the accuracy of these calculations in each iteration [19], [20], [24], [39]. Check nodes and bit nodes in the Tanner graph perform computations in parallel and then communicate with each other over connections described by the edges of the Tanner graph. The messages that communication is composed of are estimates of probabilities [8].

The nature of the nodes in the Tanner graph and the structure of the graphs interconnections are completely described by the number and location of ones in the parity check matrix H. The check nodes determine the probability that a parity check is satisfied if one particular data bit is set to be a one (or zero) and the other data bits have values with a probability distribution corresponding to the known a priori probabilities. The bit nodes determine the probability that a data bit has the value one (or zero); given the information from all of the other check nodes. Only bits and checks that are related by having a one at a specific corresponding location in the parity check matrix need to be considered in these calculations.

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

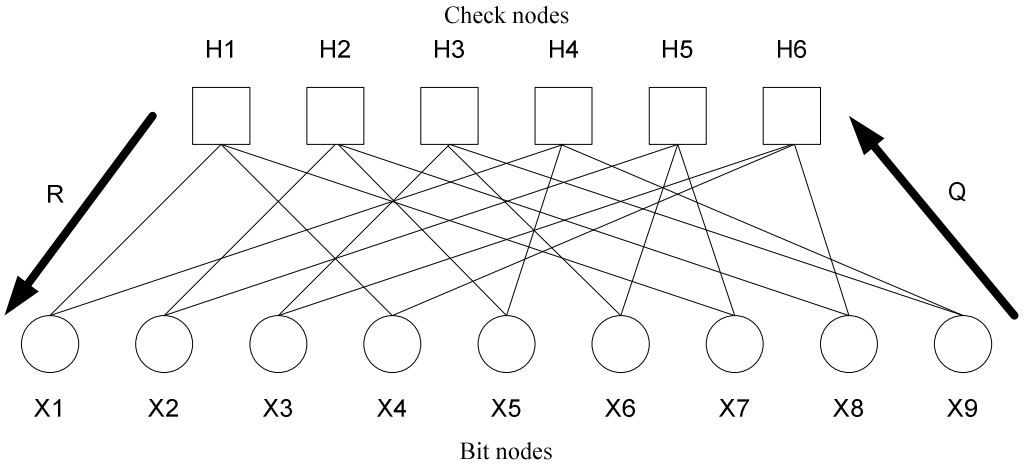

Figure 2.2. Message passing structure of an LDPC decoder

In Figure 2.2, messaging across the Tanner graph of parity check matrix H is shown. R represents messages from check nodes to bit nodes and Q represents the messages from bit nodes to check nodes. Each row of parity check matrix H corresponds to a check node in the Tanner graph. In other words each row represents a single parity check of LDPC code. Similarly each column in H represents a bit node. Consequently the number of bit nodes in the Tanner graph or the number of columns in the parity check matrix is equal to the number of bits in the codeword. The location of ones and zeros in H determine the nodes which are connected in the Tanner graph. Having a one at location row j and column i simply indicates that check node j is connected to bit node i. In the first row of H, it can be seen that there are ones in the first, fourth and seventh columns. This can be observed in the Tanner graph as connections between check node H1 (corresponding to first row in parity check matrix H) and bit nodes X1, X4 and X7(corresponding to first, fourth and seventh columns). The number of ones in a row determines the number of data inputs coming from bit nodes that the corresponding check node has. Similarly, the number of ones in a column determines the number of data inputs coming from check nodes that the corresponding bit node has. In Figure 2.3, the general structure of LDPC encoder and iterative decoder is shown.

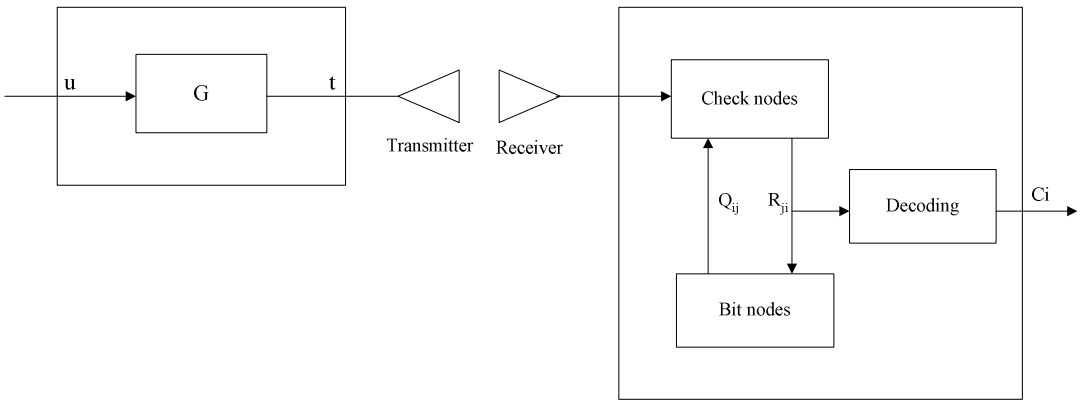

Figure 2.3. General structure of LDPC encoder and iterative decoder structure

As stated before, the content of messages include probability values but these probability values can be either real probability values or probability values in log domain. It is observed in the literature that sum product algorithm for LDPC decoding is classified into two main groups according to the structure of the messages between check nodes and

bit nodes [9]. These are sum product algorithm in probability domain and sum product algorithm in log domain. Details and sub groups of these main types of sum product algorithm will be described in detail in the next sections.

2.5.1. Sum Product Algorithm in Probability Domain

Sum Product Algorithm in Probability Domain uses real probability values in the iterative preparation of messages between check nodes and bit nodes [5]. Algorithm works as follows:

Step 1: Messages from bit nodes to check nodes (denoted as q_{ij}) are initialized to probability values calculated according to the channel characteristics and the values of decoder input bits with AWGN. This initialization is done like equations (2.2) and (2.3) where y_i is the received data with AWGN and σ is the noise variance. p_i^0 and p_i^1 represent the apriori probabilities for each bit of the received codeword determined by the data received from the AWGN channel. For the first iteration, q_{ij} values are initialized to p_i^0 and p_i^1 values. Initialization is done once for decoding of each received codeword.

$$q_{ij}^0 = 1 - p_i^1 = p_i^0 = \frac{1}{1+e^{-2y_i/\sigma^2}} \tag{2.2}$$

$$q_{ij}^1 = p_i^1 = \frac{1}{1+e^{2y_i/\sigma^2}} \tag{2.3}$$

Step 2: Messages from check nodes to bit nodes are calculated. Each check node gathers all the incoming messages from bit nodes connected to it to generate r_{ji} value where r_{ji}^0 is the probability that check j is satisfied if it is assumed that data bit $t_i = 0$.Similarly, r_{ji}^1 is the probability that check j is satisfied if it is assumed that data bit $t_i = 1$.These probabilities are computed as in equations (2.4) and (2.5) The notation $i' \in row[j] \setminus \{i\}$ means the indices i' $(1 \le i' \le n)$ of all bits in row j $(1 \le j \le m)$ which have value one, not including the current bit index, i.

$$r_{ji}^{0} = \frac{1}{2}\left[1 + \prod_{i' \in row[j]\setminus\{i\}} (q_{i'j}^{0} - q_{i'j}^{1})\right] \tag{2.4}$$

$$r_{ji}^{1} = \frac{1}{2}\left[1 - \prod_{i' \in row[j]\setminus\{i\}} (q_{i'j}^{0} - q_{i'j}^{1})\right] \tag{2.5}$$

Step 3: Messages from bit nodes to check nodes are calculated. Each bit node gathers the probability information from the check nodes that are connected to it and generate the q_{ij} values, where q_{ij}^{0} is the probability that data bit t_i = 0, given the values of all check nodes other than j.Similarly, q_{ij}^{1} is the probability that data bit t_i = 1, given the values of all check nodes other than j. These probabilities are computed as shown in equations (2.6) and (2.7). Two terms contribute to the calculation of q_{ij} values, a priori probability values used in initialization and r_{ji} values coming from check nodes. The notation $j' \in col[i]\setminus\{j\}$ means the indices $j'\ (1 \le j' \le m)$ of all checks in column i $(1 \le i \le n)$ which have value one, not including the current check index, j. α_{ij} is a normalizing value chosen so that $q_{ij}^{0} + q_{ij}^{1} = 1$.

$$q_{ij}^{0} = \alpha_{ij} p_i^{0} \prod_{j' \in col[i]\setminus\{j\}} r_{ji'}^{0} \tag{2.6}$$

$$q_{ij}^{1} = \alpha_{ij} p_i^{1} \prod_{j' \in col[i]\setminus\{j\}} r_{ji'}^{1} \tag{2.7}$$

Step 4: Extrinsic probabilities of decoder output bits are calculated. These are calculated in a similar way that q_{ij} values are calculated. These extrinsic probabilities are used to determine the values for each decoder output bit. Similar to q_{ij} values calculation, the accuracy of these probabilities improves with each iteration of the algorithm. Equations (2.8) and (2.9) show this extrinsic probability calculation, where Q_i^{0} is the probability that the decoded bit $t_i = 0$ and Q_i^{1} is the probability that the decoded bit $t_i = 1$. The notation $j' \in col[i]$ means the indices $j'\ (1 \le j' \le m)$ of all checks in column i $(1 \le i \le n)$ which have value one. α_i is a normalizing value chosen so that $Q_i^{0} + Q_i^{1} = 1$.

$$Q_i^0 = \alpha_i \prod_{j \in col[i]} r_{ji}^0 \tag{2.8}$$

$$Q_i^1 = \alpha_i \prod_{j \in col[i]} r_{ji}^1 \tag{2.9}$$

Step 5: Decoder output bit candidates are determined according to the probability values calculated in Step 4. If Q_i^1 is greater than 0.5 (or Q_i^0 is equal to or smaller than 0.5), then decoded output bit candidate $\hat{c}_i$ is determined as 1. Otherwise it is decoded as 0.

$$\hat{c}_i = \begin{cases} 1 & if\ Q_i^1 > 0.5 \\ 0 & elsewhere \end{cases} \tag{2.10}$$

Step 6: The syndrome of the decoded output candidate is calculated. As the general property of linear block codes, syndrome value indicates if the decoded output candidate is equal to the transmitted codeword. Thus, it is verified if the decoding is successful or not. The syndrome calculation is made by matrix multiplication of decoded output candidate which is a vector of $1 \times N$ and transpose of parity check matrix H^T which is a matrix of $N \times (N-K)$, where N is the number of bit nodes, K is the number of bits in the message word and N - K is the number of check nodes like:

$$\hat{c}_i \times H^T = \hat{S} \tag{2.11}$$

If $\hat{S}$ is a zero vector of $1 \times (N-K)$ then this means the received code word is decoded correctly a decoded output candidate is given out as decoder output. Otherwise, decoding continues iteratively by repeating the algorithm starting from Step 2 until the syndrome is received as zero vector. In practical applications the number of iterations are limited to some value which is usually give as a decoder parameter called maximum number of iterations.

Sum product algorithm in probability domain has some drawbacks like below:

- The algorithm suffers because of the multiplications involved which are costly as compared to additions in hardware implementations [15].

- These multiplications may result in overflow or saturation in hardware implementation when the algorithm runs for large number of iterations.
- In the calculation of messages from bit nodes to check nodes (q_{ij}) and the calculation extrinsic probabilities of decoder outputs (Q_i), there are divisions by the normalizing coefficients (represented by α_i and α_{ij}) used in the algorithm. Division by a variable is difficult to implement in hardware.

2.5.2. Sum Product Algorithm in Log Domain

Sum product algorithm in log domain is another form of sum product algorithm where the probabilities are characterized by the log-likelihood ratios (LLRs)[10].This means, the same steps are used as sum product algorithm in probability domain in but the real probability values are replaced with LLR values. Thus, instead of r_{ji} values, $L(r_{ji})$ values are used which are calculated like $L(r_{ji}) \triangleq \log \frac{r_{ji}^0}{r_{ji}^1}$ .Similarly q_{ij} values are replaced with $L(q_{ij}) \triangleq \log \frac{q_{ij}^0}{q_{ij}^1}$ values, p_i values are replaced by $L(p_i) \triangleq \log \frac{p_i^0}{p_i^1}$ values and Q_i values are replaced by $L(Q_i) \triangleq \log \frac{Q_i^0}{Q_i^1}$ .The algorithm can be described as follows:

Step 1: Messages from bit nodes to check nodes (denoted as $L(q_{ij})$) are initialized to LLR values calculated using the channel characteristics and the values of decoder input bits with AWGN. This LLR value $L(p_i)$ is calculated like equation (2.12) where yi is the received data with AWGN and σ is the noise variance. For the first iteration, $L(q_{ij})$ values are initialized to $L(p_i)$ values calculated from a priori probability values determined by the data received from the AWGN channel.

$$L(p_i) = \log \frac{p_i^0}{p_i^1} = \frac{2}{\sigma^2} y_i \quad , \; L(q_{ij}) = L(p_i) \tag{2.12}$$

Step 2: Messages from check nodes to bit nodes are calculated as LLR values. Each check node gathers all the incoming messages from bit nodes connected to it to generate $L(r_{ji})$ value. Before calculation of $L(r_{ji})$ following equations using $L(q_{ij})$ values following information should be given:

For independent random variables X_1 and X_2, the joint log-likelihood ratio $L(x_1 \oplus x_2)$ is given by:

$$L(x_1 \oplus x_2) = \ln \frac{1 + e^{L(x_1)} e^{L(x_2)}}{e^{L(x_1)} + e^{L(x_2)}} \tag{2.13}$$

Consequently, the joint log-likelihood ratio $L(x_1 \oplus ... \oplus x_l)$ can be derived like

$$L(x_1 \oplus ... \oplus x_l) = \ln \frac{1 + \prod_{i=1}^{l} \tanh(L(x_i)/2)}{1 - \prod_{i=1}^{l} \tanh(L(x_i)/2)} = 2.\tan^{-1}\left(\prod_{i=1}^{l} \tanh(L(x_i)/2) \right) \tag{2.14}$$

Thus, $L(r_{ji})$ which is composed of $L(q_{ij})$ values can be calculated like:

$$L(r_{ji}) = 2.\tan^{-1}\left(\prod_{i' \in row[j] \setminus \{i\}} \tanh(L(q_{i'j})/2) \right) \tag{2.15}$$

The notation $i' \in row[j] \setminus \{i\}$ means the indices i' $(1 \le i' \le n)$ of all bits in row j $(1 \le j \le m)$ which have value one, not including the current bit index, i.

Step 3: Messages from bit nodes to check nodes are calculated as LLR values. Similar to probability domain algorithm, each bit node gathers the probability information in LLR domain from the check nodes that are connected to it and generate the $L(q_{ij})$ values. These LLR values are computed as shown in equation (2.16). Two terms contribute to the calculation of $L(q_{ij})$ values, LLR calculated from a priori probability values used in initialization which is $L(p_i)$ and $L(r_{ji})$ values coming from check nodes. The notation

$j' \in col[i] \setminus \{j\}$ means the indices $j' (1 \le j' \le m)$ of all checks in col i $(1 \le i \le n)$ which have value one, not including the current check index, *j*. m is the number of check nodes and n is the number of bit nodes in parity check matrix.

$$L(q_{ij}) = L(p_i) + \left(\sum_{j' \in col(i) \setminus \{j\}} L(r_{j'i}) \right) \tag{2.16}$$

Step 4: Extrinsic LLR values $L(Q_i)$ of decoder output bits are calculated (in a similar way used for $L(q_{ij})$ calculation) for determining decoder output bits. Similar to $L(q_{ij})$ values calculation, the accuracy of these LLR values improves with each iteration of the algorithm. Equation (2.17) shows, $L(Q_i)$ calculation, where the notation $j' \in col[i] \setminus \{j\}$ means the indices $j' (1 \le j' \le m)$ of all checks in column i $(1 \le i \le n)$ which have value one, not including the current check index, *j*.

$$L(Q_i) = L(p_i) + \left(\sum_{j' \in col(i) \setminus \{j\}} L(r_{j'i}) \right) \tag{2.17}$$

Step 5: Decoder output bit candidates are determined according to the LLR values calculated in Step 4. If $L(Q_i)$ is negative then decoded output bit candidate $\hat{c}_i$ is determined as 1. Otherwise it is decoded as 0.

$$\hat{c}_i = \begin{cases} 1 & if\ L(Q_i) < 0 \\ 0 & elsewhere \end{cases} \tag{2.18}$$

Step 6: The syndrome of the decoded output candidate is calculated in the same way as probability domain algorithm.

$$\hat{c}_i \times H^T = \hat{S} \tag{2.19}$$

If syndrome vector $\hat{S}$ is a zero vector of $1 \times (N - K)$ then the received code word is decoded correctly and the decoded output candidate is given out as decoder output. Otherwise, decoding continues iteratively by repeating the algorithm starting from Step 2

until the syndrome is received as zero vector. Again, like in probability domain algorithm the number of iterations should be limited to some value which is usually given as a decoder parameter called maximum number of iterations.

$L(r_{ji})$ values are calculated iteratively, so that the first $L(q_{ij})$ values combined and then the result is combined with the next $L(q_{ij})$ and so on. This process is very difficult to implement is hardware. A property of joint log-likelihood ratios can be useful in implementation of this formula. Equation (2.13) can be rewritten like:

$$L(x_1 \oplus x_2) = sign\{L_{x_1} L_{x_2}\}.\min\{|L_{x_1}|, |L_{x_2}|\} + \delta(L_{x_1}, L_{x_2}), \text{ where} \qquad (2.20)$$

$$\delta(Lx_1, Lx_2) = \log(1 + e^{-|L_{x_1} + L_{x_2}|}) + \log(1 + e^{-|L_{x_1} - L_{x_2}|}))$$

Now, using this property $L(r_{ji})$ calculation can be rewritten like:

$$L(r_{ji}) = \prod_{i' \in row[j] \setminus \{i\}} sign\{L(q_{i'j})\}.\min\left(\left|L(q_{i'j})\right|\right) + \delta(L(q_{i'j})) \qquad (2.21)$$

Most of the multiplications used in probability domain algorithm become additions in log domain algorithm. Only the multiplications used in $L(r_{ji})$ calculations are left, but since these multiplications are made for sign values of $L(q_{ij})$ values, they can be implemented using XOR gates. This reduces the computational complexity in hardware implementation. In addition, division by normalizing coefficient does not exist in log domain algorithm. On the other hand, log domain algorithm has a drawback like implementation of function $\delta(x)$ which is called correction term. $\delta(x)$ is a nonlinear function and is difficult to implement in hardware. There are some solutions found for implementation of $\delta(x)$ in the literature and these solutions bring out sub types of log domain algorithm [11], [16], [21], [23], [26], [30], [31], [33].

2.5.2.1. Min-sum approximation method: In this approximation method, the correction term is completely omitted and $L(r_{ji})$ is implemented like below:

$$L(r_{ji}) = \prod_{i' \in row[j] \setminus \{i\}} sign\{L(q_{i'j})\}.\min(\left|L(q_{i'j})\right|) \qquad (2.22)$$

Omitting the correction term brings a simplification in implementation but it causes performance degradation in error correcting performance of LDPC decoder [11], [14], [23], [26].

2.5.2.2. Look-up table approximation method: Implementation of $\delta(x)$ is made using a look-up table in this method [16], [21], [31], [33]. It is sufficient to prepare a look-up table for function $y(x) = \log(1 + e^{-|x|})$ for values since both components of $\delta(x)$ can be presented like $y(x)$.Figure 2.4. shows $y(x)$ function.

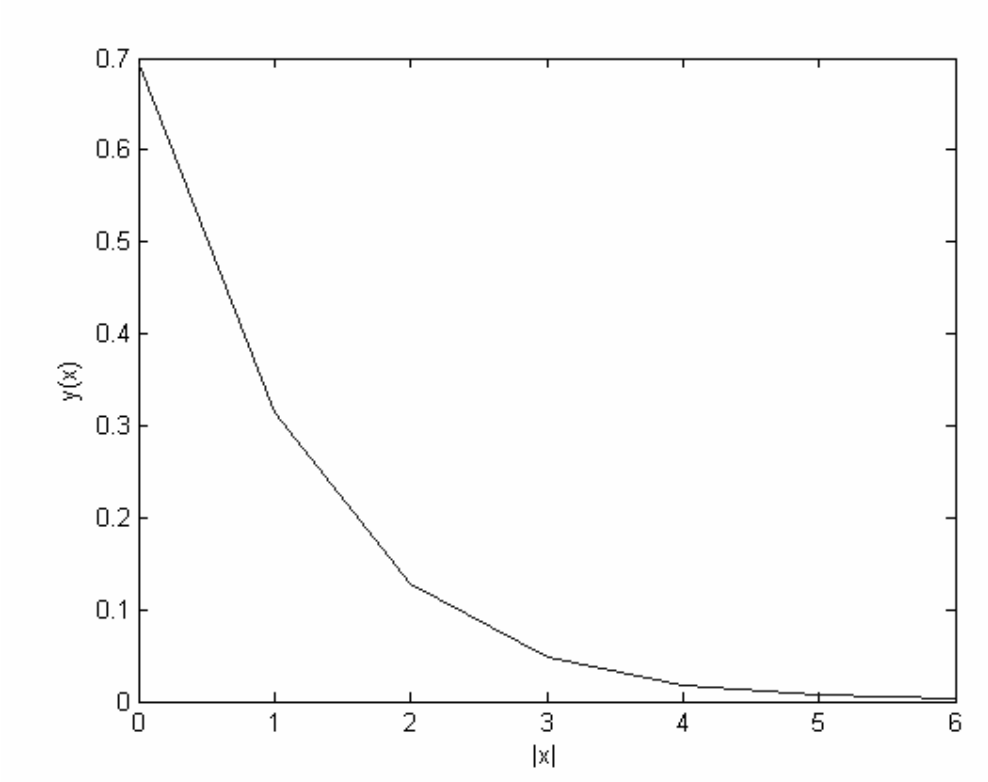

Figure 2.4. $y(x) = \log(1 + e^{-|x|})$ function

Usage of look-up table for $\delta(x)$ is commonly seen in the literature but this method has a trade-off between the size of the look-up table versus accuracy [11]. In Table 2.1, a look-up table that can be used for this method is shown. The maximum approximation error is 0.05.

|x|	**$\log(1+e^{-\|x\|})$**
[0, 0.2)	0.65
[0.2, 0.4)	0.55
[0.4, 0.7)	0.45
[0.7, 1.0)	0.35
[1.0, 1.5)	0.25
[1.5, 2.2)	0.15
[2.2,4.5)	0.05
$[4.5,+\infty)$	0.0

Table 2.1. Look-up table for $\delta(x)$ implementation

2.5.2.3. Piecewise linear approximation method: Similar to look-up table method, $y(x) = \log(1+e^{-|x|})$ is implemented in this method in order to compose $\delta(x)$. $y(x)$ non-linear equation is replaced with a group of linear equations for different margins of x. To ease the implementation in hardware, these linear equations are chosen such that the multiplication factors are chosen as factors of two [11]. In Table 2.2, an example of piecewise linear approximations of $y(x)$ is shown for six margins of x is shown. If the number of linear equations used in the approximation increase, (this means the length of the margins of x decrease) the perfection of the approximation increases but the implementation becomes more complex [16].

|x|	**$\log(1+e^{-\|x\|})$**
[0, 0.5)	$-2^{-1}\|x\| + 0.7$
[0.5, 1.6)	$-2^{-2}\|x\| + 0.575$
[1.6, 2.2)	$-2^{-3}\|x\| + 0.375$
[2.2, 3.2)	$-2^{-4}\|x\| + 0.2375$
[3.2, 4.4)	$-2^{-5}\|x\| + 0.1375$
$[4.4, +\infty)$	0.0

Table 2.2. Linear equations for piecewise linear approximation method

2.5.2.4. Linear approximation method: In this method, instead of using several linear functions for different regions, an optimum linear function is found to approximate $y(x) = \log(1 + e^{-|x|})$ in order to compose $\delta(x)$. In [13], the optimum linear approximation was found like y(x) = -0.24|x| + 0.6 as the result of some simulations. This method brings simpler implementation compared to piecewise linear approximation.

3. IMPLEMENTATIONS OF LDPC DECODER

3.1. Introduction

In this chapter, LDPC decoder hardware implementations done in this thesis are presented. The implementation of sum product algorithm in probability domain for irregular LDPC codes is presented first. Then parallel architecture implementations for different methods of sum product algorithm in log domain are presented. In addition to these, for min-sum approximation method of log domain algorithm, implementations of semi-parallel architecture and serial architecture are also presented. Since probability domain algorithm has the drawbacks explained in Section 2.5.1., this thesis aims to focus more on log domain algorithm implementations. All implementations are made suitable for AWGN channel. This means a priori probability values used in initialization are calculated for AWGN channel characteristics like $N(0,\sigma^2)$ where zero is the mean value and $\sigma^2 = 1$ is the noise variance taken in the implementations presented in this thesis.

The same top level structure is used in the implementations of parallel architecture for both sum product algorithm in probability domain and sum product algorithm in log domain. The difference in the implementation for these two types of the algorithm is the functionality of check nodes and bit nodes modules. Also the bit precisions of ports of modules are different for the implementations, because the nature of input data is different in two implementations like probability information for probability domain algorithm and log-likelihood information for log domain algorithm.

The LDPC decoders implemented in this thesis are irregular LDPC decoders which are found to be better in error correction. Parity check matrix is defined in a common VHDL package to be used by all related modules. In this thesis, a parity check matrix H of irregular structure composed of 12 check nodes and 17 bit nodes is used for both implementations. Similarly, the number of bit nodes, the number of check nodes and the number of iterations in the algorithm are also defined in the same common package. Thus in order to use another type of LDPC code, it is sufficient to change these parameters in the common package. This property brings configurability to the decoder implementations.

Top level diagram of LDPC decoders implemented in this thesis is shown in Figure 3.1. Functionality details of modules for each algorithm will be explained in the next chapters.

3.2. Implementation of Sum Product Algorithm in Probability Domain

Sum product algorithm in probability domain is implemented like each step of the flow described in Chapter 2.5.1 is implemented in one module. Each module is implemented in a parallel architecture. Bit nodes function in parallel to calculate messages to check nodes and check nodes function in parallel to calculate messages to bit nodes. Similarly, extrinsic probability values are calculated in parallel for each bit.

There is a controller module in decoder top level which enables all the modules (except create_input module) using the output_ready indication signals coming from each module. When the received codeword with AWGN is ready at the input of the LDPC decoder, the input_ready signal is set and create_input module starts functioning. It prepares the a priori probability values using a look-up table and gives the values for all the received bits from pout output bus. These values from pout bus are used by initialization module to fill in the message matrix q to check nodes. Thus the q matrix is initialized by a priori probability values for each received codeword bit.

When q matrix is ready, allchecknodes module receive it from its input bus and prepare the r matrix which is the message matrix to both allbitnodes and computebigQ module. Using r matrix coming from rout output bus of allchecknodes module, allbitnodes and computebigQ modules function starting at the same time. Allbitnodes module calculates messages from bitnodes to checknodes and updates the q matrix with the calculated values. At the same time, computebigQ module calculates the extrinsic probabilities of the received bits. Using these probability values, decodebigQ module determines the bit values for the received codeword.

Finally, compute_syndrome module calculates the syndrome vector value using the decoded codeword from decodebigQ value. If this syndrome vector is a zero vector, the decoded codeword is given out as LDPC decoder output. Otherwise, another iteration starts

by enabling allchecknodes module that updates message matrix r using the q matrix which was updated by allbitnodes in the previous iteration. Thus, the decode output is updated either when the syndrome is zero or the configurable maximum number of iterations is reached.

The functionality of input and output ports of the top level entity shown in Table 3.1 is described in first, and then the modules in top level will be explained in detail in the following chapters.

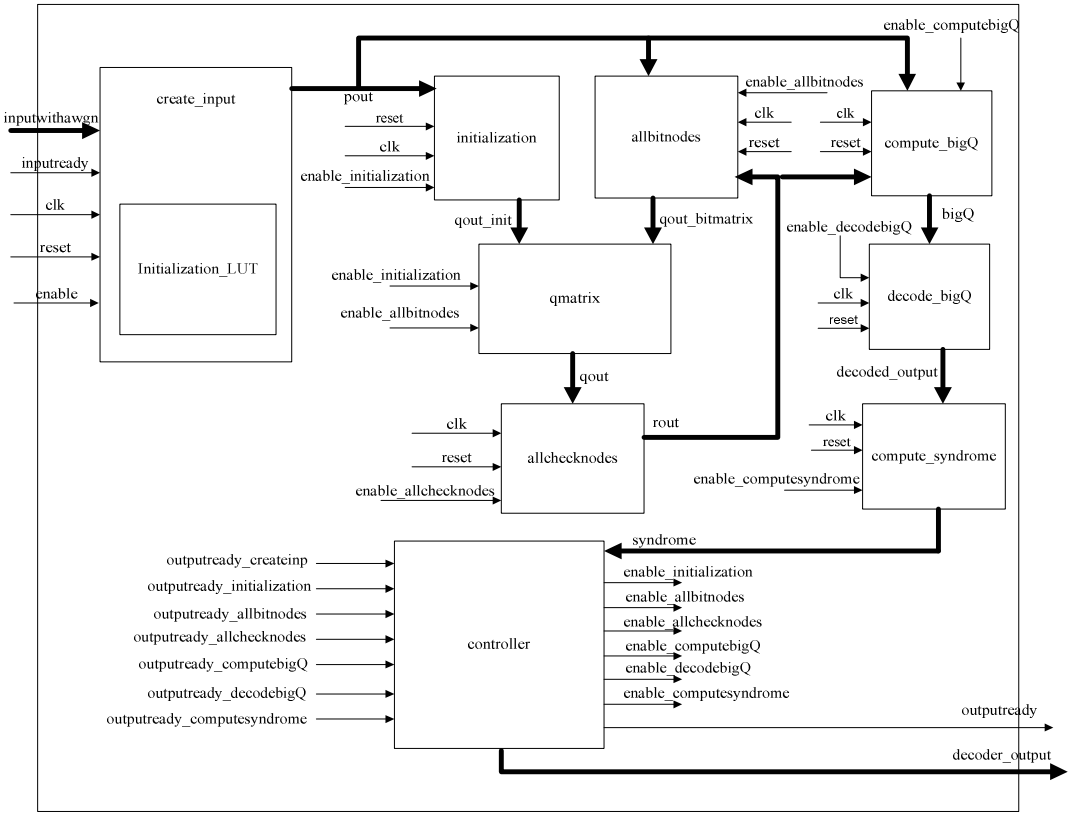

Figure 3.1. Top level diagram of LDPC decoder implementations

Port name	Direction	Number of bits
inputwithawgn	Input	$N \times 8$
inputready	Input	1
Clk	Input	1
Reset	Input	1
decoder_output	Output	N
outputready	Output	1

Table 3.1. Top level entity of probability domain sum product algorithm implementation. (N = number of bits in the codeword)

- **inputwithawgn**: This input port is a vector that brings the bits of received codeword with AWGN. As mentioned in Chapter 2.4, each bit of the codeword to be transmitted becomes -1 or 1 after encoding and given to the channel. The value of the received bits from AWGN channel is limited to range [-3, 3] in this implementation for practical purposes. Since the encoder implementation is out of the scope of this thesis, this range of -3 to 3 is be scaled to 8 bit signed value with maximum value 127 and minimum value -128. Each signed value scaled between -128 to 127 is called a soft bit. Thus inputwithawgn input port will bring N parallel soft bits where N is the number of bits in the codeword.
- **inputready**: This active high input port enables the decoder for each codeword. After decoder receives the rising edge of this input, it samples the parallel inputwithawgn bus composed of N soft bits.
- **clk**:System clock of the decoder is received from this input port.
- **reset**: Asynchronous reset for the decoder synchronous logic is received from this input port.
- **decoder_output:** This is the output port which gives out the decoded codeword.
- **outputready:** This is the output that flags that the decoded output is ready on decoder_output port.

3.2.1. Create_Input Submodule

This module includes a look-up table and a block of registers. The block diagram of the module is shown in Figure 3.2. Input data coming from inputwithawgn input port is sampled in the block of registers called input_reg. Input_reg is composed of N blocks of 8 bit registers where N is the number of bits in the codeword in the design. When inputready is set to '1', create_input module samples inputwithawgn parallel input bus in input_reg block. The next clock cycle the corresponding values from the look-up table are given out from pout output bus.

The look-up table is composed of 60 values corresponding to the received soft input bits with AWGN taking values between -3 and 3. The values between [-3,3] are quantized by 0.1 like X[k] = {-3, -2.9, -2.8, ...,0,..., 2.7, 2.8, 2.9, 3.0} and using these values, the look-up table values are computed as 8-bit signed values like:

$$Y(k) = \frac{1}{1+e^{2(X(k))/\sigma^2}} \times 127 \qquad (3.1)$$

These computed values are apriori probability information calculated for each bit in the received codeword. The soft bits coming from inputwithawgn port which are in the range of [-128,127] (corresponding to the inputs with AWGN in the range [-3,3]) go to the comparators to get their corresponding initialization value in the look-up table. This can be explained by the following example. When inputwithawgn(0)(0) value is received as 100(as 8-bit signed value), this value represents a real bit value with AWGN of 2.36. The corresponding initialization value in look-up table is $Y(2.3) = \frac{1}{1+e^{2(2.3)/\sigma^2}} \times 127 = 1$. Thus when 100 is received from inputwithawgn(0)(0), its corresponding value in the look-up table is found by the comparators and pout(0)(0) is assigned to this value 1.

When pout output vector is assigned with the corresponding values in the look-up table, outputready signal is set to one to indicate that data is ready on pout bus.

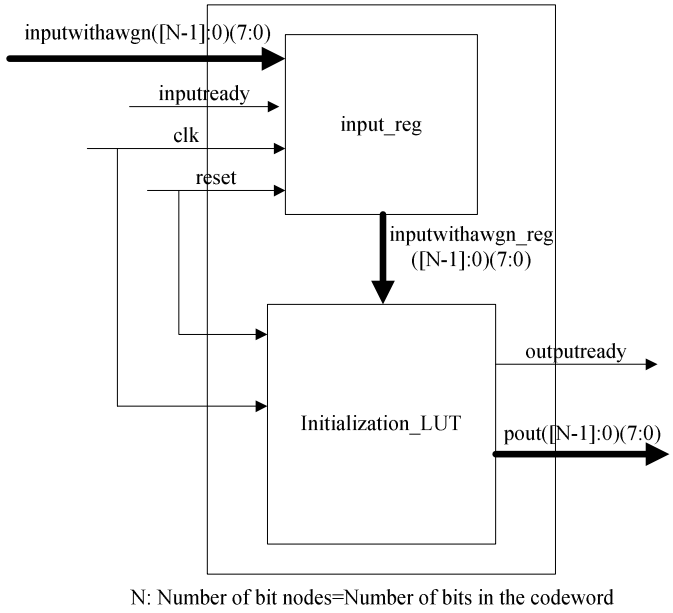

Figure 3.2. Create_input module block diagram

3.2.2. Initialization Module

This module creates initial messages to check nodes using the apriori probability values received from create_input module. It is composed of a register block q_init which is a matrix structure with i rows, and j columns where i is the number of bit nodes and j is the number of check nodes. The elements of the q_init matrix are 8-bit registers.

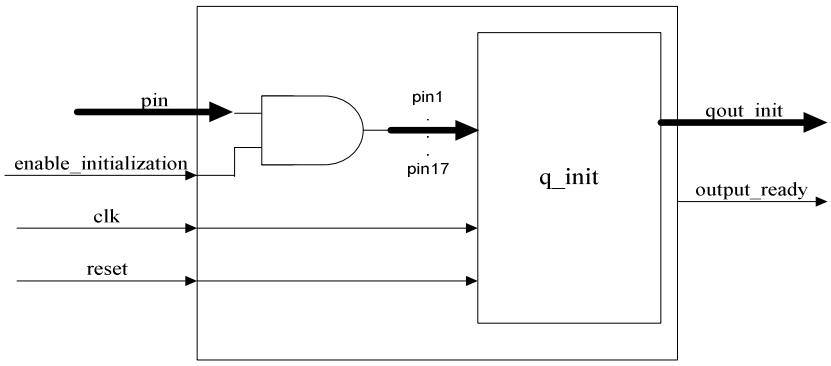

Figure 3.3. Initialization module block diagram

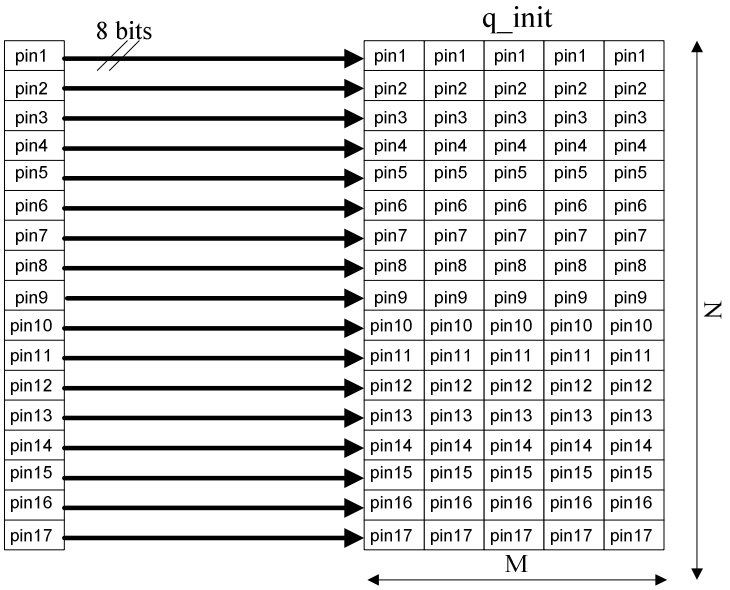

Figure 3.4. Initialization of the message matrix q_init using pin input vector
M = Number of check nodes, N = Number of bit nodes.

Data from create_input module is received by initialization module from pin input port. This input port is a vector of $N \times 8$ bits. When enable_initialization input (coming from controller module) goes high, pin input port (coming from pout output of create_input module) is stored in q_init register block columns like shown in Figure 3.4.).

When all q_init columns are set to pin values, data in q_init are given out from qout_init port which is also a matrix of $N \times M \times 8$ bits. Then output_ready is set to '1' which indicates that the first set of messages to check nodes is ready on port qout_init. qout_init is calculated only once for each received code word and message matrix to checknodes qmatrix is initialized with qout_init before allchecknodes start to function in the first iteration.

3.2.3. Allchecknodes Module

This module creates messages from check nodes to bit nodes using the message values received from qmatrix module. Allchecknodes module is composed of *M* check nodes and each check node is composed of *N* check node sub-modules where *N* is the number of bit nodes and *M* is the number of check nodes. Check node sub-module makes

the calculation of r_{ji} gathering the incoming messages coming from qmatrix if the corresponding element of parity check matrix at column i and row j is one. In this case, the check node sub-module is called active check node sub-module. Check node sub-modules are arranged in matrix structure of $M \times N$ to form allchecknodes module. The connections between the check node sub-modules and the inputs are made according to the parity check matrix structure, like the check node sub-module to calculate r_{ji} , has inputs as the ones in row j of H matrix. Figure 3.5. shows the block diagram of r_{ji} calculation in check node sub-module.

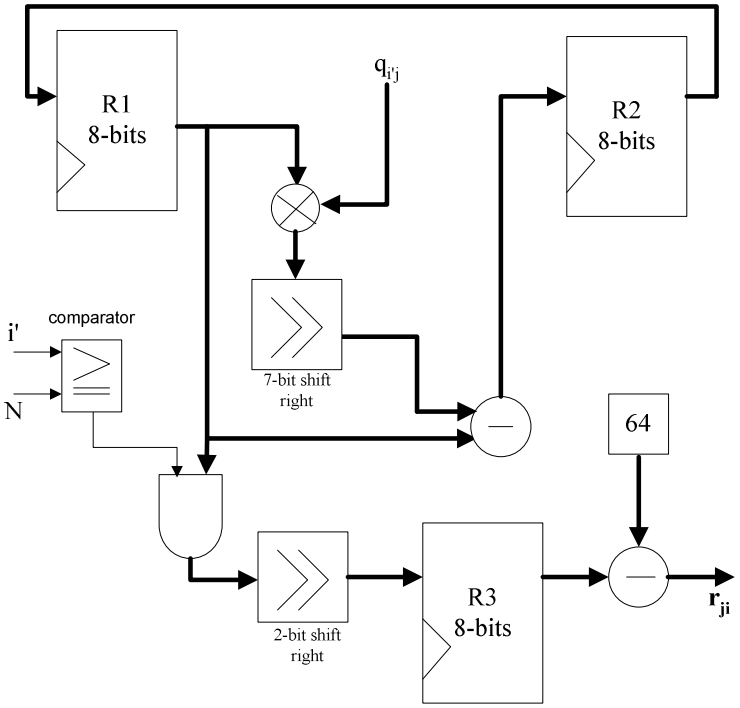

Figure 3.5. Check node sub-module r_{ji} calculation

When enable_allchecknodes signal is set to one by controller module, each check node sub-module starts to calculate its r_{ji} value. There are two subtraction and one multiplication operations in r_{ji} calculation. Divisions by constants are performed by shift operations. After all check node sub-modules calculate their outputs, output_ready indication is set and r output matrix is updated with these outputs.

3.2.4. Allbitnodes Module

Allbitnodes module creates messages from bit nodes to check nodes using the r message matrix received from allchecknodes module. Allbitnodes module is composed of *N* bit nodes that are composed of *M* bit node sub-modules where N is the number of bit nodes and M is the number of check nodes. Bit node sub-module makes the calculation of q_{ij}, gathering the incoming messages coming from r matrix if the corresponding element of parity check matrix at column i and row j is one. In this case, the bit node sub-module is called active bit node sub-module. Bit node sub-modules are arranged in matrix structure of $N \times M$ to form allbitnodes module. The connections between the bit nodes and the inputs are made according to the parity check matrix structure, like the bit node to calculate q_{ij}, has inputs as the ones in column i of H matrix. Figure 3.6. shows the block diagram of q_{ij} calculation in bit node module.

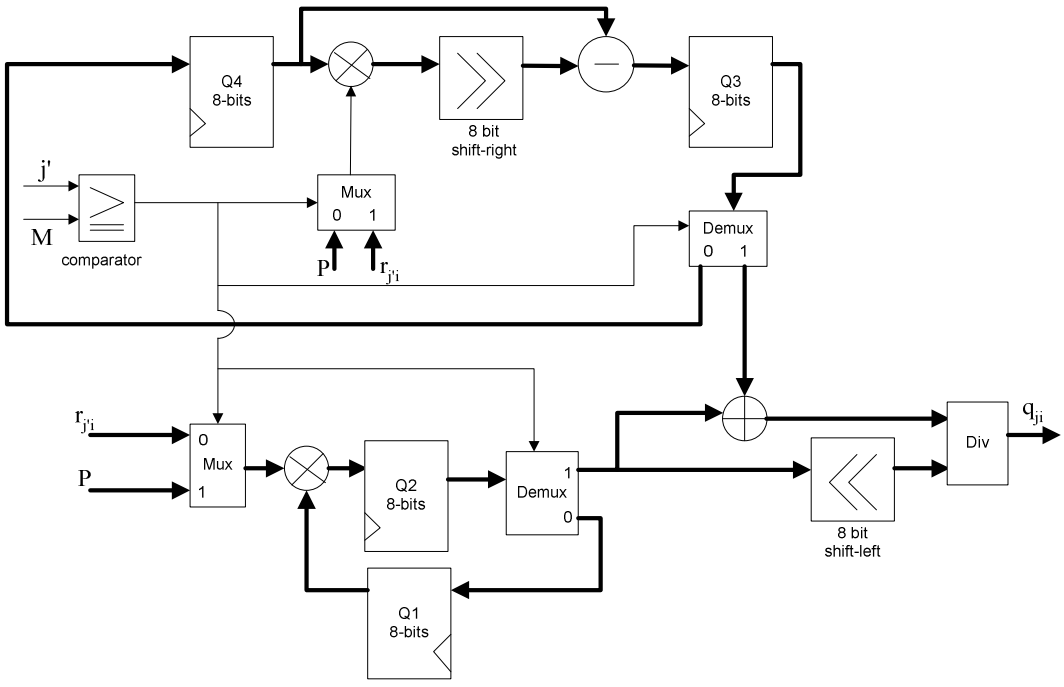

Figure 3.6. Bit node sub-module q_{ij} calculation

When enable_allbitnodes signal is set to one by controller module, each bit node sub-module starts to calculate its q_{ij} value. There are two multiplications, one subtraction

and one division operations in bit node sub-module calculation. For division by variable, Synopsys DesignWare signed divider was used. Divisions by constants are performed by shift operations. After all bit node sub-modules calculate their outputs, output_ready indication is set and q output matrix is updated with these outputs.

3.2.5. ComputebigQ Module

ComputebigQ module computes extrinsic probabilities of decoder output bits. It is composed of computebigQ sub modules which compute Q_i values using the r message matrix received from allchecknodes module. ComputebigQ sub module has the same hardware architecture as bit node sub module. The only difference is r matrix values received as incoming messages include r_{ji} , which was excluded in bit node q_{ij} calculation. ComputebigQ module is composed of *N* computebigQ sub modules where *N* is the number of bitnodes. The connections between the computebigQ submodules and the input messages are made according to the parity check matrix structure, like the computebigQ sub module to calculate Q_i , has inputs as the ones in column i of H matrix.

When enable_computebigQ signal is set to one by controller module, each computebigQ sub module starts to calculate its Q_i value. There are two multiplications, one subtraction and one division operations in bit node calculation. Divisions by constants are performed by shift operations. After all computebigQ sub modules calculate their outputs, output_ready indication is set and Q output vector is updated with these outputs.

3.2.6. DecodebigQ Module

DecodebigQ module determines decoder output bit candidates using Q_i values calculated by computebigQ module. It is composed of N comparators where N is the number of bit nodes. When controller module sets enable_decodebigQ signal to one, each comparator take Q_i^1 value as input and compares it with 64. If Q_i^1 is greater than 64, comparator output is set to one, otherwise set to zero. Consequently, decoder output bit candidates form decoder output codeword candidate $\hat{c}$ which is a vector of N bits and output_ready indication is sent to controller module.

3.2.7. Computesyndrome Module

Computesyndrome module calculates the syndrome vector using the decoded output codeword candidate. Each soft bit $\hat{c}_i$ of the codeword candidate is ANDed with the corresponding bit in each column of transposed parity check matrix H^T. A vector of N bits is received from this operation for each column of H^T. Then each column's elements are XORed between each other and the result bit becomes one of the N elements of the syndrome vector. Figure 3.7 shows calculation of jth bit of syndrome vector. All elements of jth column of H^T are ANDed with corresponding bit of $\hat{c}$.The results are XORed with each other to get jth bit of syndrome vector S.

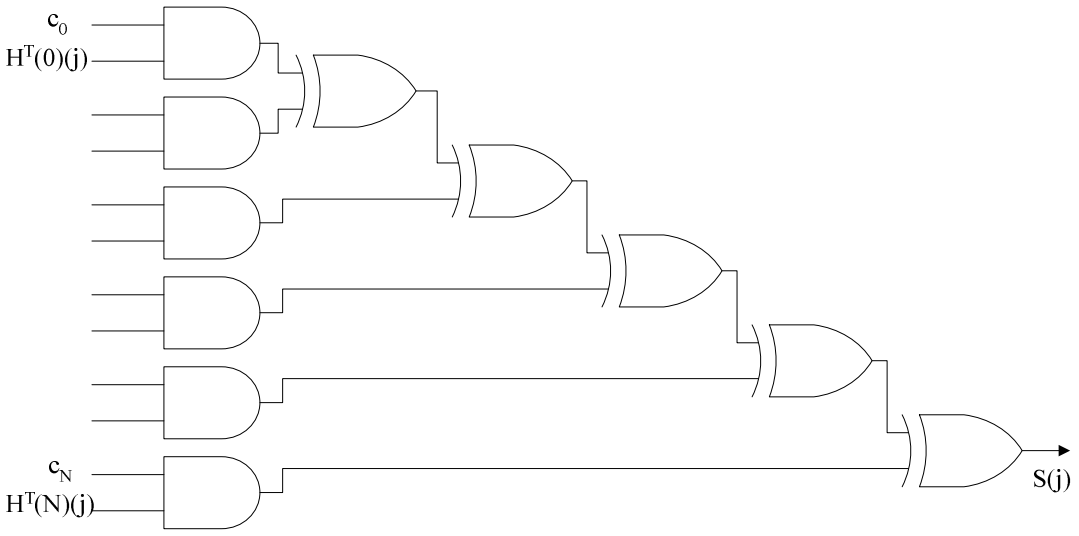

Figure 3.7. Calculation of jth bit of syndrome vector in computesyndrome module

The bit for each column of H^T is computed in parallel. Thus, there are $(N-1)\times M$ XOR gates and $N\times M$ AND gates for syndrome vector S calculation in computesyndrome module. After S is prepared, it is checked if it is zero vector or not by gating all bits using OR gates like $Syndrome = S(0)+S(1)+...+S(N-1)$ where N is the number of bit nodes and M is the number of check nodes. When syndrome output is ready, output_ready to controller module is set.

3.2.8. Controller Module

Controller module enables all modules except create_input module. It also controls the iterations of decoding and giving out the decoder output. It sets the enable signal of a module when it receives output_ready indication from the preceding module. Finally, when computesyndrome finishes and sets its output_ready indication, the controller module checks the value of syndrome. If it is zero, controller stops the iterations and sets the decoder output with the current output of decodebigQ. Otherwise, it starts another iteration by enabling allchecknodes module. Control module controls iterations by checking if the maximum number of iterations parameter is reached. The maximum number of iterations is set to 20 for the implementations done in this thesis. At that point, it stops iterations and gives out the current output of decodebigQ as decoder output. After setting decoder output it sets its output_ready signal which is also the output port of decoder top level. This port indicates that the decoding is finished.

3.3. Implementations of Sum Product Algorithm in Log Domain

Sum product algorithm in log domain is implemented for different variations of the algorithm that are described Section 2.5.2. All these variations are implemented using a parallel architecture. In addition, min-sum approximation method is implemented using semi-parallel and serial architectures in order to compare the properties of these three architecture types. Parallel architecture implementations of log domain algorithm variations use the same top level structure as probability domain algorithm implementation. The blocks of the decoder are connected the same way and the same controller module is used.

The functionality of input and output ports of the top level entity is the same as probability domain algorithm implementation as shown in Table 2.3.The common modules used in log domain algorithm implementations will be explained in detail in the following chapters.

3.3.1. Common Modules of Implementations

In the parallel architecture implementations, same modules are used for different variations of log domain algorithm except for $L(r_{ji})$ calculation. For all algorithm variations, each check node sub-module needs to implement the Equation 2.21. As described in Section 2.5.2., the implementation of $\delta(L(q_{i'j}))$ part in $L(r_{ji})$ calculation brings different variations of sum product algorithm in log domain. There is a common part for check node sub-module functionality for these variations of the algorithm which is expressed in Equation 2.22. The implementation for this common part (which is presented as min-sum approximation method in Section 2.5.2.1.) is described in this section together with other common modules of log domain algorithm implementation. $\delta(L(q_{i'j}))$ approximations for different variations of the log domain algorithm will be described in sections 3.3.3, 3.3.4. and 3.3.5.

3.3.1.1. Create_Input_Module: This module includes a look-up table and a block of registers as it does in probability domain algorithm algorithm. The functionality of the block is the same as its functionality in probability domain algorithm. The only difference is the look-up table content.

The look-up table is composed of 60 values corresponding to received soft input bits with AWGN taking values between -3 and 3. The values between [-3, 3] are quantized by 0.1 like X[k] = {-3, -2.9, -2.8, ...,0,..., 2.7, 2.8, 2.9, 3.0} and using these values, the look-up table values are computed as 8-bit signed values like:

$$Y(k) = \frac{1}{3\sigma^2} \times X(k) \times 127 \qquad (3.2)$$

These computed values are LLR information calculated for each bit in the received codeword. The soft bits coming from inputwithawgn port which are in the range of [-128,127] (corresponding to the inputs with AWGN in the range [-3, 3]) go to the comparators to get their corresponding initialization value in the look-up table. These values in look-up table are given out as $L(p_i)$ to initialization module.

3.3.1.2. Initialization Module: Initial message matrix to allchecknodes module is created in this module like in probability domain algorithm. Apriori LLR values $L(p_i)$ coming from create_input module is used in updating $L(q)$ matrix in the same way as in probability domain algorithm.

3.3.1.3. Allchecknodes Module: Allchecknodes module creates the message matrix to allbitnodes module in LLR domain. This module is composed of check nodes and each check node is composed of *N* check node sub-modules where *N* is the number of bit nodes. Check node sub-module makes the calculation of $L(r_{ji})$ gathering the incoming messages coming from qmatrix if the corresponding element of parity check matrix at column i and row j is one. In this case, the check node sub-module is called active check node sub-module.

For parallel architecture implementation, check node sub-modules are structured in the same way as probability domain algorithm implementation. The only difference with probability domain implementation is the functionality of check node sub-modules; that they compute $L(r_{ji})$ values instead of r_{ji}. Check node sub-modules are arranged in matrix structure of $M \times N$ to form allchecknodes module. The connections between the check node sub-modules and the inputs are made according to the parity check matrix structure, like the check node sub-module to calculate $L(r_{ji})$, has inputs as the ones in row j of H matrix.

In semi-parallel architecture implementation, check node sub-modules are arranged in a matrix structure of $\frac{M}{2} \times N$ to form allchecknodes module where N is the number of bit nodes and M is the number of check nodes. The number of check node sub-modules is decreased to half compared to parallel implementation. For decoding of one codeword, each check node functions twice for each iteration. In serial architecture, there are *N* check node sub-modules that form one check node's functionality. Each check node sub-module works *M* times for each iteration in decoding process of a received codeword. The details

of the construction of allchecknodes module in serial and semi-parallel architectures will be described in more detail in Section 3.3.2.2. and Section 3.3.2.3.

Two calculations are made to compute $L(r_{ji})$ messages. First calculation is the XORing the sign values of $L(q_{i'j})$ inputs connected to the check node. The second calculation is finding the minimum of $L(q_{i'j})$ inputs connected to the check node sub-module. This is made using comparators which find the minimum absolute value among their inputs. Figure 3.8 demonstrates the sign calculation part of check node sub-module and Figure 3.9 demonstrates the minimum finding part of check node sub-module.

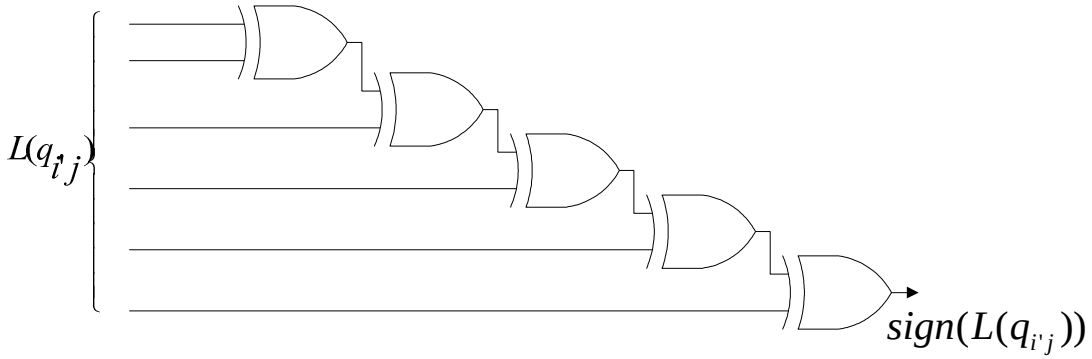

Figure 3.8. Sign values part of $L(r_{ji})$ calculation in check node sub-module

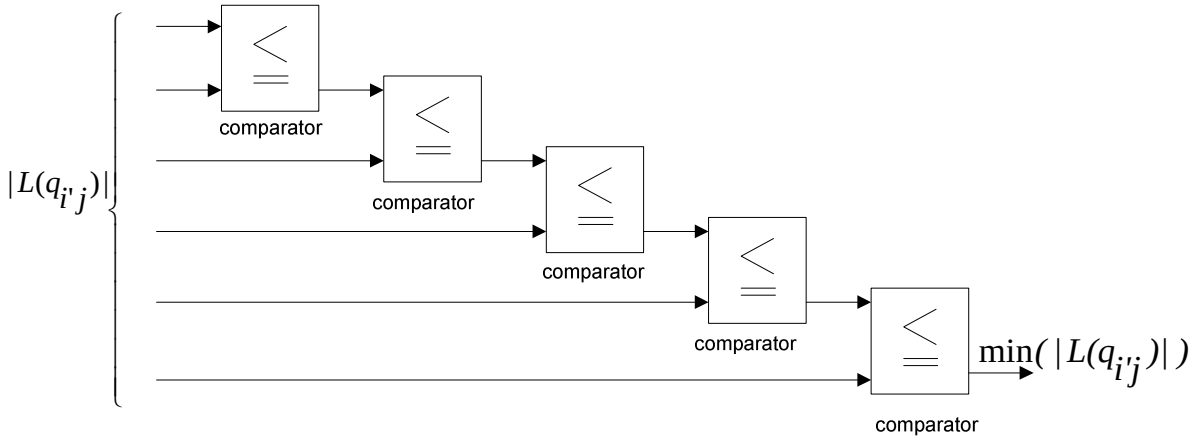

Figure 3.9. Minimum finding part of $L(r_{ji})$ calculation in check node sub-module

3.3.1.4. Allbitnodes Module: Allbitnodes module creates the message matrix to allchecknodes module in LLR domain. This module is composed of bit nodes and each bit node is composed of *M* bit node sub-modules where *M* is the number of check nodes. Bit node sub-module makes the calculation of $L(q_{ij})$ gathering the incoming messages coming from rmatrix if the corresponding element of parity check matrix at column i and row j is one. In this case, the bit node sub-module is called active bit node sub-module.

For parallel architecture implementation, bit node sub-modules are structured in the same way as probability domain algorithm implementation. The only difference with probability domain implementation is the functionality of bit node sub-modules; that they compute $L(q_{ij})$ values instead of q_{ij} .Bit node sub-modules are arranged in matrix structure of $N \times M$ to form allbitnodes module. The connections between the bit node sub-modules and the inputs are made according to the parity check matrix structure, like the check node sub-module to calculate $L(q_{ij})$, has inputs as the ones in column i of H matrix.

In semi-parallel architecture implementation, bit node sub-modules are arranged in a matrix structure of $\frac{N}{2} \times M$ to form allbitnodes module where N is the number of bit nodes and M is the number of check nodes. The number of bit node sub-modules is decreased to half compared to parallel implementation. For decoding of one codeword, each bit node functions twice for each iteration. In serial architecture, there are *M* bit node sub-modules that form one check node's functionality. Each bit node sub-module works *N* times for each iteration in decoding process of a received codeword. The details of the construction of allbitnodes module in serial and semi-parallel architectures will be described in more detail in Section 3.3.2.1.

Each bit node sub-module calculates its $L(q_{ij})$ value by adding its $L(r_{j'i})$ inputs and the apriori LLR value $L(p_i)$.Figure 3.10 shows the bit node sub-module architecture in min-sum approximation implementation.

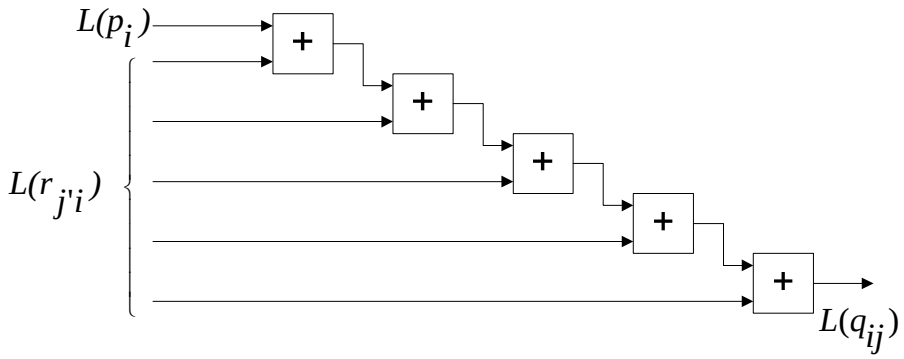

Figure 3.10. Bit node sub-module structure in min-sum approximation algorithm

3.3.1.5. ComputebigQ Module: ComputebigQ module computes extrinsic LLR values of decoder output bits. It is composed of computebigQ submodules. In parallel implementation, computebigQ module is organized the same way as the implementation of sum product algorithm in probability domain. ComputebigQ module is composed of *N* computebigQ sub modules where *N* is the number of bitnodes. The connections between the computebigQ submodules and the input messages are made according to the parity check matrix structure, like the computebigQ sub module to calculate $L(Q_i)$, has inputs as the ones in column i of H matrix. ComputebigQ submodules compute $L(Q_i)$ values using the $L(r)$ message matrix received from allchecknodes module. ComputebigQ sub module has the same hardware architecture as bit node sub module. The only difference is $L(r)$ matrix values received as incoming messages include $L(r_{ji})$, which was excluded in bit node $L(q_{ij})$ calculation.

In semi-parallel implementation, $\frac{N}{2}$ computebigQ submodules are used in computebigQ module. In this case each computebigQ submodule functions twice for each iteration of decoding a received codeword. In serial implementation, computebigQ module is composed of only one sub-module, which functions N times in each iteration. The details of the construction of computebigQ module in serial and semi-parallel architectures will be described in more detail in Section 3.3.2.1.

3.3.1.6. DecodebigQ Module: DecodebigQ module determines decoder output bit candidates using $L(Q_i)$ values calculated by computebigQ module. It is composed of N

comparators where N is the number of bit nodes. When controller module sets enable_decodebigQ signal to one, each comparator take $L(Q_i^1)$ value as input and compares it with 0 since the comparison is done for LLR values. If $L(Q_i^1)$ is greater than 0, comparator output is set to zero, otherwise set to one as shown in Equation 2.18. Like in probability domain sum-product algorithm implementation, decoder output bit candidates form decoder output codeword candidate $\hat{c}$ which is a vector of N bits and output_ready indication is sent to controller module.

3.3.1.7. Computesyndrome Module: Computesyndrome module calculates the syndrome vector using the decoded output codeword candidate. It has exactly the same structure as probability domain sum-product algorithm, taking decoder output codeword candidate $\hat{c}$ and transposed parity check matrix H^T as input and calculating the syndrome.

3.3.1.8. Controller Module: Controller module has the same structure as sum-product algorithm in probability domain. It sets the enable signals of all modules except create_input module and controls the iterations of decoding and giving out the decoder output. When the iterations are finished, it sets the decoder output and its output_ready signal which indicates that the decoding is finished.

3.3.2. Min-Sum Approximation Method Implementation

It is observed in the literature that parallel, semi-parallel and serial architectures are used for LDPC decoder implementations [14], [18], [19], [22], [25], [27], [28], [29], [31], [32], [34], [37], [38]. Min-sum approximation algorithm is implemented for parallel, semi-parallel and serial architectures in order to compare the advantages and disadvantages of each architecture. In this section, the implementation details of these three types of implementations are presented.

3.3.2.1. Parallel Architecture Implementation: Parallel architecture implementation for min-sum approximation method uses the same top level structure as probability domain algorithm implementation for parallel architecture. The blocks of the decoder are connected the same way and the same controller module is used. For parallel architecture

implementation, each module is designed using a parallel architecture composed of its own sub-modules like described in Section 3.3.1. [19], [22], [28], [29], [31], [32], [37], [38].The functionality of input and output ports of the top level entity is the same as probability domain algorithm implementation as shown in Table 2.3.

3.3.2.2. Semi-Parallel Architecture Implementation: In the literature, it is observed that semi-parallel architectures are more commonly used, since they decrease the number of functional units to a considerable level [14], [18], [19], [21], [27], [34]. In this thesis, semi-parallel architecture implementation decreases the number of functional units to half. This means, semi parallel architecture implementation includes $\frac{N}{2} \times M$ bit node sub modules, $\frac{M}{2} \times N$ check node sub modules and $\frac{N}{2}$ computebigQ sub modules.

Check nodes functionality is implemented in a module called allchecknodes like in parallel implementation. In allchecknodes module, $\frac{M}{2} \times N$ check node sub modules work in parallel. Each check node is enabled twice for one iteration of decoding. For example, in parallel implementation check node 0 is enabled once but in semi-parallel implementation, it is enabled twice. Its first functioning replaces the functioning of check node 0 of parallel implementation and its second functioning replaces the functioning of check node $\frac{M}{2}$. Similarly, check node C in semi-parallel implementation replaces check node 1 and check node $\frac{M}{2} + C$. A control logic is used in allchecknodes module to enable the $\frac{M}{2} \times N$ check node sub modules. This logic enables check nodes first by the enable signal coming from controller module to prepare the first half of the $L(r_{ji})$ matrix. Then after receiving the output_ready signal from checknodes for this first functioning, it enables the check nodes again to prepare the second half of the $L(r_{ji})$ matrix. After the second functioning finishes, the output_ready signal of allcheknodes module is set.

The same strategy exists for bit nodes and computebigQ functionality. Similarly allbitnodes and computebigQ modules exist that are enabled twice by the controlling logic

inside them. Allbitnodes module's control logic enables bit nodes twice; first time it creates the first half of $L(q_{ij})$ matrix and second time it creates the second half of the matrix. At the same time, computebigQ module's control logic enables computebigQ sub-modules twice, first time it creates the first half of $L(Q_i)$ array and second time it creates the second half of the array.

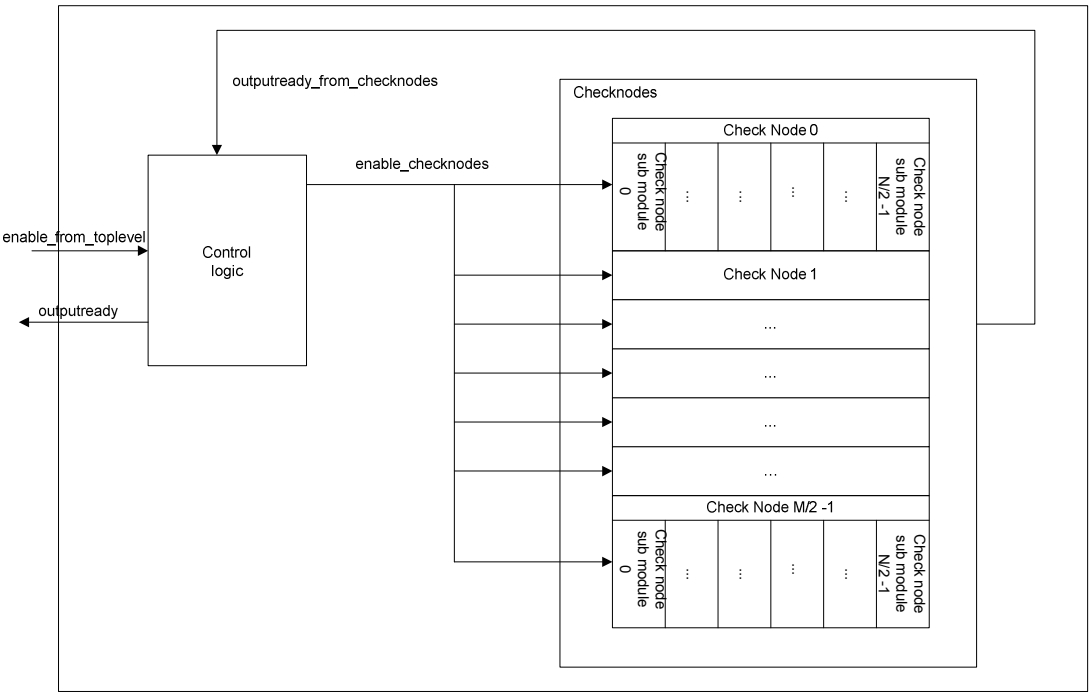

Figure 3.11. Semi-parallel architecture of allchecknodes module

The rest of the modules are implemented like described in Section 3.3.1. The functionality of input and output ports of the top level entity is the same as probability domain algorithm implementation as shown in Table 2.3.

3.3.2.3. Serial Architecture Implementation: In serial architecture implementation, there is only one functional unit for each functional unit type [22], [25], [37]. This means there is one check node, one bit node and one computebigQ sub module in the decoder. These functional units work in a serial architecture under the control of a block similar to the control logic described in semi-parallel architecture.

The check node functions M/2 times for each iteration. The control logic enables the check node first with the enable signal coming from top-level controller module. Then for the next times, it enables the check node using the output_ready signal generated from the previous functioning. After the check node functions M/2 times, output_ready signal of the check node module is sent to the top-level controller module.

Similarly, bit node and computebigQ modules are enabled by the control logic. Bit node and computebigQ sub module are both enabled N/2 times. After the last functioning, the control logic sets the related output_ready signals to top-level controller module.

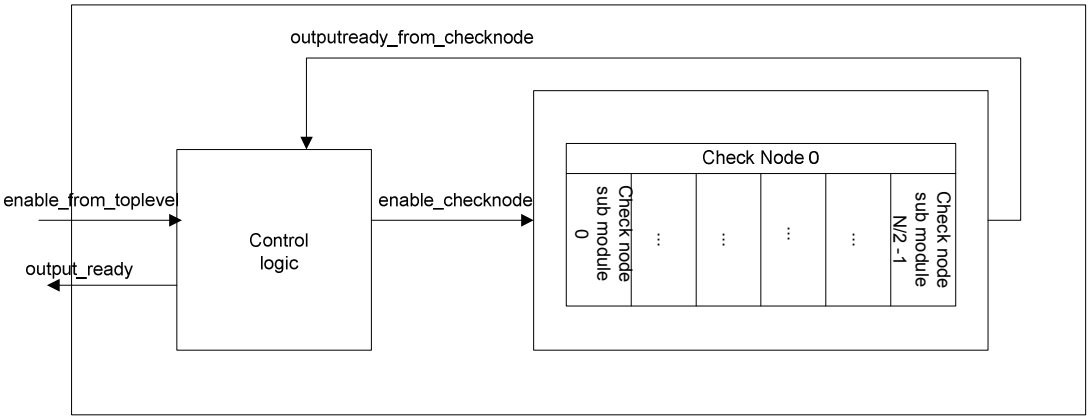

Figure 3.12. Check node module in serial architecture implementation

The rest of the modules are implemented like described in Section 3.3.1. The functionality of input and output ports of the top level entity is the same as probability domain algorithm implementation as shown in Table 2.3.

3.3.3. Look-up Table Approximation Method Implementation

Look-up table approximation method is implemented using parallel architecture. The top-level structure is the same as probability domain algorithm implementation for parallel architecture. The blocks that the implementation is composed of are the common modules described in Section 3.3.1. except for allchecknodes module. In allchecknodes module, besides the common part used in min-sum approximation, $\delta(x)$ approximation is

implemented using the look-up table demonstrated in Section 2.5.2.2. Top-level structure for check node sub-module unit is shown in Figure 3.13.

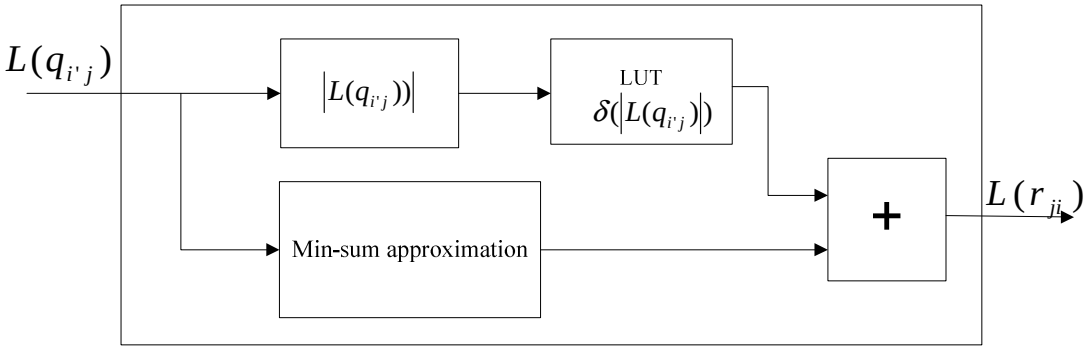

Figure 3.13. Check node sub-module structure for look-up table approximation implementation

3.3.4. Piecewise Linear Approximation Method Implementation

Similar to look-up table approximation method implementation, piecewise linear approximation method is implemented using parallel architecture with the common blocks described in Section 3.3.1. except for allchecknodes module. In allchecknodes module, in addition to the common part used in min-sum approximation, $\delta(L(q_{i'j}))$ is implemented using 5 linear equations shown in Section 2.5.2.3. for 5 regions of x. Since the coefficients of the equations are chosen as powers of 2, multiplication with these coefficients is implemented using shift registers. Like shown in Figure 3.14, there is a decoder which chooses the equation to calculate the $\delta(L(q_{i'j}))$ approximation according to $L(q_{i'j})$ value. According the decoder output, the $L(q_{i'j})$ value is sent to the shift register. A look-up table holds the constant parts of the linear equations. The constant part of the linear equation to be implemented is added to shift register output to form $\delta(L(q_{i'j}))$ approximation. Finally, $\delta(L(q_{i'j}))$ approximation is added to min-sum approximation value to form $L(r_{ji})$ value.

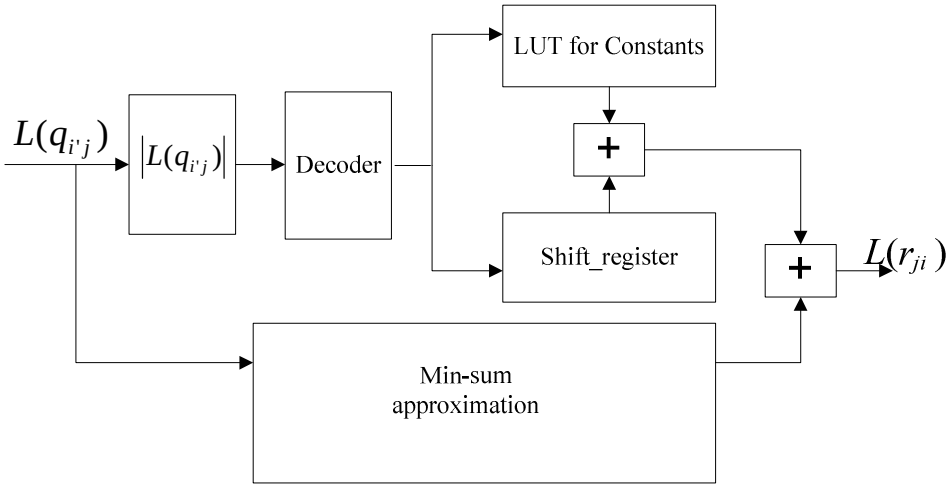

Figure 3.14. $L(r_{ji})$ calculation in piecewise linear approximation method

3.3.5. Optimum Linear Approximation Method Implementation

In optimum linear approximation method implementation, $\delta(x)$ function is implemented using the linear equation mentioned in Section 2.5.2.4. For ease of implementation, the coefficient 0.24 is changed to the nearest power of two 0.25. By this way, only one adder and shift register is enough for implementation of $\delta(x)$ approximation.

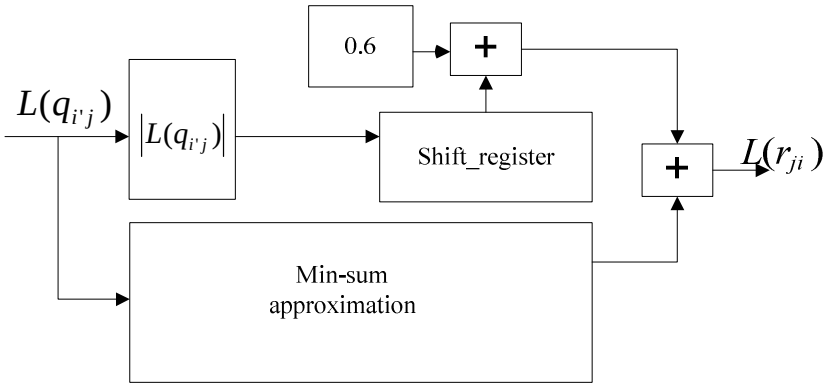

Figure 3.15. $L(r_{ji})$ calculation in optimum linear approximation method

4. VERIFICATION AND BIT ERROR RATE ANALYSIS

Implementations of LDPC decoder are functionally verified in a VHDL verification environment using Cadence NCSIM tool. As stated in the previous sections, sum product algorithm is not suitable for hardware implementations because of the drawbacks explained in Section 2.5.1. The implementation is done for this algorithm but bit error analysis is not done. For all of the log domain algorithm implementations, the same verification environment is used; including top level VHDL testbench of the design and input stimuli for top level generated using MATLAB.

For input stimuli generation, a 10^6 random message words are generated first. These message words are encoded to codewords in a MATLAB program which makes the matrix multiplication c = m·G, where G generating matrix is derived from LDPC codes' parity check matrix.Each bit of the codeword is converted to integer numbers {-1, +1} according to $x_i = (-1)^{ci}$ formula in this MATLAB program. Codewords are collected in an array of size 10^6.After the addition of AWGN (by using awgn function of MATLAB), each element of this array is converted to 8-bit signed numbers between -128 and 127. Using this way, input stimuli composed of 10^6 random message words is received. At the same time, the 10^6 codewords generated as the result of c = m·G (before AWGN addition) are dumped to a text file which will be the reference data to be compared to the decoder outputs. This process is repeated for different signal to noise ratio (SNR) values that is given as a parameter to awgn function. Finally 9 input files for 9 different SNR values are received. These files are used in frame error rate (FER) analysis of decoder implementations using Monte Carlo analysis. For this purpose, a simulation is run for each input pattern file by feeding the data in the file to the design under test in the VHDL testbench. The outputs were dumped to a text file in the VHDL testbench. This test file content should contain 10^6 decoded code words which are compared to the code words in the reference data file.

For each different LDPC decoder implementation, 9 simulations are run and the simulation outputs are compared to reference codeword files to find how many bits differ

between decoded received codewords and reference codewords. In each simulation, $17x10^6$ bits are processed.

Another important result received from the simulations is the latency information which shows how many cycles it takes for the decoder to decode a codeword. In sum-product algorithm, decoding is an iterative process and the number of iterations to decode a codeword for the implementations in this thesis can take 1 to 20 iterations. That's why in order to compare the implementations in terms of latency; the latency of only one iteration is compared between the implementations. Table 4.1 shows the latency information for each implementation.

Implementation Type	**Latency in Modules (Clock Cycles)**	**Total Decoding Latency (Clock Cycles)**
Parallel Architecture Implementation (Min Sum Approximation Method , Look-up Table Approximation Method , Piecewise Linear Approximation Method, Optimum Linear Approximation)	Createinput Module: 1 Initialization :1 Allchecknodes :20 Allbitnodes : 15 DecodebigQ : 17 ComputeSyndrome: 12 Control: 1	67
Semi-Parallel Architecture Implementation (Min Sum Approximation Method)	Createinput Module: 1 Initialization :1 Allchecknodes :44 Allbitnodes : 34 DecodebigQ : 17 ComputeSyndrome: 12 Control: 1	110
Serial Architecture Implementation (Min Sum Approximation Method)	Createinput Module: 1 Initialization :1 Allchecknodes :254 Allbitnodes : 274 DecodebigQ : 17 ComputeSyndrome: 12 Control: 1	560

Table 4.1. Decoding latency values for decoder implementations

In Table 4.1., latency values are given for modules of each implementation. Since allbitnodes module and computebigQ modules function concurrently, only the latency of allbitnodes is given because it has more latency than computebigQ module.

5. SYNTHESIS AND POWER ANALYSIS

Decoder implementations are synthesized using Synopsys Design Compiler tool. A CMOS 65 nm technology library is used in the synthesis of the circuits. As the result of synthesis, area report of each implementation is received. Table 5.1. shows the area results for each implementation in terms of NAND gate area.

Architecture Type	Approximation Method	Combinational Area(KGates)	Sequential Area	Total Area
Parallel Architecture	Min Sum Approximation Method	144	182	326
	Look-up Table Approximation Method Implementation	146	182	328
	Piecewise Linear Approximation Method Implementation	152	183	335
	Optimum Linear Approximation Method Implementation	149	183	332
Semi-parallel Architecture	Min Sum Approximation Method	89	133	222
Serial Architecture	Min Sum Approximation Method	13	76	89

Table 5.1. Area Values of LDPC Decoder Implementations

Power efficiency is an important issue in the analysis of different LDPC decoder implementations [36].After that estimated power consumption of each circuit is found for each implementation using Synopsys Power Compiler tool. Table 5.2 shows the power estimation values for each implementation. The operating voltage is chosen like 1.2V and temperature is chosen 25C. The clock frequency used in power estimation is 100 MHz. Average toggling rate for the gates in implementations are taken as five per cent. Three different types of power values are compared for each implementation.

- Internal Power: Internal power is any power dissipated within the boundary of a cell. During switching, a circuit dissipates internal power by the charging or discharging of any existing capacitances internal to the cell. Internal power also includes power dissipated by a momentary short circuit between the P and N transistors of a gate, called short-circuit power.
- Switching Power: Switching power is the power dissipated by the charging and discharging of the load capacitance at the output of the cell. The total load capacitance at the output of a driving cell is the sum of the net and gate capacitances on the driving output. Because such charging and discharging are the result of the logic transitions at the output of the cell, switching power increases as logic transitions increase.
- Leakage Power: Leakage power is dissipated when the gate is not switching, that is when it is inactive or static. There are two leakage power sources [41]. The first and the major one is leakage current caused by the sub-threshold currents of the transistors. There is leakage current in MOS transistors even when V_{GS} is below threshold voltage.The other leakage power source is leakage current flowing through the reverse biased diode junctions of the transistors located between the source and the substrate.

Architecture Type	Approximation Method	Internal Power (uW)	Switching Power (uW)	Lekage Power (uW)
Parallel Architecture	Min Sum Approximation Method	8965	4694	10.67
	Look-up Table Approximation Method Implementation	9020	4723	10.74
	Piecewise Linear Approximation Method Implementation	9212	4824	10.97
	Optimum Linear Approximation Method Implementation	9130	4780	10.87
Semi-Parallel Architecture	Min Sum Approximation Method	6105	3196	7.27
Serial Architecture	Min Sum Approximation Method	2447	1281	2.91

Table 5.2. Power estimation values for LDPC decoder implementations

6. SUMMARY OF FINDINGS

As the result of simulations, FER performance improvement that $\delta(x)$ approximation methods bring compared to min-sum approximation method is observed. At the same time, it is observed from synthesis results that $\delta(x)$ approximation methods do not bring important hardware overhead compared to min-sum approximation method implementation. Figure 6.1. shows the FER performances of LDPC decoder implementations. Piecewise linear approximation implementation is found to give the best error correcting performance. On the other hand, it has the biggest area and power consumption. Look-up table approximation method is the second in FER performance but it is has the smallest area and power consumption after min-sum approximation method. Optimum linear approximation method implementation has larger area and worse FER performance than look-up table approximation implementation.

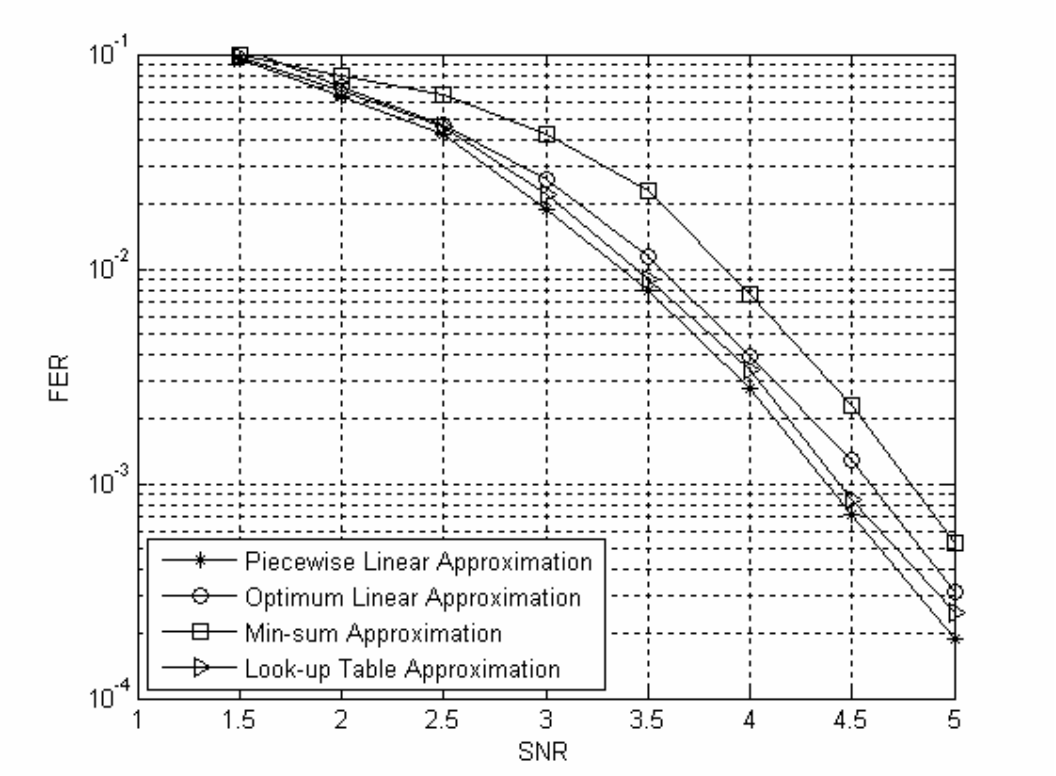

Figure 6.1. FER performances of LDPC decoders for different SNR values

Consequently, it can be stated that if a comparison is made among parallel architecture implementations, the most suitable decoder implementation should be chosen taking into account the trade-off between gate count and FER performance. If smaller gate count is more necessary, look-up table approximation should be chosen. This choice brings smaller area and less power consumption with a very small degradation in FER performance compared to piecewise linear approximation implementation. Otherwise, if the FER performance is more important, piecewise linear approximation implementation should be chosen.

Another evaluation can be made for parallel, semi-parallel and serial architecture implementations of min-sum approximation method. It can be seen that parallel implementation has a gate count 326 K composed of 144 K combinational part and 182 K sequential part. For semi-parallel implementation the gate count decreases to 222 K with 89 K combinational part and 133 K sequential part. Finally, for serial implementation, total number of gates is 89K with 13K combinational part and 76K sequential part. Number of functional units in semi-parallel implementation is the half of number functional units in parallel implementation but decreasing of gate count due to this property is more significant in combinational part than in sequential part. Similarly, there is only one of each functional unit (check node, bit node and computebigQ submodule) in serial implementation. It is observed that combinational part decreases approximately 90 percent compared to parallel implementation but sequential part does not decrease this much. This is due to the reason that even if the number of functional units is decreased; an important amount of registers to hold the message values should exist.

Besides, if the decoding latencies are considered, it can be observed that serial implementation has a big latency like 560 cycles which is approximately eight times the latency of parallel implementation. Semi-parallel architecture implementation has 110 cycles of decoding latency which is more acceptable compared to serial implementation. Therefore, semi-parallel architecture is found to be the optimum for hardware implementation of LDPC decoder. It has less gate count compared to parallel architecture which brings advantage both in area and power consumption and it has smaller latency compared to serial architecture.

As the result, piecewise linear approximation implementation using semi-parallel architecture is found to be the optimum implementation if FER performance is has priority. If gate count and power consumption is more important, look-up table approximation implementation using semi-parallel architecture is more convenient for hardware implementation.

REFERENCES

1. Gallager R. G., “Low-Density Parity-Check Codes”, M.I.T. Press, Cambridge, 1963.

2. MacKay and R. M. Neal, “Near Shannon Limit Performance of Low-Density Parity-Check Codes” *Electronic Letters*, Volume 33, Issue 6, pp. 457 – 458, March 1997.

3. Reed S. and X.Chen, *Error – Control Coding for Data Networks*, Kluwer Academic Publishers, Boston, 1999.

4. Weldeselassie Y.T. and T.Y.V Yolande, “Error Correcting Codes over Graphs”, Abdus Salam International Center for Theoretical Physics, 2003.

5. Richardson T. J. and R. Urbanke, “The capacity of low-density parity check codes under message passing decoding”, *IEEE Transactions on Information Theory*, Volume 47, pp. 599 - 618, February 2001.

6. Chung Y., G. D. Forney and Thomas J. Richardson, “On the Design of Low-Density Parity-Check Codes within 0.0045 dB of the Shannon Limit”, *IEEE Communication Letters*, Volume 5, pp. 58 - 60, February 2001.

7. Richardson T. J., M. A. Shokrollahi, and R. Urbanke, “Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes”, *IEEE Transactions on Information Theory*, Volume 47, pp. 619–637, February 2001.

8. Levine B. and R.Taylor, “Implementation of Near Shannon Limit Error-Correcting Codes Using Reconfigurable Hardware”, *2000 - IEEE Symposium on Field-Programmable Custom Computing Machines,* pp. 217-226.

9. Futaki H. and T. Ohtsuki, “Low –Density Parity Check (LDPC) Coded OFDM Systems”, *IEEE VTS 54th Vehicular Technology Conference,* Volume1, pp. 82-86, 2001.

10. Murphy G. and E.M. Popovoci, “Design and Implementation of a Parameterizable LDPC Decoder IP Core”, *International Conference on Microelectronics*, Volume 2, pp.747- 750, May 2004.

11. Hu Y. and E.Eleftheriu, “Efficient Implementations of the Sum-Product Algorithm for Decoding LDPC codes”, *Global Telecommunications Conference 2001*,Volume 2, pp. 1036, 2001.

12. McKay D.J.C, “Good error-correcting codes based on very sparse matrices”, *IEEE Transactions On Information Theory*, Volume 45, pp.399 – 431, March 1999.

13. Richter G., G.Schmidt and M.Bossert, “Optimization of a reduced-complexity decoding algorithm for LDPC codes by density evolution”, *IEEE International Conference on Communications 2005*, Volume 1, pp.642 – 646, May 2005.

14. Karkooti M. and J.R. Cavallaro, **“**Semi-parallel reconfigurable architectures for real-time LDPC decoding”, *IEEE International Conference on Information Technology 2004*, Volume 1, pp. 579 – 585, April 2004.

15. Lee W.L. and A.Wu, “VLSI implementation for low density parity check decoder”, *IEEE International Conference on Electronics, Circuits and Systems 2001*, Volume 3, pp. 1223-1226, 2001.

16. Theocharides T., G.Link and E.Swankoski, “Evaluating alternative implementations for LDPC decoder check node function”, *IEEE Computer society Annual Symposium on VLSI 2004*, pp.77-82, 2004.

17. Wymeersch H., H. Steendam and M. Moeneclaey, “Log-domain decoding of LDPC codes over GF(q)”, *IEEE International Conference on Communications 2004*, Volume 2, pp. 772-776, June 2004.

18. Yukei P. and Y.Liuguo, “Design of irregular LDPC codec on a single chip FPGA”, IEEE 6th Circuits and Systems Symposium on Emerging Technologies: Frontiers of Mobile and Wireless Communication 2004, Volume 1, pp. 221-224, June 2004.

19. Fanucci L. and F.Rossi, “A throughput/complexity analysis for the VLSI implementation of LDPC decoder”, *IEEE International Symposium on Signal Processing and Information Technology 2004*, pp. 409 -412, December 2004.

20. Karkooti M. and P. Radosavljevic, “Configurable, High Throughput, Irregular LDPC Decoder Architecture: Tradeoff Analysis and Implementation”, *International Conference on Application-specific Systems, Architectures and Processors 2006*, pp. 360-367, September 2006.

21. Yang L. and S.Manyuan, “An FPGA implementation of low-density parity-check code decoder with multi-rate capability”, *Proceedings of the ASP Design Automation Conference 2005*, Volume 2, pp. 760 – 763, January 2005.

22. Liao E., Y. Engling and B. Nikolic, “Low-density parity-check code constructions for hardware implementation”, *IEEE International Conference on Communications*, Volume 5, pp. 2573-2577, June 2004.

23. Lechner G., A. Bolzer and J, Sayir, “Implementation of an LDPC decoder on a vector signal processor”, *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers 2004,* Volume 1, pp. 549 – 553, November 2004.

24. Huang X., S.Ding and Z.Yang, “Fast Min-Sum Algorithms for Decoding of LDPC over GF(q)”, *IEEE Information Theory Workshop 2006*, pp. 96-99, October 2006.

25. Darabiha A., A. Carusone and A.C. Kschischang, “A bit-serial approximate min-sum LDPC decoder and FPGA implementation”, *IEEE International Symposium on Circuits and Systems*, pp. 4, May 2006.

26. Ruiyuan H., L. Jing and E. Kurtas, “Quantization and quantization sensitivity of soft-output product codes for fast-speed applications”, *IEEE/Sarnoff Symposium on Advances in Wired and Wireless Communication 2004,* pp.175-178, April 2004.

27. Olcer S., “Decoder architecture for array-code-based LDPC codes”, *Global Telecommunications Conference 2003,* Volume 4, Issue 1, pp. 2046-2050, December 2003.

28. Nagarajan V., N. Javakumar and S. Khatri, “High-throughput VLSI implementations of iterative decoders and related code construction problems”, *IEEE Global Telecommunications Conference 2004*, Volume 1, pp. 361-365, Dec. 2004.

29. Rawi A. and J. Cioffi, “A highly efficient domain-programmable parallel architecture for iterative LDPCC decoding”, *International Conference on Information Technology: Coding and Computing 2001*, pp.569–577, April 2001.

30. Jianguang Z., F. Zarkeshvari and A.H. Banihashemi, “On implementation of min-sum algorithm and its modifications for decoding low-density Parity-check (LDPC) codes”, *IEEE Transactions on Communications*, Volume 53, Issue 4, pp.549 – 554, April 2005.

31. Sungwook K. and G. Sobelman, “Parallel VLSI architectures for a class of LDPC codes”, *IEEE International Symposium on Circuits and Systems 2002*, Volume 2, pp. 93-96, 2002

32. Yeo E., P. Pakzad and B. Nikolic, “High throughput low-density parity-check decoder architectures”, *IEEE Global Telecommunications Conference 2001*, Volume 5, pp. 3019-3024, November 2001.

33. Leung W.K. and W.L. Lee, “ Efficient implementation technique of LDPC decoder”, *Electronic Letters*, Volume 37, Issue 20, pp. 1231 – 1232, September 2001.

34. Zhang T. and K. Parhi, “A 54 Mbps (3,6)-regular FPGA LDPC decoder”, *IEEE Workshop on Signal Processing Systems 2002*, pp. 127-132, October 2002.

35. Yanping L., L. Moon and G. Hwang, “New implementation for the scalable LDPC-decoders”, *IEEE 59th Vehicular Technology Conference 2004*, Volume 1, pp. 343-346, May 2004.

36. Yijun L., M. Elassal and M. Bayoumi, “Power efficient architecture for (3,6)-regular low-density parity-check code decoder”, *Proceedings of the 2004 International Symposium on Circuits and Systems*, Volume 4, pp. 81- 84, May 2004.

37. Malema G. and M. Liebelt, “Programmable low-density parity-check decoder”, *Proceedings of 2004 International Symposium on Intelligent Signal Processing and Communication Systems 2004*, pp. 801- 804, November 2004.

38. Babvey S., A. Bourgeois and J.A. Fernandez-Zepeda, “A parallel implementation of the message-passing decoder of LDPC codes using a reconfigurable optical model”, *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2005*, pp. 288 – 293, May 2005.

39. Babvey S., A. Bourgeois and J. A. Fernandez-Zepeda, “An Efficient R-Mesh Implementation of LDPC Codes Message-Passing Decoder”, *19th IEEE International Parallel and Distributed Processing Symposium 2005*, pp. 202a – 202, April 2005.

40. Kschischang F.R., B.J. Frey and H.A. Loeliger, “Factor graphs and the sum-product algorithm”, *IEEE Transactions on Information Theory*, Volume 47, Issue 2, pp. 489 – 519, February 2001.

41. Rabaey, J., Digital Integrated Circuits, Prentice-Hall, New Jersey, 1996.