

國立交通大學

電子工程學系 電子研究所碩士班

碩士論文

低密度對偶檢查碼解碼演算法之改進以及其高
速解碼器架構之設計



An Improved LDPC Decoding Algorithm and Designs of
High-Throughput Decoder Architecture

研究生：邱敏杰

指導教授：陳紹基 博士

中華民國九十五年七月

低密度對偶檢查碼解碼演算法之改進以及其高速解碼
器架構之設計

**An Improved LDPC Decoding Algorithm and Designs
of High-Throughput Decoder Architecture**

研究生：邱敏杰

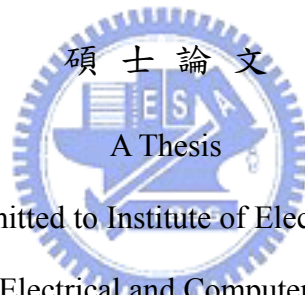
Student：Min-chieh Chiu

指導教授：陳紹基 博士

Advisor：Sau-Gee Chen

國立交通大學

電子工程學系 電子研究所碩士班



Submitted to Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Electronics Engineering

July 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年七月

低密度對偶檢查碼解碼演算法之改進以及 其高速解碼器架構之設計

學生：邱敏杰

指導教授：陳紹基 博士

國立交通大學

電子工程學系 電子研究所碩士班

摘 要

由於低密度對偶檢查碼 (LDPC) 的編碼增益接近向農 (Shannon) 極限以及解碼程序上擁有低複雜度的特性，所以在近年來受到廣泛的討論。在解碼的理論裡，尤其又以 min-sum 演算法最廣泛地被運用。因為想較於 sum-product 演算法，min-sum 演算法比較適合在硬體電路的實現。本文中，我們在 min-sum 演算法的運算式子中加了兩個參數：固定補償參數以及動態誤差參數，相較於固定補償 min-sum 演算法來說，增進了解碼器的解碼效能約 0.2dB。此外，在解碼器的設計上，我們使用部分平行 (partial-parallel) 的架構，此架構可同時處理兩筆不同之 codewords 來加快傳輸速度及資料路徑的工作效率，且共用運算單元以縮減晶片面積的大小，設計一個碼率為 1/2、長度為 576 位元、最大循環解碼次數為 10 的非規則低密度對偶檢查碼解碼器，在 0.18 μm 製程下，此解碼器之資料流為每秒 1.31bps、面積為 95 萬個邏輯閘、消耗功率為 620mW。

An Improved LDPC Decoding Algorithm and Designs of High-Throughput Decoder Architecture

Student: Min-Chieh Chiu Advisor: Dr. Sau-Gee Chen

Department of Electronics Engineering &
Institute of Electronics
National Chiao Tung University



ABSTRACT

In recent years, low-density parity-check (LDPC) codes have attracted a lot of attention due to the near Shannon limit coding gains when iteratively decoded. The min-sum decoding algorithm is extensively used because it is more suitable for VLSI implementations than sum-product algorithm. In this thesis, we propose a dynamic normalized-offset technique for min-sum algorithm and achieve a better decoding performance by about 0.2dB than normalization min-sum algorithm. Based on a partial-parallel architecture, an irregular LDPC decoder has been implemented, assuming code rate of 1/2, code length of 576 bits, and the maximum number of decoding iterations is 10. This architecture can process two different codewords concurrently to increase throughput and data path efficiency. The irregular LDPC decoder can achieve a data decoding throughput rate up to 1.31Gbps, an area of 950k gates, and a power consumption of 620mW using UMC 0.18 μm process technology.

誌 謝

本篇論文的完成承蒙指導教授 陳紹基博士兩年多來的悉心指導教誨，讓我能夠確立研究的方向，給予我多方面的協助，在此至上由衷的感激。

其次，感謝曲健全學長無私地提供協助，使我受益良多。謝謝實驗室同學譽桀、昀震、金融、文威、彥欽以及勝國，謝謝你們在課業及生活上給予我許多的幫助。還有實驗室的學弟妹們，瑞徽、飛群、思恆、至良、宜融、曉嵐，謝謝你們帶給我們許多美好的回憶。

最後，謹以此論文獻給我最深愛的家人，爺爺邱魁金、奶奶邱傳菊妹、叔公邱瑞金、父親邱正弘，感謝您們從小到大對我的包容與呵護，含辛茹苦地栽培。另外，也要感謝我的女友湘宜多年來對我的包容及生活上無微不至地照顧，讓我可以專心地完成本論文。願將此一喜悅獻給我親愛的家人及所有關心我的朋友們，謝謝您們。

Contents

中文摘要.....	I
ABSTRACT	II
ACKNOWLEDGEMENT	III
CONTENTS.....	IV
LIST OF TABLES	VI
LIST OF FIGURES	VII
Chapter 1 Introduction.....	1
1.1 Background of LDPC Codes	1
1.2 Thesis Organization	2
Chapter 2 Low-Density Parity-Check Code.....	3
2.1 Fundamental Concept of LDPC Code	3
2.2 Constructions of LDPC Codes.....	5
2.2.1 Random Code Construction.....	6
2.2.2 Deterministic Code Construction.....	8
2.3 Encoding of LDPC Codes.....	11
2.3.1 Conventional Method.....	12
2.3.2 Richardson’s Method	13
2.3.3 Quasi-Cyclic Code.....	17
2.3.3 Quasi-Cyclic Based Code	19
2.4 Conventional LDPC Decoding Algorithm.....	20
2.4.1 Bit-Flipping Algorithm	20
2.4.2 Message Passing Algorithm.....	24
Chapter 3 Modified Min-Sum Algorithms	36
3.1 Normalization Technique for Min-Sum Algorithm	36
3.2 Dynamic Normalization Technique for Min-Sum Algorithm.....	42
3.3 Proposed Dynamic Normalized-Offset-Compensation Technique for Min-Sum Algorithm.....	43
Chapter 4 Simulation Results and Analysis.....	45
4.1 Floating-Point Simulations	46
4.2 Fixed-Point Simulations.....	50
Chapter 5 Architecture Designs of LDPC Code Decoders	52

5.1 The Whole Decoder Architecture	52
5.2 Hardware Performance Comparison and Summary	65
Chapter 6 Conclusions and Future Work.....	68
6.1 Conclusions.....	68
6.2 Future Work	68
Appendix A: LDPC Codes Specification in IEEE 802.16e OFDMA.....	70
References.....	73
Autobiography.....	76



List of Tables

Table 2.1	Efficient computation step of $p_1^T = -\phi^{-1}(-ET^{-1}A + C)s^T$	15
Table 2.2	Efficient computation step of $p_2^T = -T^{-1}(As^T + Bp_1^T)$	15
Table 2.3	Summary of Richardson's encoding procedure.	16
Table 2.4	Summary of sum-product algorithm.....	33
Table 2.5	Summary of min-sum algorithm.....	34
Table 5.1	Comparison of direct and backhanded CNU architectures.....	61
Table 5.2	Area, speed, and power consumption of the CNU using min-sum algorithm and modified min-sum algorithm.....	66
Table 5.3	Comparison of LDPC decoders.....	67



List of Figures

Figure 2.1	Example of a (8, 4, 2)-regular LDPC code and its corresponding Tanner graph.	4
Figure 2.2	Example of an LDPC code matrix, where $(n, r, c) = (20, 4, 3)$	7
Figure 2.3	The parity-check matrix H of a block-LDPC code.....	9
Figure 2.4	Example of a rate-1/2 quasi-cyclic code from two circulant matrices, where $a_1(x) = 1 + x$ and $a_2(x) = 1 + x^2 + x^4$	10
Figure 2.5	The parity-check matrix in an approximate lower triangular form.....	13
Figure 2.6 (a)	Example of a rate-1/2 quasi-cyclic code. (a) Parity-check matrix with two circulants, where $a_1(x) = 1 + x$ and $a_2(x) = 1 + x^2 + x^4$	18
Figure 2.6 (b)	Example of a rate-1/2 quasi-cyclic code. (b) Corresponding generator matrix in systematic form	18
Figure 2.7	Tanner graph of the given example parity-check matrix.	25
Figure 2.8	Serial configuration for check node update function.....	29
Figure 2.9	Check node update function of sum-product algorithm	30
Figure 2.10	Check node update function of min-sum algorithm	30
Figure 2.11	Notations for iterative decoding procedure.....	31
Figure 2.12	The whole LDPC decoding procedure.....	33
Figure 3.1	Function plot of $\phi(x)$	37
Figure 3.2	The absolute difference between the normalization technique and sum-product algorithm, vs. the of normalization factor β	42
Figure 3.3	BER performance vs. threshold values K for rate 1/2 LDPC code.....	44
Figure 4.1	Decoding performance at different iteration numbers.....	46
Figure 4.2	BER performance of the rate-1/2 code at different codeword lengths, in AWGN channel, maximum iteration=10	47

Figure 4.3	Floating-point BER simulations of two decoding algorithms in AWGN channel with code length=576, code rate=1/2, maximum iteration=10	47
Figure 4.4	Floating-point BER simulations of normalized min-sum decoding algorithms in AWGN channel with code length=576, code rate=1/2, maximum iteration=10.....	48
Figure 4.5	Floating-point BER simulations under normalized-offset technique in min-sum decoding algorithms, in AWGN channel with code length=576, code rate=1/2, maximum iteration=10.....	48
Figure 4.6	Floating-point BER simulations of the dynamic normalized-offset min-sum decoding algorithm and its comparison with other algorithms, in AWGN channel with code length=576, code rate=1/2, maximum iteration=10.....	49
Figure 4.7	Floating-point BER simulations under normalized-offset compensated technique and dynamic normalization technique in min-sum algorithm.	49
Figure 4.8	Fixed-point BER simulations of three different quantization configurations of min-sum decoding algorithm, in AWGN channel, code length=576, code rate=1/2, maximum iteration=10.....	50
Figure 4.9	Floating-point vs. fixed-point BER simulations of the normalization and dynamic normalized-offset min-sum algorithm.....	51
Figure 5.1	The parity-check matrix H of block-LDPC code.....	52
Figure 5.2	The partition of parity-check matrix H	53
Figure 5.3	I/O pin of the decoder IP.....	53
Figure 5.4	The whole LDPC decoder architecture for the block-LDPC code.....	54

Figure 5.5	A simple parity-check matrix example, based on shifted identity matrix	55
Figure 5.6(a)	The sub-modules of the whole decoder	55
Figure 5.6(b)	The outputs of the module INDEX.....	56
Figure 5.7(a)	Values shuffling before sending to check node updating unit	56
Figure 5.7(b)	Values shuffling before sending to bit node updating unit	57
Figure 5.8(a)	The architecture of CNU using min-sum algorithm	58
Figure 5.8(b)	The architecture of CNU using modified min-sum algorithm.....	58
Figure 5.9	Block diagram of CS6 module.....	59
Figure 5.10(a)	Block diagram of CMP-4 module.....	60
Figure 5.10(b)	Block diagram of CMP-6 module.....	60
Figure 5.11	CNU architecture using min-sum algorithm.....	61
Figure 5.12	The architecture of the bit node updating unit with 4 inputs	62
Figure 5.13(a)	The architecture of RE-4B based MMU	63
Figure 5.13(b)	The timing diagram of the message memory units.....	64
Figure 5.14	The message passing snapshots between MMU0 and MMU1	65
Figure A.1	Base matrix of the rate 1/2 code	71
Figure A.2	Base matrix of the rate 2/3, type A code	71
Figure A.3	Base matrix of the rate 2/3, type B code	72
Figure A.4	Base matrix of the rate 3/4, type A code	72
Figure A.5	Base matrix of the rate 3/4, type B code	72
Figure A.6	Base matrix of the rate 5/6 code	72

Chapter 1

Introduction

1.1 Background of LDPC Codes

Low-density parity check (LDPC) code, a linear block code defined by a very sparse parity-check matrix, was first introduced by Gallager [1]. Due to the difficulty in circuit implementation, LDPC codes have been ignored for about forty years except for the study of codes defined on graphs by Tanner [3]. The rediscovery of LDPC code was done by MacKay [10]. It has engaged much research interest ever since, because the sparse property of parity-check matrix makes the decoding algorithm simple and practical with good communication throughput rates. LDPC code is currently widely considered a serious competitor to the turbo codes. The main advantages of LDPC codes over turbo codes are that LDPC decoders are known to require an order of magnitude less arithmetic computations, and the decoding algorithms for LDPC codes are parallelizable and can potentially be accomplished at significantly greater speeds. The main decoding algorithm of LDPC codes is sum-product algorithm [10]. However, sum-product algorithm is prone to quantization errors while realized in hardware. Thus, several reduced-complexity algorithms with different levels of performance degradation have been proposed [14]. The thesis proposed a dynamically normalized-offset technique to improve the decoding performance and reduce the decoding complexity.

The implementation of LDPC codes decoders can be classified into fully parallel decoders, and partial-parallel decoders. The fully parallel decoders directly map the corresponding bipartite graph [17] into hardware and all the processing units are hard-wired according to the connectivity of the graph. Thus they can achieve very high decoding speed but have a high hardware cost. Another approach is to have a partial-parallel decoder [19], in which the functional units are reused in order to decrease the chip-area. Moreover, this architecture can process two different codewords concurrently to has moderate throughput. The other aim of this thesis is to improve the partial-parallel architecture and save chip area, with little degradation of the throughput.

1.2 Thesis Organization



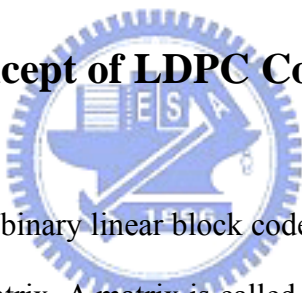
This thesis is organized as follows. In chapter 2, basic concept of the LDPC codes: the code construction, encoding concept and various decoding algorithms will be introduced. Chapter 3 will first introduce the modified min-sum algorithm which uses the normalized technique. Then we propose a new dynamic normalized-offset technique for min-sum decoding algorithm. In chapter 4, the simulation results for the LDPC code which is discussed in chapter 2 and chapter 3 will be shown. In chapter 5, hardware architecture of the LDPC decoder will be discussed here. In the end of this thesis, brief conclusions and future work will be presented in chapter 6.

Chapter 2

Low-Density Parity-Check Codes

In this chapter, an introduction to low-density parity-check code will be given, including the fundamental concepts of LDPC code, code construction, encoding mechanism and decoding algorithm.

2.1 Fundamental Concept of LDPC Code



A binary LDPC code is a binary linear block code that can be defined by a sparse binary $m \times n$ parity-check matrix. A matrix is called a sparse matrix because there is only a small fraction of its entries are ones. In other words, most part of the parity-check matrix are zeros and the else part of that are ones.

For any $m \times n$ parity-check matrix H , it defines a (n, k, r, c) -regular LDPC code if every column vector of H has the same weight c and every row vector of H has the same weight r . Here the weight of a vector is the number of ones in the vector. $k = n - m$. By counting the ones in H , it follows that $n \times c = k \times r$. Hence if $m < n$, then $c < r$. Suppose the parity-check matrix has full rank, the code rate of H is $(r - c)/r = 1 - c/r$. If all the column-weights or the row-weights are not the same, an LDPC code is said to be irregular.

As suggested by Tanner [7], an LDPC code can be represented by a bipartite graph. An LDPC code corresponds to a unique bipartite graph and a bipartite graph also corresponds to a unique LDPC code. In a bipartite graph, one type of nodes, called the variable (bit) nodes, correspond to the symbols in a codeword. The other type of nodes, called the check nodes, correspond to the set of parity check equations. If the parity-check matrix H is an $m \times n$ matrix, it has m check nodes and n variable nodes. A variable node v_i is connected to a check node c_j by an edge, denoted as (v_i, c_j) , if and only if the entry $h_{i,j}$ of H is one. A cycle in a graph of nodes and edges is defined as a sequence of connected edges which starts from a node and ends at the same node, and satisfies the condition that no node (except the initial and final node) appears more than one time. The number of edges on a cycle is called the length of the cycle. The length of the shortest cycle in a Tanner graph is called the girth of the graph.

Regular LDPC codes are those where all nodes of the same type have the same degree. The degree of a node is the number of edges connected to that node. For example, Figure 2.1 shows a (8, 4, 4, 2)-regular LDPC code and its corresponding

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

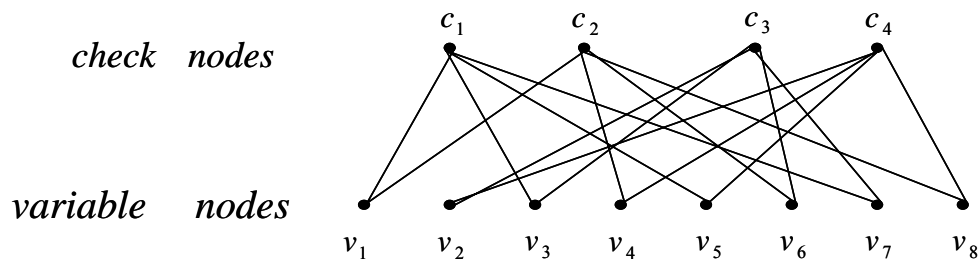


Figure 2.1 (8, 4, 4, 2)-regular LDPC code and its corresponding Tanner graph.

Tanner graph. In this example, there are 8 variable nodes (v_i), 4 check nodes (c_i), the row weight is 4 and the column weight is 2. The edges (c_1, v_3) , (v_3, c_3) , (c_3, v_7) , and (v_7, c_1) depict a cycle in the Tanner graph. Since this turns out to be the shortest cycle, the girth of the Tanner graph is 4. Irregular LDPC codes were introduced in [8] and [9].

2.2 Constructions of LDPC Codes

This section is going to discuss the parity-check matrix H of LDPC code. The design of H is the moment when the asymptotical constraints (the parameters of the class you designed, like the degree distribution, the rate) have to meet the practical constraints (finite dimension, girths).

Here, we describe some recipes which take some practical constraints into account. Two techniques exist in the literature: random and deterministic ones. The design compromise is that for increasing the girth, the sparseness has to be decreased, so is the code performance decreased due to a low minimum distance. On the contrary, for high minimum distance, the sparseness has to be increased yielding the creation of low-length girth, due to the fact that H dimensions are finite, and thus, yielding a poor convergence of sum-product algorithm.

2.2.1 Random Code Construction

The first constructions of LDPC codes are random ones. They were proposed by Gallager [1] and MacKay [10]. The parity check matrix in Gallager's method is a concatenation and/or superposition of sub-matrices; these sub-matrices are created by performing some permutations on a particular (random or not) sub-matrix which usually has a column weight of 1. The parity check matrix in MacKay's method is computer-generated. These two methods are introduced below.

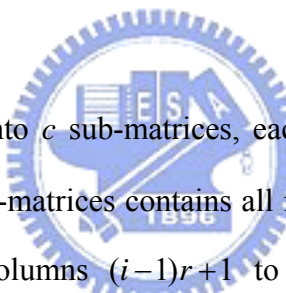
Gallager's method [1]



Define an (n, r, c) parity check-matrix as a matrix of n columns that has c ones in each column, r ones in each row, and zeros elsewhere. Following this definition, an (n, r, c) parity-check matrix has nc/r rows and thus a rate of $\text{coderate} \geq 1 - c/r$. In order to construct an ensemble of (n, r, c) matrices, consider first the special (n, r, c) matrix in Figure 2.2, where n, r and c are 20, 4 and 3, respectively.

1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1
1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0
0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1

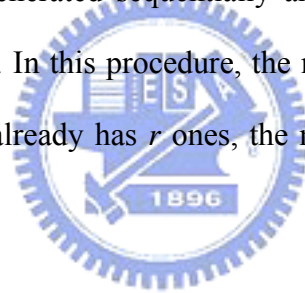
Figure 2.2 Example of an LDPC code matrix, where $(n, r, c)=(20,4,3)$



This matrix is divided into c sub-matrices, each containing a single 1 in each column. The first of these sub-matrices contains all its 1's in descending order where the i^{th} row contains 1's in columns $(i-1)r+1$ to ir . The other sub-matrices are merely column permutations of the first. We define the ensemble of (n, r, c) codes as the ensemble resulting from random permutations of the columns of each of the bottom $(c-1)$ sub-matrices of a matrix such as in Figure 2.2 with equal probability assigned to each permutation. This definition is somewhat arbitrary and is made for mathematical convenience. In fact such an ensemble does not include all (n, r, c) codes as just defined. Also, at least $(c-1)$ rows in each matrix of the ensemble are linearly dependent. This simply means that the codes have a slightly higher information rate than the matrix indicates.

MacKay's method [10]

A computer-generated code was introduced by MacKay [10]. The parity-check matrix is randomly generated. First, parameters n , m , r , and c are chosen to conform an (n, m, r, c) -regular LDPC code where n , r and c are the same as in Gallager's code and m is the number of the parity-check equations in H . Then, 1's are randomly generated into c different positions of the first column. The second column is generated in the same way, but checks are made to insure that no two columns have a 1 in the same position more than twice in order to avoid 4-cycle in the Tanner graph. If there is a 4-cycle in the Tanner graph, the decoding performance will be reduced by about 0.5dB. Avoidance of 4-cycles in a parity-check matrix is therefore required. The next few columns are generated sequentially and checks for 4-cycles must be performed in each generation. In this procedure, the number of 1's in each row must be recorded, and if any row already has r ones, the next-column generation will not select that row.

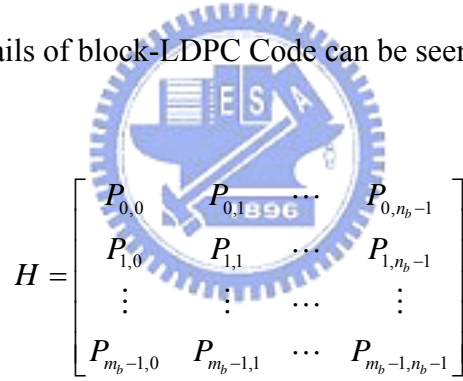


2.2.2 Deterministic Code Construction

A parity-check matrix H by random construction is sparse, but its corresponding generator matrix is not. This property will increase encoding complexity. To circumvent this problem, deterministic code construction schemes have been proposed. It can lead to low encoding complexities. Some forms of the deterministic code construction include block-LDPC code, quasi-cyclic code [5], and quasi-cyclic based code [21]. They are introduced below.

Block-LDPC Code

The parity check matrix H based on block-LDPC code is composed by several sub-matrices. The size of H is m -by- n . The sub-matrices are shifted identity matrices or zero matrices. The matrix form of H is shown in Figure 2.3. Sub-matrix $P_{i,j}$ is one of a set of z -by- z permutation matrices or a z -by- z zero matrix. Matrix H is expanded from a binary base matrix H_b of size m_b -by- n_b , where $m = z \cdot m_b$, $n = z \cdot n_b$, and z is an integer ≥ 1 . The base matrix is expanded by replacing each 1 in the base matrix with a z -by- z permutation matrix, and each 0 with a z -by- z zero matrix. The used permutations are circular right shifts, and the set of permutation matrices contains the z -by- z identity matrix and circularly right-shifted versions of the identity matrix. The details of block-LDPC Code can be seen in Appendix A.



$$H = \begin{bmatrix} P_{0,0} & P_{0,1} & \cdots & P_{0,n_b-1} \\ P_{1,0} & P_{1,1} & \cdots & P_{1,n_b-1} \\ \vdots & \vdots & \cdots & \vdots \\ P_{m_b-1,0} & P_{m_b-1,1} & \cdots & P_{m_b-1,n_b-1} \end{bmatrix}$$

Figure 2.3 The parity-check matrix H of a block-LDPC code

Quasi-Cyclic Code [5]

A code is quasi-cyclic if, for any cyclic shift of a codeword by l places, the resulting word is also a codeword. A cyclic code is a quasi-cyclic code with $l = 1$. Consider the binary quasi-cyclic codes described by a parity-check matrix

$$H = [A_1, A_2, \dots, A_l] \quad (2.10)$$

where A_1, A_2, \dots, A_l are binary $v \times v$ circulant matrices. The algebra of $(v \times v)$ binary circulant matrices is isomorphic to the algebra of polynomials modulo $x^v - 1$

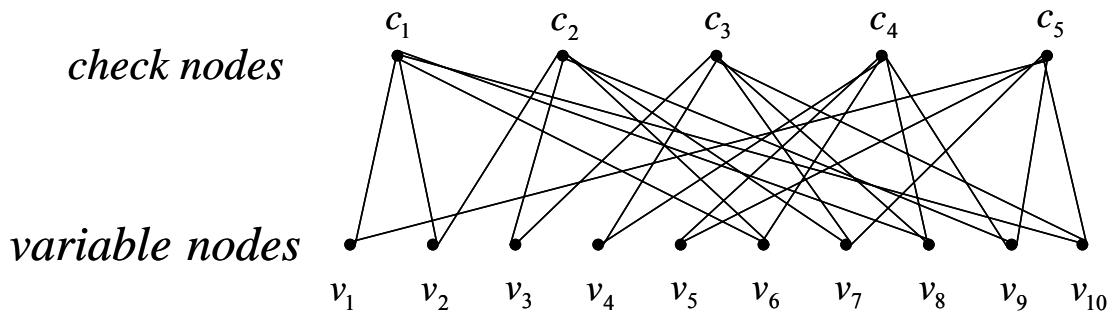
over GF(2). A circulant matrix A is completely characterized by the polynomial

$$a(x) = a_0 + a_1x + a_2x^2 + \dots + a_{v-1}x^{v-1} \quad (2.11)$$

where the coefficients are from the first row of A , and a code C with parity-check matrix of the form (2.10) can be completely characterized by the polynomials $a_1(x), a_2(x), \dots, a_l(x)$. Figure 2.4(a) shows an example of a rate-1/2 quasi-cyclic code, where $a_1(x) = 1 + x$ and $a_2(x) = 1 + x^2 + x^4$. Figure 2.4(b) shows the corresponding Tanner graph representation. For this example, we can see the edges $(c_1, v_6), (v_6, c_4), (c_4, v_8), (v_8, c_1)$ depict a 4-cycle in this graph which is to be avoided for performance consideration.

$$H = \left[\begin{array}{cccc|ccccc} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{array} \right]$$

(a) A parity-check matrix with two circulant matrices



(b) Tanner graph representation

Figure 2.4 Example of a rate-1/2 quasi-cyclic code from two circulant matrices, where

$$a_1(x) = 1 + x \quad \text{and} \quad a_2(x) = 1 + x^2 + x^4$$

Quasi-Cyclic Based Code [21]

The code is constructed with a base of quasi-cyclic code. The parity check matrix is in the following form.

$$H = \begin{bmatrix} A_1 & A_2 & \dots & A_{l-1} & 0 \\ B_1 & B_2 & \dots & B_{l-1} & B_l \end{bmatrix} \quad (2.12)$$

where $A_1, A_2, \dots, A_{l-1}, B_1, B_2, \dots,$ and B_l are all $v \times v$ circulant matrices. The code length is vl and the code rate is $(1 - \frac{2}{l})$. We can use the difference families [21] to determine the polynomials of each of the circulant matrix $a_i(x)$ and $b_j(x)$, where $i \in \{1, 2, \dots, l-1\}$ and $j \in \{1, 2, \dots, l\}$, just as the quasi-cyclic code. In order to avoid any 4-cycles in the new structure of the parity-check matrix, we provide a new difference family to solve this problem. First, construct two $(v, \gamma, 1)$ difference families Family A and Family B and combine the two families to form a new difference Family C, subject to the following two constraints.

Constraint 1: The differences $[(a_{i,x} - a_{i,y}) \bmod v]$ and $[(b_{i,x} - b_{i,y}) \bmod v]$, where $i = 1, 2, \dots, l-1; x, y = 1, 2, \dots, \gamma, x \neq y$, give each element, can not be the same.

Constraint 2: The differences $[(a_{i,x} - a_{j,y}) \bmod v]$ and $[(b_{i,x} - b_{j,y}) \bmod v]$, where $i, j = 1, 2, \dots, l-1, i \neq j; x, y = 1, 2, \dots, \gamma$, give each element, can not be the same.

2.3 Encoding of LDPC Codes

Since LDPC code is a linear block code, it can be encoded by conventional methods. However, conventional methods require encoding complexities proportional to the quadratic of the code length. The high encoding cost of LDPC code becomes a major drawback when compared to the turbo codes which have linear time encoding

complexity. In this section, we will introduce some improved methods.

2.3.1 Conventional Method

Let $u = [u_0, u_1, u_2, \dots, u_{k-1}]$ be a row vector of message bits with length k and $c = [c_0, c_1, c_2, \dots, c_{n-1}]$ be a codeword with length n . Let G with dimension $k \times n$ be the generating matrix of this code, and

$$c = uG . \quad (2.12)$$

If H is the parity-check matrix of this code with dimension $m \times n$, where $m = n - k$, then

$$\begin{aligned} Hc^T = 0^T &\Rightarrow cH^T = 0 \\ &\Rightarrow uGH^T = 0 \\ &\Rightarrow GH^T = 0 \end{aligned} \quad (2.13)$$

Suppose a sparse parity-check matrix H with full rank is constructed. Gaussian elimination and column reordering can be used to derive an equivalent parity-check matrix in the systematic form $H_{systematic} = [P|I_r]$. Thus equation (2.13) can be solved to get the generating matrix in a systematic form as

$$G_{systematic} = [I_k | P^T] . \quad (2.14)$$

Finally, the generating matrix G can be obtained by doing the reverse column reordering to the $G_{systematic}$.

Triangularized parity-check matrix form [4]

In [4], it suggests to force the parity-check matrix to be in a lower triangular form. Under this restriction, it guarantees a linear time encoding complexity, but, in

general, it also results in some loss of performance.

2.3.2 Richardson's Method [3]

Richardson's method is the most extensively used among LDPC encoding algorithms. Figure 2.5 shows how to bring a parity-check matrix into an approximate lower triangular form using row and column permutations. Note that since this

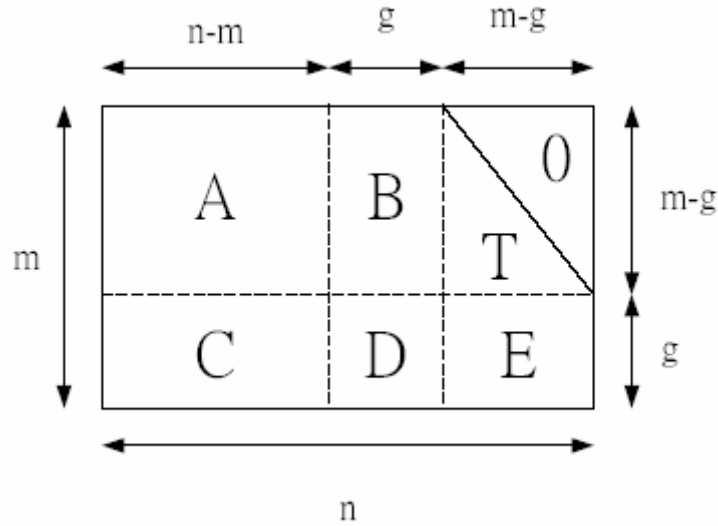


Figure 2.5 The parity-check matrix in an approximate lower triangular form transformation was accomplished solely by permutations, the parity check matrix H is still sparse. This method is to cut the parity check matrix H into 6 sub-matrices: A , B , T , C , D , E . Especially, the sub-matrix T is in lower triangular form.

More precisely, it is assumed that the matrix is written in the form

$$H = \begin{pmatrix} A & B & T \\ C & D & E \end{pmatrix} \quad (2.15)$$

where A is $(m-g) \times (n-m)$, B is $(m-g) \times g$, T is $(m-g) \times (m-g)$, C is $g \times (n-m)$, D is $g \times g$, and E is $g \times (m-g)$. Further, all these matrices are sparse and T is lower triangular with ones along the diagonal. Let $x = (s, p_1, p_2)$ denote the

codeword of this parity-check matrix where s is the message bits with length $(m-n)$, p_1 combined with p_2 are the parity bits, and p_1 and p_2 have length g , and $(m-g)$, respectively. Multiplying the matrix in equation (2.16) on both sides of the constraint equation $H \cdot x^T = 0$

$$\begin{pmatrix} I & 0 \\ -ET^{-1} & I \end{pmatrix} \quad (2.16)$$

can result in

$$\begin{pmatrix} A & B & T \\ -ET^{-1}A + C & -ET^{-1}B + D & 0 \end{pmatrix} \cdot x^T = 0. \quad (2.17)$$

Expanding equation (2.17), one can get equations (2.18) and (2.19)

$$As^T + Bp_1^T + Tp_2^T = 0 \quad (2.18)$$

$$(-ET^{-1}A + C)s^T + (-ET^{-1}B + D)p_1^T = 0. \quad (2.19)$$

Define $\phi = -ET^{-1}B + D$ and assume for the moment that ϕ is nonsingular. Then from equation (2.19) we conclude that

$$p_1^T = -\phi^{-1}(-ET^{-1}A + C)s^T. \quad (2.20)$$

Hence, once the $g \times (n-m)$ matrix $-\phi^{-1}(-ET^{-1}A + C)s^T$ has been pre-computed, the determination of p_1 can be accomplished with a time complexity of $O(g \times (n-m))$ simply by performing a multiplication with this (generally dense) matrix. This complexity can be further reduced as shown in Table 2.1. Rather than pre-computing $-\phi^{-1}(-ET^{-1}A + C)s^T$ and then multiplying with s^T , p_1 can be determined by breaking the computation into several smaller steps, each of which is computationally efficient. To this end, we first determine As^T , which has complexity of $O(n)$, because A is sparse. Next, we multiply the result by T^{-1} . Since $T^{-1}[As^T] = y^T$ is equivalent to the system $[As^T] = Ty^T$, this can also be accomplished in $O(n)$ time by back-substitution method, because T is lower triangular and sparse. The remaining steps are fairly straightforward. It follows that

the overall complexity of determining p_1 is $O(n + g^2)$. In a similar manner, noting from equation (2.18) that $p_2^T = -T^{-1}(As^T + Bp_1^T)$, we can determine p_2 in time complexity of $O(n)$, as shown step by step in Table 2.2.

A summary of this efficient encoding procedure is given in Table 2.3. It contains two steps, the preprocessing step and the actual encoding step. In the preprocessing step, we first perform row and column permutations to bring the parity-check matrix into the approximate lower triangular form with as small a gap g as possible. In actual encoding, it contains the steps listed in Table 2.1 and 2.2. The overall encoding complexity is $O(n + g^2)$, where g is the gap of the approximate triangularization.

Table 2.1 Efficient computation steps of $p_1^T = -\phi^{-1}(-ET^{-1}A + C)s^T$

Operation	Comment	Complexity
As^T	Multiplication by sparse matrix	$O(n)$
$T^{-1}[As^T]$	$T^{-1}[As^T] = y^T \Leftrightarrow [As^T] = Ty^T$	$O(n)$
$-E[T^{-1}As^T]$	Multiplication by sparse matrix	$O(n)$
Cs^T	Multiplication by sparse matrix	$O(n)$
$[-ET^{-1}As^T] + [Cs^T]$	Addition	$O(n)$
$-\phi^{-1}[-ET^{-1}As^T + Cs^T]$	Multiplication by dense $g \times g$ matrix	$O(g^2)$

Table 2.2 Efficient computation steps of $p_2^T = -T^{-1}(As^T + Bp_1^T)$

Operation	Comment	Complexity
As^T	Multiplication by sparse matrix	$O(n)$
Bp_1^T	Multiplication by sparse matrix	$O(n)$
$[As^T] + [Bp_1^T]$	Addition	$O(n)$

$-T^{-1}[As^T + Bp_1^T]$	$-T^{-1}[As^T + Bp_1^T] = y^T \Leftrightarrow -[As^T + Bp_1^T] = Ty^T$	$O(n)$
--------------------------	--	--------

Table 2.3 Summary of Richardson's encoding procedure

<p>Preprocessing: Input: Non-singular parity-check matrix H. Output: An equivalent parity-check matrix of the form $\begin{pmatrix} A & B & T \\ C & D & E \end{pmatrix}$ such that $-ET^{-1}B + D$ is non-singular.</p> <ol style="list-style-type: none"> [Triangularization] Perform row and column permutations to bring the parity-check matrix H into the approximate lower triangular form $H = \begin{pmatrix} A & B & T \\ C & D & E \end{pmatrix}$ with as small a gap g as possible. [Check] Check that $-ET^{-1}B + D$ is non-singular, performing further column permutations if necessary to ensure this property. $\begin{pmatrix} I & 0 \\ ET^{-1} & I \end{pmatrix} \begin{pmatrix} A & B & T \\ C & D & E \end{pmatrix} = \begin{pmatrix} A & B & T \\ -ET^{-1}A + C & -ET^{-1}B + D & 0 \end{pmatrix}$ <p>Encoding: Input: Parity-check matrix of the form $\begin{pmatrix} A & B & T \\ C & D & E \end{pmatrix}$ such that $-ET^{-1}B + D$ is non-singular and a vector s denote the message bits has length $(m - n)$. Output: The vector $x = (s, p_1, p_2)$ where p_1 has length g and p_2 has length $(m - g)$, such that $Hx^T = 0^T$.</p> <ol style="list-style-type: none"> Determine p_1 as shown in Table 2.1. Determine p_2 as shown in Table 2.2.

2.3.3 Quasi-Cyclic Code [5]

As reviewed in section 2.2, the quasi-cyclic code can be described by a parity-check matrix $H = [A_1, A_2, \dots, A_l]$ and each of a circulant matrix A_j is completely formed by the polynomial $a(x) = a_0 + a_1x + \dots + a_{v-1}x^{v-1}$ with coefficients from its first row. A code C with parity-check matrix H can be completely characterized by the polynomials $a_1(x), a_2(x), \dots,$ and $a_l(x)$. As for the encoding, if one of the circulant matrices is invertible (say A_l) the generator matrix for the code can be constructed in the following systematic form.

$$G = \begin{bmatrix} & (A_l^{-1} A_1)^T & & \\ & (A_l^{-1} A_2)^T & & \\ I_{v(l-1)} & & \dots & \\ & & & (A_l^{-1} A_{l-1})^T \end{bmatrix} \quad (2.21)$$

It results in a quasi-cyclic code of length vl and dimension $v(l-1)$. The encoding process can be achieved with linear complexity using a $v(l-1)$ -stage shift register.

Regarding the algebraic computation, the polynomial transpose is defined as

$$a(x)^T = \sum_{i=0}^{n-1} a_i x^{n-i}, \quad x^n = 1. \quad (2.22)$$

For a binary $[n, k]$ code, length $n = vl$ and dimension $k = v(l-1)$, the k -bit message

$[i_0, i_1, \dots, i_{k-1}]$ is described by the polynomial $i(x) = i_0 + i_1x + \dots + i_{k-1}x^{k-1}$ and the

codeword for this message is $c(x) = [i(x), x^k p(x)]$, where $p(x)$ is given by

$$p(x) = \sum_{j=1}^{l-1} i_j(x) * (a_l^{-1}(x) * a_j(x))^T, \quad (2.23)$$

where $i_j(x)$ is the polynomial representation of the information bits $i_{v(j-1)}$ to i_{vj-1} ,

$$i_j(x) = i_{v(j-1)} + i_{v(j-1)+1}x + \dots + i_{vj-1}x^{v-1} \quad (2.24)$$

and polynomial multiplication $(*)$ is mod $x^v - 1$.

As an example, consider a rate-1/2 quasi-cyclic code with $\nu = 5$, $l = 2$, the first circulant is described by $a_1(x) = 1 + x$ and the second circulant is described by $a_2(x) = 1 + x^2 + x^4$, which is invertible and

$$a_2^{-1}(x) = x + x^2 + x^4. \quad (2.25)$$

The generator matrix contains a 5×5 identity matrix and the 5×5 matrix described by the polynomial

$$(a_2^{-1}(x) * a_1(x))^T = (1 + x^2)^T = 1 + x^3. \quad (2.26)$$

Figure 2.6 shows the example parity-check matrix and the corresponding generator matrix.

$$H = \left[\begin{array}{ccccc|ccccc} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{array} \right]$$

(a) A parity-check matrix with two circulants

$$G = \left[\begin{array}{ccccc|ccccc} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{array} \right]$$

(b) The corresponding generator matrix in systematic form

Figure 2.6 Example of a rate-1/2 quasi-cyclic code. (a) Parity-check matrix with two circulants, where $a_1(x) = 1 + x$ and $a_2(x) = 1 + x^2 + x^4$. (b) Corresponding generator matrix in systematic form.

Quasi-Cyclic Based Code [21]

As reviewed in section 2.2, the quasi-cyclic based code can be described by a

parity-check matrix $H = \begin{bmatrix} A_1 & A_2 & \dots & A_{l-1} & 0 \\ B_1 & B_2 & \dots & B_{l-1} & B_l \end{bmatrix}$, where

$A_1, A_2, \dots, A_{l-1}, B_1, B_2, \dots,$ and B_l are all $v \times v$ circulant matrices. Regarding the encoding for the quasi-cyclic based structure, suppose that two of the circulant matrices A_{l-1} and B_l are invertible, we can derive two generator matrices in the following systematic forms

$$G_{1 \text{ systematic}} = \begin{bmatrix} (A_{l-1}^{-1} A_1)^T \\ (A_{l-1}^{-1} A_2)^T \\ \dots \\ (A_{l-1}^{-1} A_{l-2})^T \end{bmatrix} = [I_{v(l-2)} G_1] \quad (2.27)$$

and

$$G_{2 \text{ systematic}} = \begin{bmatrix} (B_l^{-1} B_1)^T \\ (B_l^{-1} B_2)^T \\ \dots \\ (B_l^{-1} B_{l-1})^T \end{bmatrix} = [I_{v(l-1)} G_2]. \quad (2.28)$$

Let $c = [d, p_1, p_2]$ denote the codeword of the proposed parity-check matrix where d is the message bits with length $v(l-2)$, and p_1 combined with p_2 are the parity bits, each having the same length v . The encoding procedure is partitioned into two steps.

Encoding Step 1: We can use the generator matrix G_1 to get the parity bits p_1 . That is

$$p_1 = d \times G_1. \quad (2.29)$$

Then, combine the parity bits p_1 with the message bits d to form an intermediate codeword c' where $c' = [d, p_1]$.

Encoding Step 2: The last parity bits p_2 can be derived from the generator matrix G_2 and the intermediate codeword c' . That is

$$p_2 = c' \times G_2. \quad (2.30)$$

2.4 Conventional LDPC Code Decoding Algorithm

There are several decoding algorithms for LDPC codes. The LDPC decoding algorithms can be summarized as: bit-flipping algorithm [20], and message passing algorithm [11]. In the following, we will make an introduction of the decoding algorithms.



2.4.1 Bit-Flipping Algorithm [20]

The idea for decoding is the fact that in case of low-density parity-check matrices the syndrome weight increases with the number of errors in average until errors weights are much larger than half the minimum distance. Therefore, the idea is to flip one bit in each iteration, and the bit to be flipped is chosen such that the syndrome weight decreases. It should be noted that not only rows of the parity-check matrix can be used for decoding, but in principle all vectors of the dual code with minimum (or small) weight. In the following, we will introduce two of the bit-flipping algorithms [20].

Notation and Basic Definitions

The idea behind this algorithm is to “flip” the least number of bits until the parity check equation $H \cdot x^T = 0$ is satisfied. Suppose a binary (n, k) LDPC code is used for error control over a binary-input additive white Gaussian noise (BIAWGN) channel with zero mean and power spectral density σ^2 . The letter n is the code length and k is the message length. Assume binary phase-shift-keying (BPSK) signaling with unit energy is adopted. A codeword $c = (c_0, c_1, \dots, c_{n-1}) \in \{GF(2)\}^n$ is mapped into bipolar sequence $x = (x_0, x_1, \dots, x_{n-1})$ before its transmission, where $x_i = 2 \cdot (c_i - 1)$, $0 \leq i \leq n-1$. Let $y = (y_0, y_1, \dots, y_{n-1})$ be the soft-decision received sequence at the output of the receiver matched filter. For $0 \leq i \leq n-1$, $y_i = x_i + n_i$, where n_i is a Gaussian random variable with zero mean and variance σ^2 . An initial binary hard decision of the received sequence, $z^{(0)} = (z_0^{(0)}, z_1^{(0)}, \dots, z_{n-1}^{(0)})$, is determined as follows

$$z_i^{(0)} = \begin{cases} 1, & y_i \geq 0 \\ 0, & y_i < 0 \end{cases} \quad (2.31)$$

For any tentative binary hard decision z made at the end of each decoding iteration, we can compute the syndrome vector as $s = H \cdot z^T$. One can define the log-likelihood ratio (LLR) for each channel output y_i , $0 \leq i \leq n-1$:

$$L_i = \ln \frac{p(c_i = 1 | y_i)}{p(c_i = 0 | y_i)} \quad (2.32)$$

The absolute value of L_i , $|L_i|$, is called the reliability of the initial decision $z_i^{(0)}$.

For any binary vector $v = (v_0, v_1, \dots, v_{n-1})$, let $wt(\mathbf{v})$ be the Hamming weight of \mathbf{v} . Let u_i be the n dimensional unit vector, i.e., a vector with “1” at the i -th position and “0” everywhere else.

Algorithm I

Step (1) Initialization: Set iteration counter $k = 0$. Calculate $z^{(0)}$ and $S^{(0)} = wt(H \cdot z^{(0)T})$.

Step (2) If $S^{(k)} = 0$, then go to Step (8).

Step (3) $k \leftarrow k+1$. If $k > k_{\max}$, where k_{\max} is the maximum number of iterations, go to Step (9).

Step (4) For each $i = 0, 1, \dots, n-1$, calculate $S_i^{(k)} = wt[H \cdot (z^{(k-1)} + u_i)^T]$

Step (5) Find $j^{(k)} \in \{0, 1, \dots, n-1\}$ with $j^{(k)} = \arg(\min_{0 \leq i < n} S_i^{(k)})$.

Step (6) If $j^{(k)} = j^{(k-1)}$, then go to Step (9).

Step (7) Calculate $z^{(k)} = z^{(k-1)} + u_{j^{(k)}}$ and $S^{(k)} = wt(H \cdot z^{(k)T})$. Go to Step (2).

Step (8) Stop the decoding and return $z^{(k)}$.

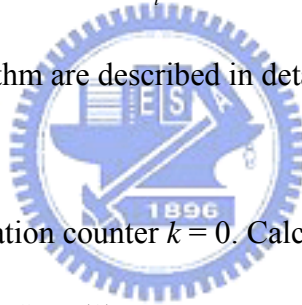
Step (9) Declare a decoding failure and return $z^{(k-1)}$.

So the algorithm flips only one bit at each iteration and the bit to be flipped is chosen according to the fact that, in average, the weight of the syndrome increases with the weight of the error. Note that in some cases, the decoder can choose a wrong position j , and thus introduce a new error. But there is still a high likelihood that this new error will be corrected in some later step of the algorithm.

Algorithm II

Algorithm I can be modified, with almost no increase in complexity, to achieve better error performance, by including some kind of reliability information (or measure) of the received symbols. Many algorithms for decoding linear block codes

based on this reliability measure have been devised. Consider the received soft-decision sequence $y = (y_0, y_1, \dots, y_{n-1})$. For the AWGN channel, a simple measure of the reliability, L_i , of a received symbol y_i is its magnitude, $|y_i|$. The larger the magnitude $|y_i|$ is, the larger the reliability of the hard-decision digit z_i is. If the reliability of a received symbol y_i is high, we want to prevent the decoding algorithm from flipping this symbol, because the probability of this symbol being erroneous is less than the probability of this symbol being correct. This can be achieved by appropriately increasing the values S_i in the decoding algorithm. The solution is to increase the values of S_i by the following term: $|L_i|$. The larger value of $|L_i|$ implies that the hard-decision z_i is more reliable. The steps of the soft version of the decoding algorithm are described in detail below:



Step (1) Initialization: Set iteration counter $k = 0$. Calculate $z^{(0)}$ and $S^{(0)} = wt(H \cdot z^{(0)T})$.

Step (2) If $S^{(k)} = 0$, then go to Step (8).

Step (3) $k \leftarrow k+1$. If $k > k_{\max}$, go to Step (9).

Step (4) For each $i = 0, 1, \dots, n-1$, calculate $S_i^{(k)} = wt[H \cdot (z^{(k-1)} + u_i)^T] + |L_i|$

Step (5) Find $j^{(k)} \in \{0, 1, \dots, n-1\}$ with $j^{(k)} = \arg(\min_{0 \leq i < n} S_i^{(k)})$.

Step (6) If $j^{(k)} = j^{(k-1)}$, then go to Step (9).

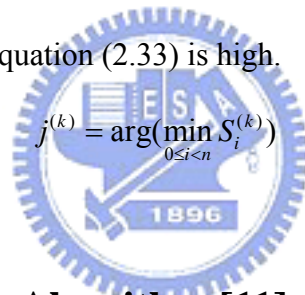
Step (7) Calculate $z^{(k)} = z^{(k-1)} + u_{j^{(k)}}$ and $S^{(k)} = wt(H \cdot z^{(k)T})$. Go to Step (2).

Step (8) Stop the decoding and return $z^{(k)}$.

Step (9) Declare a decoding failure and return $z^{(k-1)}$.

It is important to point out that, in both algorithms, though the maximum number of iteration is specified, the algorithms have an inherent stopping criterion. The decoding process stops either when a valid codeword is obtained (Step 2) or when the minimum syndrome weight at the k th iteration and the minimum syndrome weight at the $(k-1)$ th iteration are found in the same position (Step 6).

The bit-flipping algorithm just corrects at most one error bit in one iteration. The codeword length of LDPC code is usually hundreds (or thousands) of bits. When the channel SNR (signal-to-noise ratio) is low, the decoding iteration number of the bit-flipping algorithm needs to be high to correct the erroneous bits. This will lower the throughput of the decoder. And according to Step (5), equation (2.33) is to find the minimal value of the n numbers. The value of n (codeword length) is usually large. The hardware complexity of equation (2.33) is high.



$$j^{(k)} = \arg(\min_{0 \leq i < n} S_i^{(k)}) \quad (2.33)$$

2.4.2 Message Passing Algorithm [11]

Since the bit-flipping algorithm is hard to be implemented in hardware, the message passing algorithm is extensively used for LDPC decoding. The message passing algorithm is an iterative decoding process. Messages between variable nodes and check nodes are exchanged back and forth. The decoder expects that error will be corrected progressively by using this iterative message-passing algorithm. At present, there are two types of iterative decoding algorithms applied to LDPC codes in general.

- Sum-product algorithm, also known as belief propagation algorithm.
- Min-sum algorithm

Both of sum-product algorithm and min-sum algorithm are message passing algorithms. In the following, we will discuss these two algorithms in detail. First, we explain the decoding procedure in Tanner graph below.

Decoding Procedure in Tanner Graph Form

Now we make a description of the message passing algorithm using Tanner graph form. Here is a simple example of irregular LDPC code. The parity-check matrix is shown below.

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \begin{matrix} S_1 \\ S_2 \end{matrix}$$

x_1 x_2 x_3 x_4

Tanner graph of this parity-check matrix is shown in Figure 2.7.

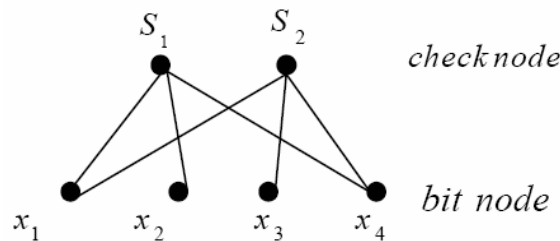
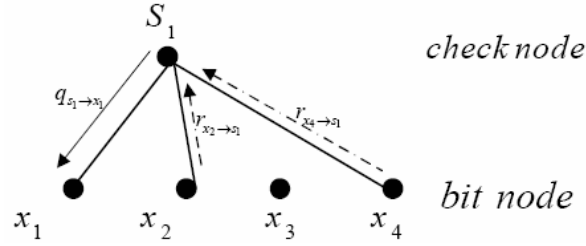


Figure 2.7 Tanner graph of the given example parity-check matrix

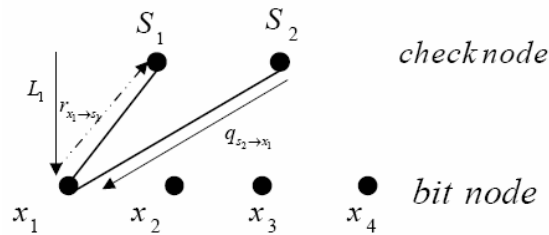
Assume every line in the Tanner graph has two information messages. One is expressed in a solid line and the other is expressed in a dotted line. We use the messages to decode the received signal. For convenience of explanation, we take one part of Tanner graph which is shown below.



The solid line and the dotted line are represented by $q_{s \rightarrow x}$ and $r_{x \rightarrow s}$, respectively. In this example, we can get $q_{s_1 \rightarrow x_1}$ by $r_{x_2 \rightarrow s_1}$ and $r_{x_4 \rightarrow s_1}$. Equation (2.34) shows how to compute $q_{s_1 \rightarrow x_1}$.

$$q_{s_1 \rightarrow x_1} = \text{CHK}(r_{x_2 \rightarrow s_1} \oplus r_{x_4 \rightarrow s_1}) \quad (2.34)$$

On the other hand, we can also get $r_{x_1 \rightarrow s_1}$ by $q_{s_2 \rightarrow x_1}$ and L_1 , where L_1 is the initialization value. The initialization value L_1 will be discussed later. Equation (2.35) shows how to compute $r_{x_1 \rightarrow s_1}$.



$$r_{x_1 \rightarrow s_1} = \text{VAR}(q_{s_2 \rightarrow x_1} \oplus L_1) \quad (2.35)$$

There is CHK function in equation (2.34) and VAR function in equation (2.35). The two special functions will be introduced in the following contents. In the Tanner graph, we can compute the solid line message $q_{s \rightarrow x}$ by the dotted line messages $r_{x \rightarrow s}$ which are connected to the same check node. In the same way, we can compute the

dotted line message $r_{x \rightarrow s}$ by the real line messages $q_{s \rightarrow x}$ which are connected to the same bit node. So the values of $r_{x \rightarrow s}$ and $q_{s \rightarrow x}$ are updated iteratively. We call this iterative decoding.

Decoding Procedure in Matrix Form

Because Tanner graph is a representation of the parity-check matrix H , we can also use the matrix form to replace Tanner graph form. Let us take the same parity-check matrix H in the previous section $H = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$ as an example. In equation (2.36) and equation (2.37), we define matrix Q and matrix R . The positions of the nonzero values in R and Q are the same as those of the ones in H .

$$Q = \begin{bmatrix} q_{s_1 \rightarrow x_1} & q_{s_1 \rightarrow x_2} & 0 & q_{s_1 \rightarrow x_4} \\ q_{s_2 \rightarrow x_1} & 0 & q_{s_2 \rightarrow x_3} & q_{s_2 \rightarrow x_4} \end{bmatrix} \quad (2.36)$$

$$R = \begin{bmatrix} r_{x_1 \rightarrow s_1} & r_{x_2 \rightarrow s_1} & 0 & r_{x_4 \rightarrow s_1} \\ r_{x_1 \rightarrow s_2} & 0 & r_{x_3 \rightarrow s_2} & r_{x_4 \rightarrow s_2} \end{bmatrix} \quad (2.37)$$

The elements in the matrix Q are computed by the elements in the matrix R , for example, $q_{s_1 \rightarrow x_1} = CHK(r_{x_2 \rightarrow s_1} \oplus r_{x_4 \rightarrow s_1})$. On the other hand, the elements in the matrix R are computed by the elements in the matrix Q . For example, $r_{x_1 \rightarrow s_1} = VAR(q_{s_2 \rightarrow x_1} \oplus L_1)$, where L_1 is the initialization value. So the elements in matrix R and Q are updated iteratively. We can also regard the CHK function as the horizontal step and VAR function as the vertical step in the decoding procedure.

In the LDPC iterative decoding procedure, there are two main functions: *VAR* and *CHK*. Equation (2.38) shows the *VAR* function with two inputs and equation (2.39) is the general form of the *VAR* function.

$$VAR(q_1 \oplus q_2) = q_1 + q_2 \quad (2.38)$$

$$VAR(q_1 \oplus q_2 \oplus \dots \oplus q_l) = q_1 + q_2 + \dots + q_l \quad (2.39)$$

The *VAR* function is fixed regardless of the decoding algorithms. It is just a summation operation.

The *CHK* function with two inputs can be reformulated in different forms.

There are

$$\begin{aligned} CHK(L_1 \oplus L_2) &= 2 \tanh^{-1} \left(\tanh\left(\frac{L_1}{2}\right) \times \tanh\left(\frac{L_2}{2}\right) \right) \\ &= \text{sign}(L_1) \text{sign}(L_2) \phi(\phi(|L_1|) + \phi(|L_2|)) \end{aligned} \quad (2.40)$$

where

$$\phi(x) = -\ln \left(\tanh\left(\frac{x}{2}\right) \right) = \ln \left(\frac{e^x + 1}{e^x - 1} \right) \quad \text{and} \quad \phi(\phi(x)) = x, \quad (2.41)$$

and

$$\begin{aligned} CHK(L_1 \oplus L_2) &= \ln(\cosh(\frac{L_1 + L_2}{2})) - \ln(\cosh(\frac{L_1 - L_2}{2})) \\ &= \left| \frac{L_1 + L_2}{2} \right| - \left| \frac{L_1 - L_2}{2} \right| + \ln \frac{1 + e^{-|L_1 + L_2|}}{1 + e^{-|L_1 - L_2|}} \\ &= \text{sign}(L_1) \times \text{sign}(L_2) \times \min(|L_1|, |L_2|) + \ln \frac{1 + e^{-|L_1 + L_2|}}{1 + e^{-|L_1 - L_2|}} \end{aligned} \quad (2.42)$$

$$\approx \text{sign}(L_1) \times \text{sign}(L_2) \times \min(|L_1|, |L_2|). \quad (2.43)$$

When *CHK* function is in the form of equation (2.40) or equation (2.42), we call the decoding algorithm as sum-product algorithm. The fourth term $\ln \frac{1 + e^{-|L_1 + L_2|}}{1 + e^{-|L_1 - L_2|}}$ in equation (2.42) is called the correction factor. When the check node computation is in

the form of equation (2.43), or in other words an approximate form, we call it the min-sum algorithm.

The above discussion of check node computation is only about the *CHK* function with two inputs. Now, we will discuss the general form of the *CHK* function. The general form of the *CHK* function can be expressed in equation (2.44).

$$CHK(L_1 \oplus L_2 \oplus \dots \oplus L_l) = CHK(CHK(\dots CHK(CHK(L_1 \oplus L_2) \oplus L_3) \dots) \oplus L_l) \quad (2.44)$$

The purpose of equation (2.44) is to unfold $CHK(L_1 \oplus L_2 \oplus \dots \oplus L_l)$. The procedure is: first, compute $a_1 = CHK(L_1 \oplus L_2)$, then $a_2 = CHK(a_1 \oplus L_3)$, ..., $a_{l-1} = CHK(a_{l-2} \oplus L_l)$. The computation result of equation (2.44) is a_{l-1} . This can be viewed as serial computation. Figure 2.8 shows the serial configuration for the general form of the *CHK* function.

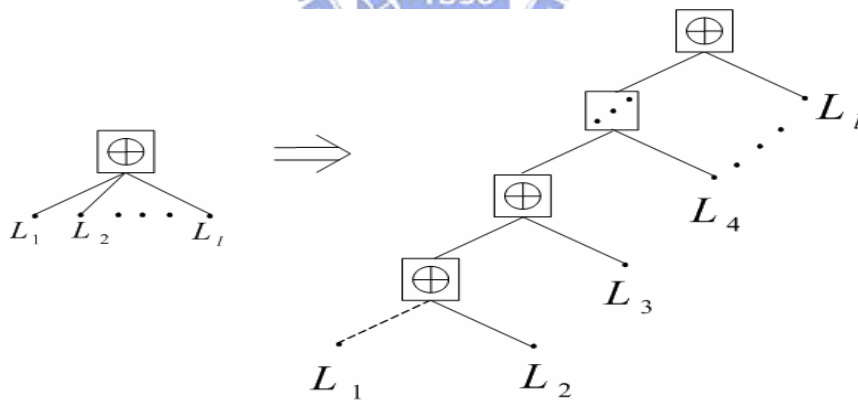


Figure 2.8 Serial configuration for check node update function

The serial computation has a long critical path in the check node update unit. From equations (2.40), (2.43), and (2.44), we can generalize the *CHK* function as equation (2.45) for sum-product algorithm, and equation (2.46) for min-sum algorithm.

$$CHK(L_1 \oplus L_2 \oplus \dots \oplus L_l) = \prod_{i=1}^l \text{sign}(L_i) \phi[\phi(|L_1|) + \phi(|L_2|) + \dots + \phi(|L_l|)] \quad (2.45)$$

where $\phi(x) = \ln\left(\frac{e^x + 1}{e^x - 1}\right)$

$$CHK(L_1 \oplus L_2 \oplus \dots \oplus L_l) = \text{sign}(L_1) \cdot \text{sign}(L_2) \cdot \dots \cdot \text{sign}(L_l) \min[|L_1|, |L_2|, \dots, |L_l|] \quad (2.46)$$

Equations (2.45) and (2.46) tell us that the check node update function can also be viewed as parallel configuration. If we derive the check node update function in parallel configuration, the critical path of the check node update function will be reduced. Figure 2.9 and 2.10 respectively show the check node updating function of the sum-product algorithm and the min-sum algorithm. These two figures neglect the multiplication of the sign symbols for an artistic view of the figures.

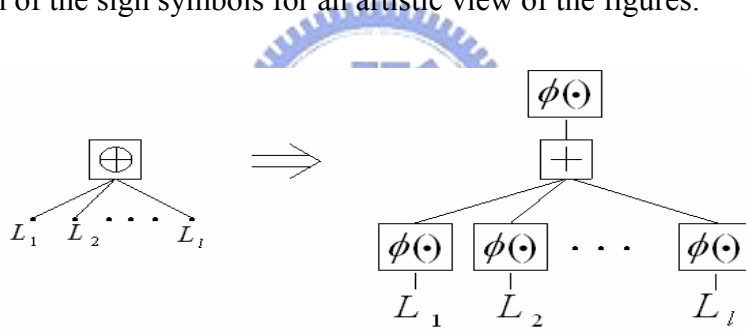


Figure 2.9 Check node update function of sum-product algorithm

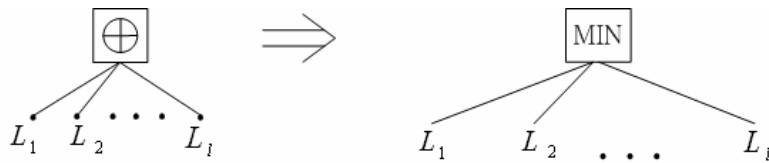


Figure 2.10 Check node update function of min-sum algorithm

Iterative Decoding Procedure [12]

The discussion in section 2.4.2 is only part of the whole iterative decoding procedure. Now, we consider the actual decoding procedure. It means that there will involve many iterations for a decoding process. First, let us describe some notations for the iterative decoding procedure in Figure 2.11. $M(l)$ denotes the set of check nodes that are connected to the variable node l , i.e., positions of “1”s in the l^{th} column of the parity-check matrix. $L(m)$ denotes the set of variable nodes that participate in the m^{th} parity-check equation, i.e., the positions of “1”s in the m^{th} row of the parity-check matrix. $L(m) \setminus l$ represents the set $L(m)$ excluding the l^{th} variable node and $M(l) \setminus m$ represents the set $M(l)$ excluding the m^{th} check node. $q_{m \rightarrow l}$ denotes the probability message that check node m sends to variable node l . $r_{l \rightarrow m}$ denotes the probability message that variable node l sends to check node m . The probability message of $q_{m \rightarrow l}$ and $r_{l \rightarrow m}$ are computed in LLR domain. The iterative decoding procedure is shown below.

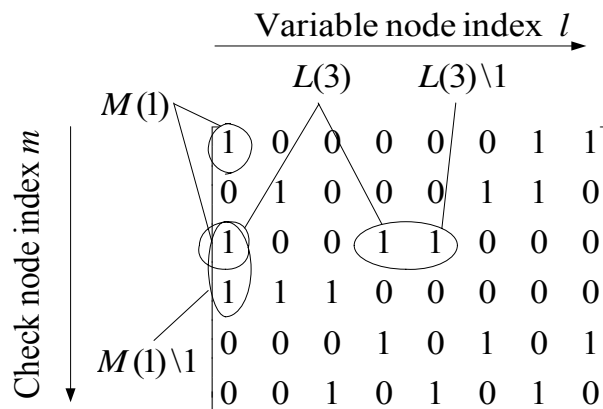


Figure 2.11 Notations for iterative decoding procedure

1. Initialization

Let

$$L_l = \ln \frac{P(y_l | x_l = 1)}{P(y_l | x_l = 0)} = \frac{2}{\sigma^2} y_l \quad (2.46)$$

be the log likelihood ratio of a variable node, where $P(a|b)$ specifies that given b is transmitted, the probability that the receiver receives a , where σ^2 is the noise variance of the Gaussian channel. For every position (m, l) such that $H_{m,l} = 1$, $q_{m \rightarrow l}$ is initialized as

$$q_{m \rightarrow l} = L_l. \quad (2.47)$$

2. Message passing

Step1 (message passing from check nodes to variable nodes): Each check node m gathers all the incoming message $q_{m \rightarrow l}$'s, and update the message on the variable node l based on the messages from all other variable nodes connected to the check node m .

$$r_{l \rightarrow m} = \text{CHK} \left(\sum_{l' \in L(m) \setminus l} \oplus q_{m \rightarrow l'} \right). \quad (2.48)$$

$L(m)$ denotes the set of variable nodes that participate in the m^{th} parity-check equation. $L(m)$ can also be viewed as the horizontal set in the parity check matrix H .

Step2 (message passing from variable nodes to check nodes): Each variable node l passes its probability message to all the check nodes that are connected to it.

$$q_{m \rightarrow l} = \text{VAR} \left(\text{VAR} (r_{l \rightarrow m}), L_l \right) = L_l + \sum_{m \in M(l) \setminus l} r_{l \rightarrow m} \quad (2.49)$$

Step3 (decoding): For each variable node l , messages from all the check nodes that are connected to the variable node l are summed up.

$$q_l = \text{VAR} \left(\text{VAR} (r_{l \rightarrow m}), L_l \right) = L_l + \sum_{m \in M(l)} r_{l \rightarrow m}. \quad (2.50)$$

Hard decision is made on q_l . The decoded vector \hat{x} is decided as

$$x_l = \begin{cases} 0, & q_l > 0 \\ 1, & q_l \leq 0 \end{cases}, 0 \leq l < n.$$

The resulting decoded vector \hat{x} is checked against the parity-check equation $H\hat{x}^T = 0$. If $H\hat{x}^T = 0$, the decoder stops and outputs \hat{x} .

Otherwise, it goes to step1 until the parity-check equation is procured or the specific

maximum iteration number is reached. The whole LDPC decoding procedure can be

expressed in Figure 2.12.

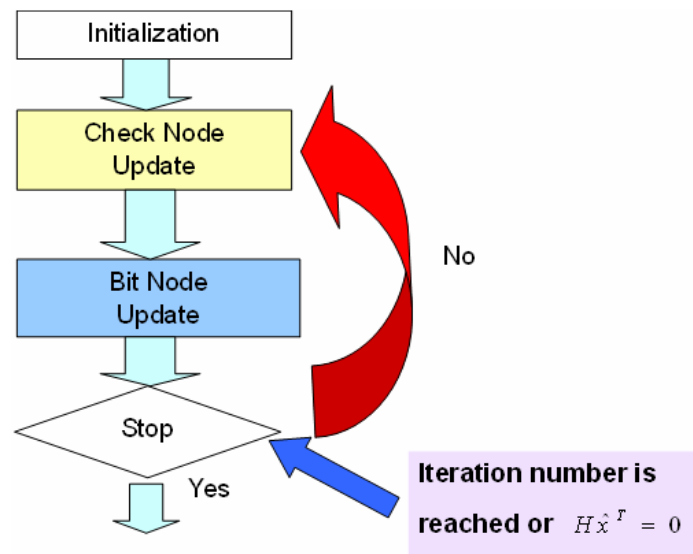


Figure 2.12 The whole LDPC decoding procedure

Table 2.4 Summary of sum-product algorithm

<p>1. Initialization:</p> <p>For $1 \leq l \leq n$</p> $L_l = \ln \frac{P(y_l x_l = 0)}{P(y_l x_l = 1)} = \frac{2}{\sigma^2} y_l, \text{ where } \sigma^2 \text{ is the noise variance}$ <p>For every l, m such that $H_{m,l} = 1$</p> $q_{m \rightarrow l} = L_l$ <p>2. Message passing:</p> <p>Step1: Message passing from check nodes to variable nodes. For each l, m,</p> <p>compute $r_{l \rightarrow m} = \text{CHK}(\sum_{i \in L(m) \setminus l} \oplus q_{m \rightarrow i})$</p>

$$= \text{sign}(q_{m \rightarrow l}) \prod_{l' \in L(m)} \text{sign}(q_{m \rightarrow l'}) \times \phi[\phi(\sum_{l' \in L(m)} |q_{m \rightarrow l'}|) - \phi(|q_{m \rightarrow l}|)]$$

where $\phi(x) = -\ln\left(\tanh\left(\frac{x}{2}\right)\right) = \ln\left(\frac{e^x + 1}{e^x - 1}\right)$ and $\phi(\phi(x)) = x$.

Step2: Message passing from variable nodes to check nodes. For each l, m ,

$$\text{compute } q_{m \rightarrow l} = \text{VAR}\left(\text{VAR}(r_{l \rightarrow m'}, L_l), L_l\right) = L_l + \sum_{m' \in M(l) \setminus m} r_{l \rightarrow m'}$$

Step3: Decoding

For each l ,

$$q_l = \text{VAR}\left(\text{VAR}(r_{l \rightarrow m}), L_l\right) = L_l + \sum_{m \in M(l)} r_{l \rightarrow m}$$

For $1 \leq l \leq n$,

$$\hat{x}_l = 0 \text{ if } q_l > 0, \hat{x}_l = 1 \text{ if } q_l < 0$$

If $(H\hat{x}^T = 0, \text{ then } \hat{x} \text{ is the estimated codeword,}$
or the iteration number is reached a predetermined threshold)

\Rightarrow the algorithm stops

else

\Rightarrow return to step1

Table 2.5 Summary of min-sum algorithm

1. Initialization:

For $1 \leq l \leq n$

$$L_l = \ln \frac{P(y_l | x_l = 0)}{P(y_l | x_l = 1)} = \frac{2}{\sigma^2} y_l, \text{ where } \sigma^2 \text{ is the noise variance}$$

For every l, m such that $H_{m,l} = 1$

$$q_{m,l} = L_l$$

2. Message passing:

Step1: Message passing from check nodes to variable nodes. For each l, m ,

compute

$$r_{l \rightarrow m} = \text{CHK} \left(\sum_{l' \in L(m) \setminus l} \oplus q_{m \rightarrow l'} \right)$$

$$= \text{sign}(q_{m \rightarrow l}) \prod_{l' \in L(m)} \text{sign}(q_{m \rightarrow l'}) \times \min \{ |q_{m \rightarrow l'}| \}$$

Step2: Message passing from variable nodes to check nodes. For each l, m ,
compute

$$q_{m \rightarrow l} = \text{VAR} \left(\text{VAR}_{m' \in M(l) \setminus m} (r_{l \rightarrow m'}), L_l \right) = L_l + \sum_{m' \in M(l) \setminus m} r_{l \rightarrow m'}$$

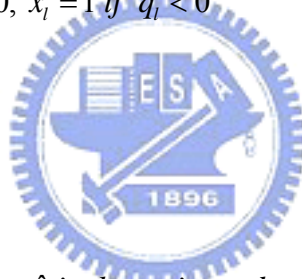
Step3: Decoding

For each l ,

$$q_l = \text{VAR} \left(\text{VAR}_{m \in M(l)} (r_{l \rightarrow m}), L_l \right) = L_l + \sum_{m \in M(l)} r_{l \rightarrow m}$$

For $1 \leq l \leq n$,

$$\hat{x}_l = 0 \text{ if } q_l > 0, \hat{x}_l = 1 \text{ if } q_l < 0$$



If $(H\hat{x}^T = 0)$, then \hat{x} is the estimated codeword ,
or the number of iteration exceeds a predetermined threshold
 \Rightarrow the algorithm stops
else
 \Rightarrow return to step 1

Chapter 3

Modified Min-Sum Algorithms

In this chapter, we will introduce modified LDPC decoding algorithms. As mentioned in chapter 2, the sum-product algorithm has better performance than min-sum algorithm. In the following, we will depict the difference between sum-product algorithm and min-sum algorithm. Our final goal is to modify min-sum algorithm in order to achieve decoding performances close to sum-product algorithm.



3.1 Normalization Technique for Min-Sum Algorithm [14]

Equation (3.1) is the check node updating function in the sum-product algorithm.

In equation (3.1), there is a major function $\phi(x) = \ln\left(\frac{e^x + 1}{e^x - 1}\right)$. The function plot of $\phi(x)$ is shown in Figure 3.1. Implementation of the nonlinear function $\phi(x)$ is complicated. Even the commonly adopted table-look-up scheme suffers loss in error performance because of the large quantization error, especially when x is small.

$$CHK(L_1 \oplus L_2 \oplus \dots \oplus L_l) = \prod_{i=1}^w \text{sign}(L_i) \phi[\phi(|L_1|) + \phi(|L_2|) + \dots + \phi(|L_l|)] \quad (3.1)$$

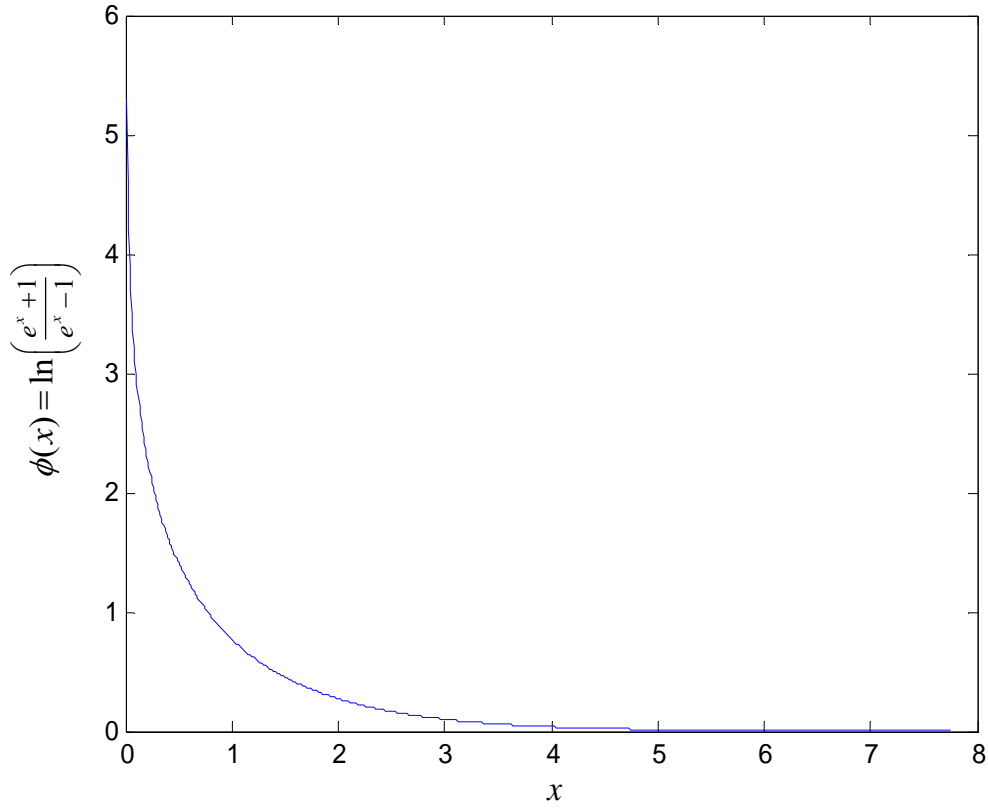


Figure 3.1 Function plot of $\phi(x)$

Equation (3.2) is the check node update function used in min-sum algorithm. The key part of equation (3.2) is to find the minimal value among w numbers: $\min[|L_1|, |L_2|, \dots, |L_w|]$. The value of w is decided by the row weight of the parity check matrix H and it's usually small (say, 6 or 7). Therefore, the min-sum algorithm is more suitable to for implementation in hardware.

$$CHK(L_1 \oplus L_2 \oplus \dots \oplus L_w) = \prod_{i=1}^w \text{sign}(L_i) \min[|L_1|, |L_2|, \dots, |L_w|] \quad (3.2)$$

As we mentioned in chapter 2, equation (3.2) is an approximate form of equation (3.1). Assume the result of equation (3.1) is A and the that of equation (3.2) is B . In [14], it proves the following two statements about the relationship between A and B .

Statements:

(1) Values A and B have the same sign, i.e., $sign(A) = sign(B)$;

(2) The absolute magnitude of B is always greater than that of A , i.e., $|B| > |A|$

Statement (1) is quite straightforward because $\phi(x)$ and $\min(x)$ are both positive functions. For convenience of proving statement (2), we assume $|B| = |L_i|$, where i is an arbitrary number between 1 and w .

$$\phi(|L_i|) < \phi(|L_1|) + \phi(|L_2|) + \dots + \phi(|L_w|)$$

Take the function $\phi()$ on both sides, one has

$$\Rightarrow \phi(\phi(|L_i|)) > \phi[\phi(|L_1|) + \phi(|L_2|) + \dots + \phi(|L_w|)]$$

$$\Rightarrow |L_i| > \phi[\phi(|L_1|) + \phi(|L_2|) + \dots + \phi(|L_w|)] \text{ because } \phi(\phi(|L_i|)) = L_i, 0 < i \leq w$$

$$\Rightarrow |B| > |A|$$

Note that because $\phi(x)$ is a decreasing function, the comparison symbol should be changed if one takes the function $\phi(x)$ on both inequality sides. Hence statement (2) is proved.

These two statements suggest the use of normalization to get more accurate soft values from B . In other words, one can multiply B by a factor β which is smaller than 1 to get a better approximation of A . To determine the normalization factor β , one can consider the criterion of forcing the mean of the normalized magnitude $\beta \cdot |B|$ to equal the mean of the magnitude $|A|$ [14], i.e.

$$\beta = \frac{E(|A|)}{E(|B|)} \quad (3.3)$$

The normalization factor β that makes $\beta \cdot |B|$ equal to $|A|$ in the average sense may not be the best, but it seems a quite reasonable choice. In the following, a theoretical value of β is derived.

It is assumed the channel is a Gaussian channel with noise variance σ^2 . For convenience, one denotes the set $\{L_i : i=1,2,\dots,w\}$. Then L_i are independent, and identically distributed (i.i.d.) random variables. The probability density function (p.d.f.) of L_i depends on SNR and code rate. One can also write

$$E[|A|] = E\{\phi[\phi(L_1) + \phi(L_2) + \dots + \phi(L_w)]\} \quad (3.4)$$

$$E[|B|] = E\{\min[|L_1|, |L_2|, \dots, |L_w|]\} \quad (3.5)$$

One first generates the random vectors $\{L_1, L_2, \dots, L_w\}$, and then calculate the means of $|A|$ and $|B|$ statistically based on equations (3.4) and (3.5). The normalization factor can be obtained from equation (3.3). One can calculate equations (3.4) and (3.5) by the theory of probabilities.

First, one can calculate $E[|B|]$. Let $M_i = |L_i|$, $i=1,2,\dots,w$, so that the p.d.f. of M_i is

$$f_{M_i}(m) = (f_{L_i}(l) + f_{L_i}(-l))u(l) = 2 \cdot f_{L_i}(l)u(l) \quad (3.6)$$

where $f_{L_i}(\cdot)$ is the p.d.f. of L_i and $u(l)$ is a unit-step function of l . It follows

$$\begin{aligned} P(|B| > l) &= P[\min(L_1, L_2, \dots, L_w) > l] \\ &= P[L_1 > l, L_2 > l, \dots, L_w > l] \\ &= [P(L_1) > l]^w \end{aligned} \quad (3.7)$$

The last equation in (3.7) follows from the fact that $\{L_i\}$ are i.i.d. random variables.

Since $|B| > 0$, one can write

$$\begin{aligned} E(|B|) &= \int_0^\infty [P(M_1 > m)]^w dm \\ &= \int_0^\infty \left[\int_m^\infty f_{M_1}(m_1) dm_1 \right]^w dm \\ &= \int_0^\mu \left[1 - Q\left(\frac{\mu - m}{\sigma_m}\right) + Q\left(\frac{\mu + m}{\sigma_m}\right) \right]^w dm \\ &\quad + \int_\mu^\infty \left[Q\left(\frac{m - \mu}{\sigma_m}\right) + Q\left(\frac{m + \mu}{\sigma_m}\right) \right]^w dm \end{aligned} \quad (3.8)$$

The second integration in (3.8) can be omitted and finally one obtains

$$E(|B|) \approx \int_0^\mu [1 - Q(\frac{\mu - m}{\sigma_m}) + Q(\frac{\mu + m}{\sigma_m})]^w dm \quad (3.9)$$

where $\mu = \frac{2}{\sigma^2}$, $\sigma_m = \frac{4}{\sigma^2}$, σ^2 is the channel noise variance, and

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-x^2/2} dx.$$

Next one can calculate $E[|A|]$ in equation (3.4).

$$\begin{aligned} \phi[\phi(|L_1|) + \phi(|L_2|) + \dots + \phi(|L_w|)] &= \phi[\ln\{\frac{e^{|L_1|} + 1}{e^{|L_1|} - 1} \cdot \frac{e^{|L_2|} + 1}{e^{|L_2|} - 1} \dots \frac{e^{|L_w|} + 1}{e^{|L_w|} - 1}\}] \\ &= \phi[\ln(X)] \text{ , where } X = \frac{e^{|L_1|} + 1}{e^{|L_1|} - 1} \cdot \frac{e^{|L_2|} + 1}{e^{|L_2|} - 1} \dots \frac{e^{|L_w|} + 1}{e^{|L_w|} - 1} \\ &= \ln \frac{X + 1}{X - 1} \\ &= 2(X^{-1} + \frac{X^{-3}}{3} + \frac{X^{-5}}{5} + \dots) \text{ , using Taylor's series} \end{aligned} \quad (3.10)$$

Let $p_k = E[X^{-k}]$. Since $\{L_i\}$ are i.i.d. random variables, one can get

$$\begin{aligned} p_k &= E[X^{-k}] \\ &= E\left(\prod_{i=1}^w \frac{e^{|L_i|} - 1}{e^{|L_i|} + 1}\right)^k \\ &= \{E\left[\left(\frac{e^{|L_1|} - 1}{e^{|L_1|} + 1}\right)^k\right]\}^w \\ &= [E(\tanh(|L_1|/2)^k)]^w \\ &= [E(\tanh(M_1/2)^k)]^w \end{aligned} \quad (3.11)$$

From equations (3.10) and (3.11), one can get

$$\begin{aligned} E[|A|] &= 2\{E[X^{-1}] + \frac{E[X^{-3}]}{3} + \frac{E[X^{-5}]}{5} + \dots\} \\ &= 2(p_1 + \frac{p_3}{3} + \frac{p_5}{5} + \dots) \end{aligned} \quad (3.12)$$

A few lower-order terms of equation (3.12) are enough to give a very good estimation of $E[|A|]$ in most cases. Combined with value $E[|B|]$ given in equation (3.9), one

can obtain the theoretical value of the normalization factor β . But in practical, the theoretical value of β is hard to compute. To use the theoretical value of β for different SNR values seems to be impractical. Thus, for a specific LDPC code, one can associate a fixed normalization factor through simulations.

Now, let's set the number of w (the input number of a check node updating function) to 6. This is because the row-weight of H is 6 or 7 in 802.16e standard (see appendix A). Assume

$$A = \phi[\phi(|L_1|) + \phi(|L_2|) + \dots + \phi(|L_6|)] \quad (3.13)$$

$$B = \min[|L_1|, |L_2|, \dots, |L_6|] \quad (3.14)$$

The purpose of Figure 3.2 is to find the normalization factor β . The vertical axis of Figure 3.2 is $|\beta \cdot B - A|$, and the horizontal axis is β . In hardware implementation, only a certain value of β will be chosen for finite-precision representation. For example, one can set β to be a multiple of 0.125 for simple hardware implementation. Through Figure 3.2, our objective is to choose the most appropriate β so that the value of $|\beta \cdot B - A|$ is as small as possible. From simulations, $\beta = 0.75$ is found to be a suitable value. When β is 0.75, it is shown that $|\beta \cdot B - A|$ is less than 0.2.

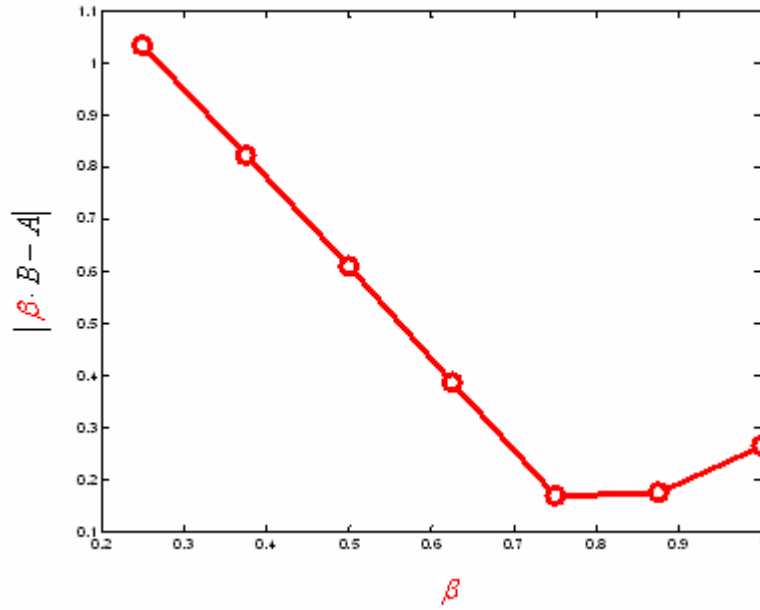


Figure 3.2 The absolute difference between the normalization technique and sum-product algorithm, vs. the normalization factor β

3.2 Dynamic Normalization Technique for Min-Sum Algorithm [23]



In section 3.1, one can use the normalized factor β to compensate the result of equation (3.2) so that it can approximate equation (3.1) more accurately. In [23], it shows the idea to adjust the normalized factor β dynamically to get better decoding performance. Thus the normalization factor β can have the form:

$$\beta = \begin{cases} \beta_1, & \text{when } B < K \\ \beta_2, & \text{when } B \geq K \end{cases} \quad (3.15)$$

In [23], it selects two normalization factors β_1 and β_2 first. For convenience of hardware implementation, only certain simple values of β_1 and β_2 should be chosen for finite-precision realizations. For check node degree of 6, it found that $\beta_1 = 0.75$ and $\beta_2 = 0.875$ are good choices. Then through simulations, one can find the optimum threshold value K to have the lowest decoder BER. The detailed

simulation results are in chapter 4.

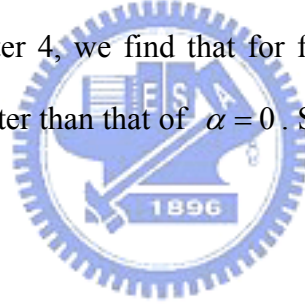
3.3 Proposed Dynamic Normalized-Offset-Compensation

Technique for Min-Sum Algorithm

Compared to the dynamic normalization technique, one can extend the idea by adding an additional offset factor α to equation (3.2) [6] in order to get even more accurate check-node updating values. Equation (3.16) shows the normalized-offset technique for min-sum algorithm.

$$CHK(L_1 \oplus L_2 \oplus \dots \oplus L_w) = \prod_{i=1}^w \text{sign}(L_i) \{ \beta \cdot \min[|L_1|, |L_2|, \dots, |L_w|] + \alpha \} \quad (3.16)$$

In section 3.1, we have decided the value 0.75 of β when the check node degree is 6. Through simulations in chapter 4, we find that for fixed value of α , the decoding performance is not always better than that of $\alpha = 0$. So we have the idea to adjust the offset factor α dynamically.



Now, we have the inspiration if the offset factor α can be dynamically adjusted to get better performance. Equation (3.17) shows the dynamic offset factor α .

$$\alpha = \begin{cases} \alpha_1, & \text{when } B < K \\ \alpha_2, & \text{when } B \geq K \end{cases} \quad (3.17)$$

Through simulations, we can decide the best values of α_1 and α_2 . As we discuss in section 3.1, In hardware implementation, only certain simple values of α_1 and α_2 will be chosen for finite-precision realizations. For check node degree of 6, we found that $\alpha_1 = 0$ and $\alpha_2 = 0.125$ are good choices.

In the following, we are going to decide the threshold K for a particular LDPC Code. Figure 3.3 shows the selection of K for rate 1/2 LDPC code vs. SNRs. $K=0$

means that we have fixed offset factor α . Otherwise, we have the dynamic offset factor α . In Figure 3.4, we can find the threshold value K equal to 1.5 is a good choice. The detail simulation results will be shown in chapter 4.

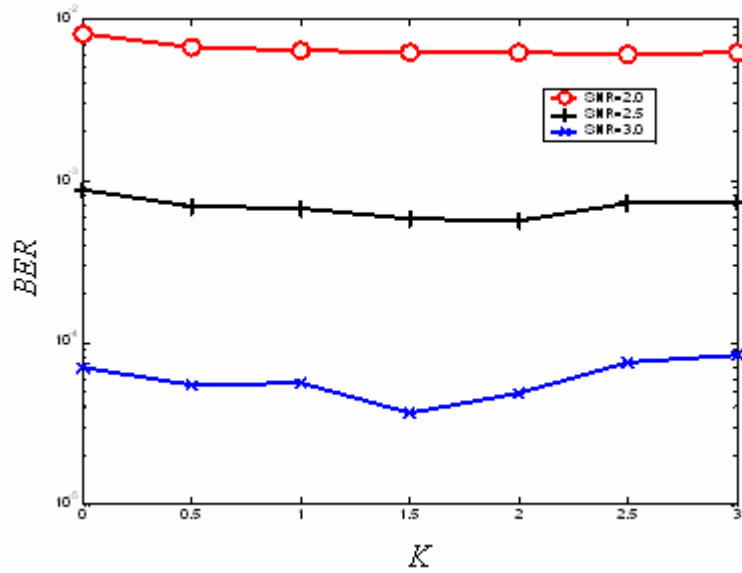


Figure 3.3 BER performance vs. threshold values K for rate 1/2 LDPC code

Chapter 4

Simulation Results and Analysis

In the beginning of this chapter, we will make a comparison of error correction performances by using different structures of the parity-check matrices such as randomly constructed code, and block-LDPC code in 802.16e standard. Then we will make a comparison of error correction performance with major decoding algorithms for LDPC codes such as sum-product algorithm, min-sum algorithm, and the proposed improved min-sum algorithm. In the end, we will furthermore analyze the finite-precision effects on the decoding performance, and decide proper word lengths of variables considering tradeoffs between the performance and the hardware cost.

Before proceeding to the following simulations, some parameters should be described here:

- 1: The randomly constructed codes are derived from [22], and they have a regular column weight and row weight.
- 2: The block-LDPC code used is for 802.16e standard.
- 3: For the decoding algorithm, we adopt the sum-product algorithm, min-sum algorithm, and the proposed modified min-sum algorithm.
- 4: We assume AWGN channels and BPSK modulation as our test environment conditions.

4.1 Floating-Point Simulations

One of the most important factors of concern when decoding the received signals is the iteration number. As the number becomes larger, the correct codewords are more likely to be decoded. However, more iterations imply higher computation cost and latency. Therefore, we need to choose a proper iteration number in the decoding process. In Figure 4.1, we show the BER simulation results vs. SNR, with different iteration numbers, for the LDPC code at rate 1/2 and length 576, BPSK, and sum-product decoding algorithm are adopted. We can find that the performance improvement tends to be insignificant after 10 iterations, which is about 0.2 dB. As a result, LDPC decoding with 10 iterations is considered as a good choice for practical implementation.

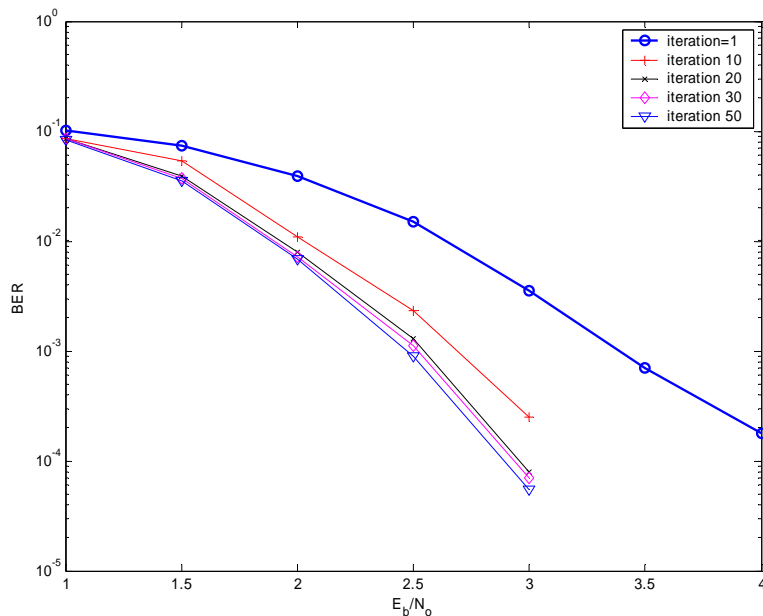


Figure 4.1 Decoding performance at different iteration numbers.

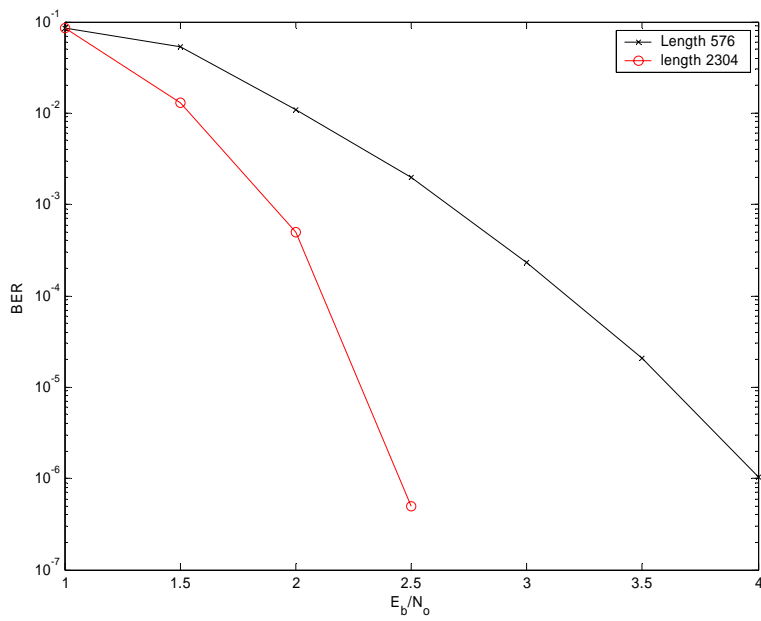


Figure 4.2 BER Performance of the rate-1/2 code at different codeword lengths, in AWGN channel, maximum iteration=10.

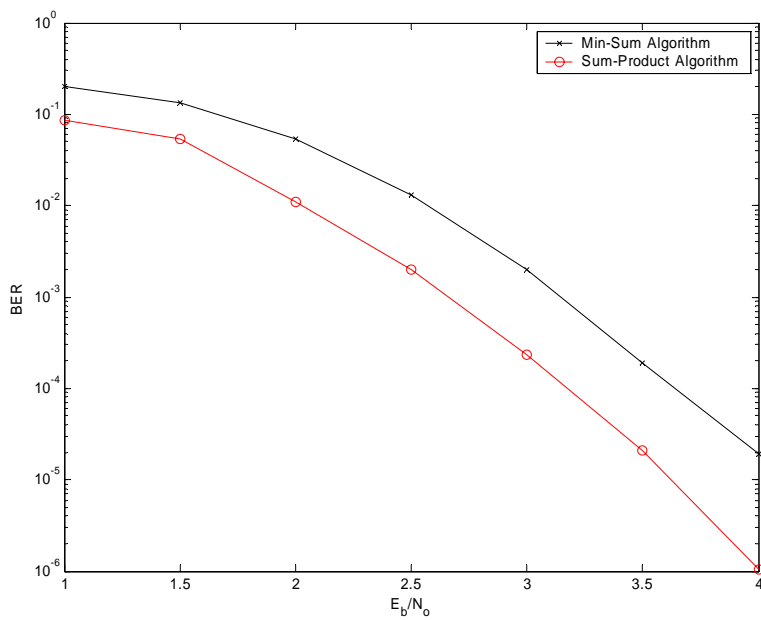


Figure 4.3 Floating-point BER simulations of two decoding algorithms in AWGN channel with code length=576, code rate=1/2, maximum iteration=10.

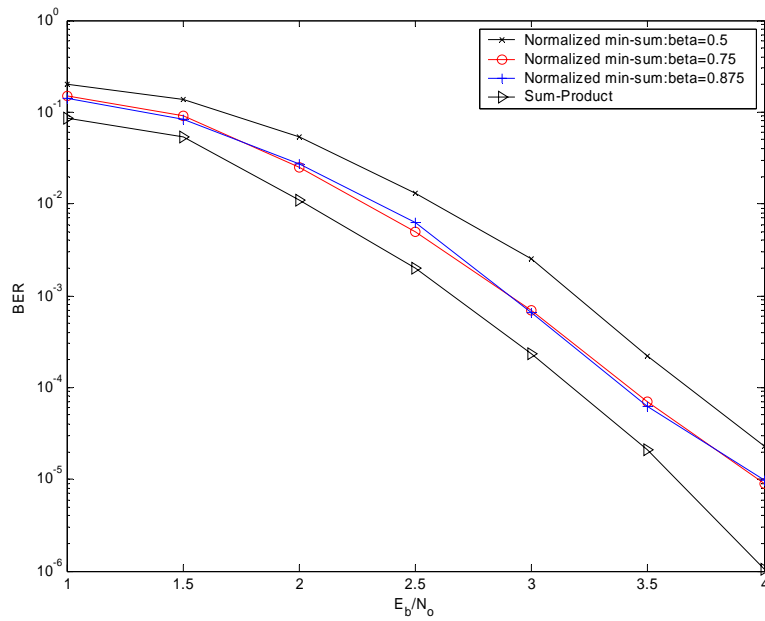


Figure 4.4 Floating-point BER simulations of normalized min-sum decoding algorithms in AWGN channel with code length=576, code rate=1/2, maximum iteration=10.

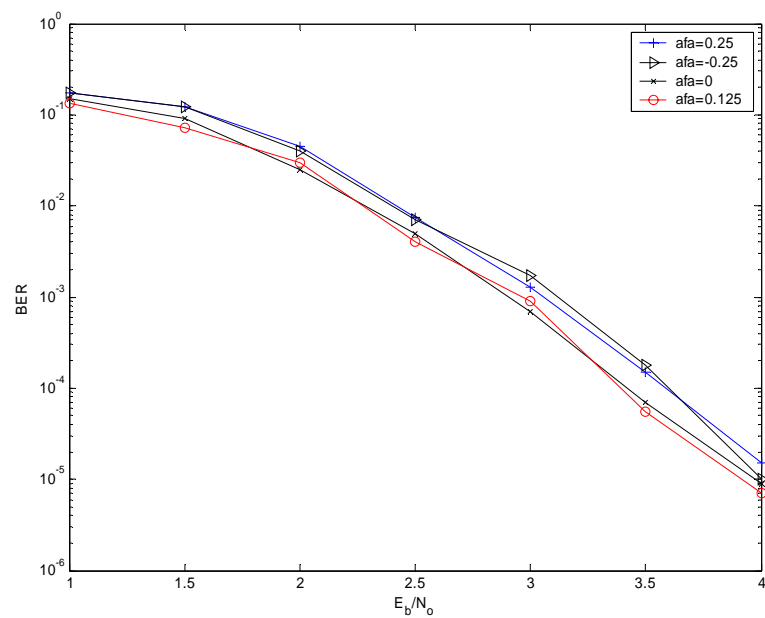


Figure 4.5 Floating-point BER simulations under normalized-offset technique in min-sum decoding algorithms, in AWGN channel with code length=576, code rate=1/2, maximum iteration=10.

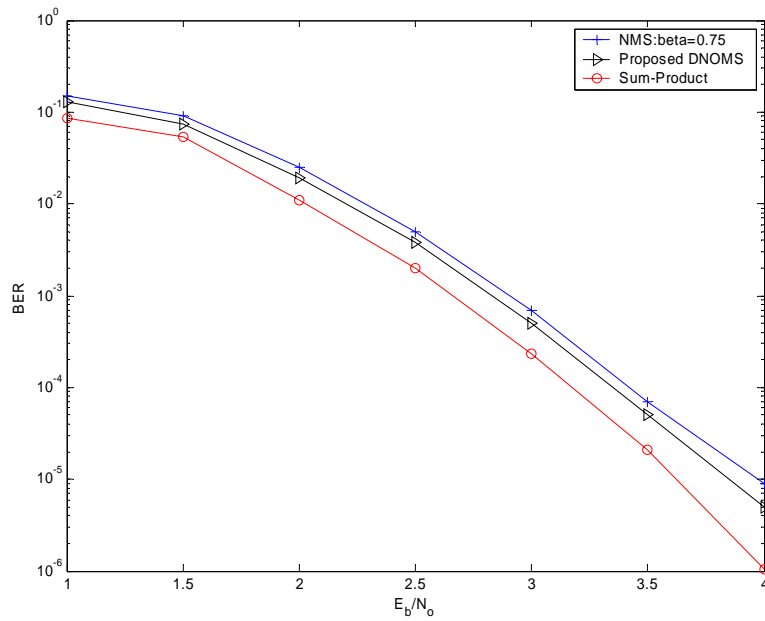


Figure 4.6 Floating-point BER simulations of the dynamic normalized-offset min-sum decoding algorithm and its comparison with other algorithms, in AWGN channel with code length=576, code rate=1/2.

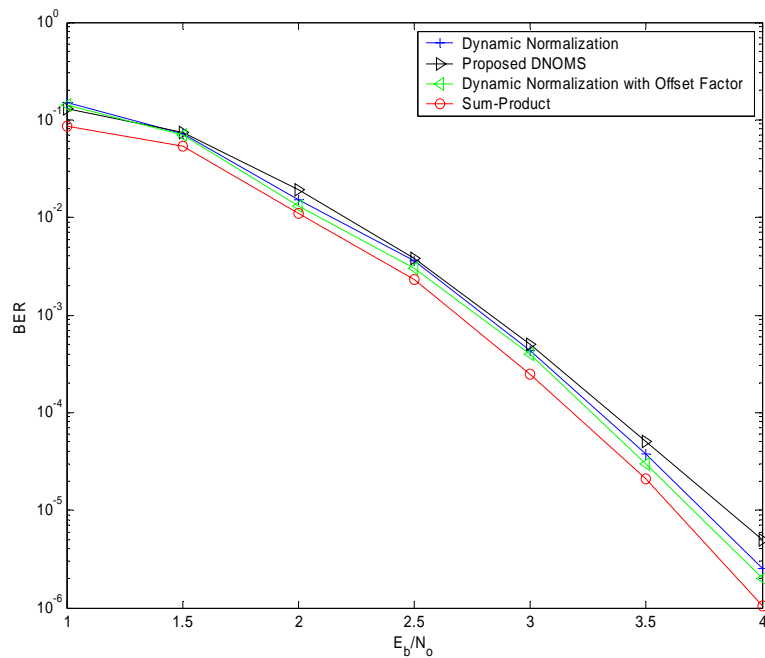


Figure 4.7 Floating-point BER simulations under normalized-offset-compensated technique and dynamic normalization technique in min-sum algorithm.

4.2 Fixed-Point Simulations

In this section, we furthermore analyze the finite-word-length performance of the LDPC decoder. Possible tradeoff between hardware complexity and decoding performance will be discussed. Let $[t:f]$ denote the quantization scheme in which a total of t bits are used, and f bits are used for the fractional part of the values. Various quantization configurations such as $[6:3]$, $[7:3]$, $[8:4]$ are investigated here.

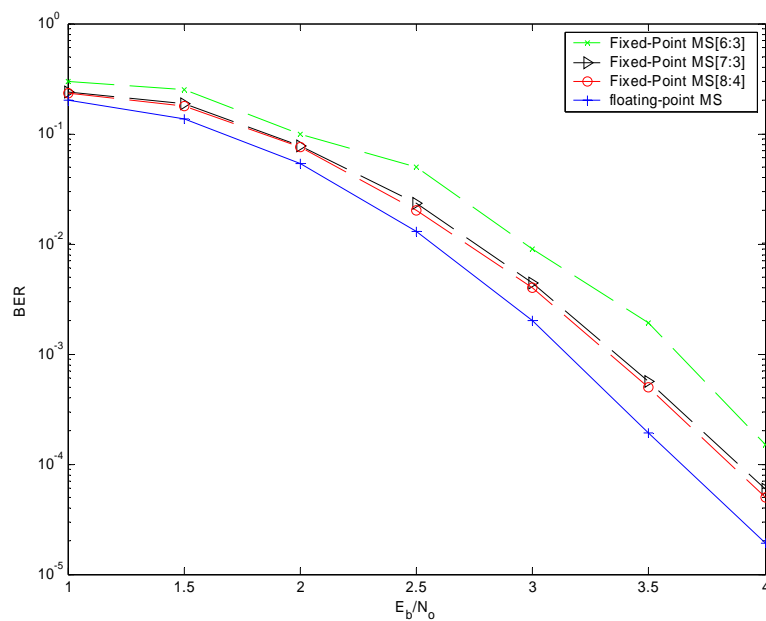


Figure 4.8 Fixed-point BER simulations of three different quantization configurations of min-sum decoding algorithm, in AWGN channel, code length=576, code rate=1/2, maximum iteration=10.

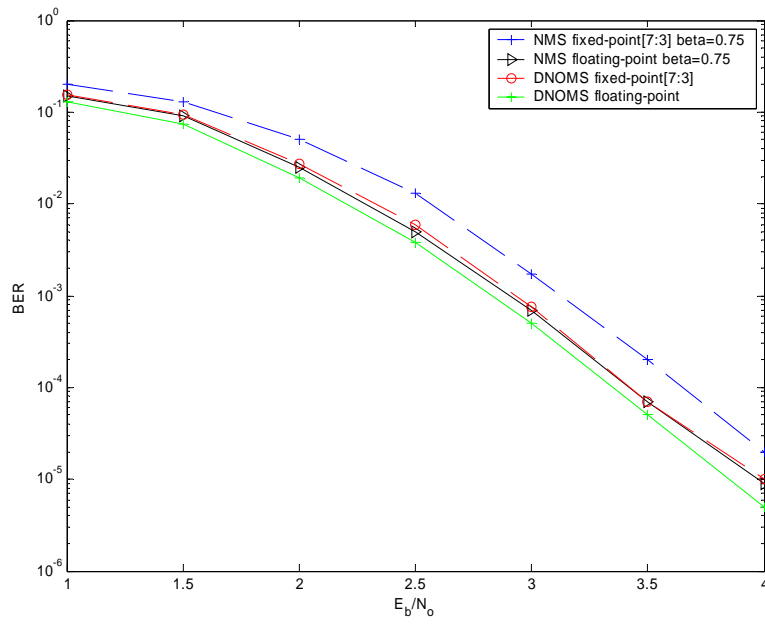


Figure 4.9 Floating-point vs. fixed-point BER simulations of the normalization and dynamic normalized-offset min-sum algorithm.



Chapter 5

Architecture Designs of LDPC Code Decoders

In this chapter, we will introduce the hardware architectures of the LDPC code decoder in our design and discuss the implementation of an irregular LDPC decoder for 802.16e standard. The decoder has a code rate 1/2 and code length of 576 bits. The parity-check matrix of this code is listed in Appendix A.


5.1 The Whole Decoder Architecture

The parity-check matrix H in our design is in block-LDPC form as we discuss in section 2.2. The parity-check matrix is composed of $m_b \times n_b$ sub-matrices. The sub-matrices are zero matrices or permutation matrices with the same size of $z \times z$. The permutations used are circular right shifts, and the set of permutation matrices contains the $z \times z$ identity matrix and circular right shifted versions of the identity matrix.

$$H = \begin{bmatrix} P_{0,0} & P_{0,1} & \cdots & P_{0,n_b-1} \\ P_{1,0} & P_{1,1} & \cdots & P_{1,n_b-1} \\ \vdots & \vdots & \cdots & \vdots \\ P_{m_b-1,0} & P_{m_b-1,1} & \cdots & P_{m_b-1,n_b-1} \end{bmatrix}$$

Figure 5.1 The parity check matrix H of block-LDPC Code

In our design, we consider a LDPC code with code-rate 1/2 and 288-by-576 parity-check matrix for 802.16e standard. While considering circuit complexity, the 288-by-576 parity-check matrix H of LDPC code are divided into four 144-by-288 sub-matrices to fit partial-parallel architecture, which is shown in Figure 5.2. The LDPC code decoder architecture in our design is illustrated in Figure 5.4. This architecture contains 144 CNUs, 288 BNUs and two dedicated message memory units (MMU). The set of data processed by CNUs are $\{h_{00}, h_{01}\}$ and $\{h_{10}, h_{11}\}$, whereas the data fed into BNUs should be $\{h_{00}, h_{10}\}$ and $\{h_{01}, h_{11}\}$. Note that two MMUs are employed to process two different codewords concurrently without stalls. Therefore, the LDPC decoder is not only area-efficient but also its the decoding speed is comparable with fully parallel architectures.



$$H = \begin{array}{cc|cc} h_{00} & h_{01} & \text{CNU Set1} & \\ \hline h_{10} & h_{11} & \text{CNU Set2} & \\ \hline \text{BNU Set1} & \text{BNU Set2} & & \end{array}$$

Figure 5.2 The partition of parity-check matrix H

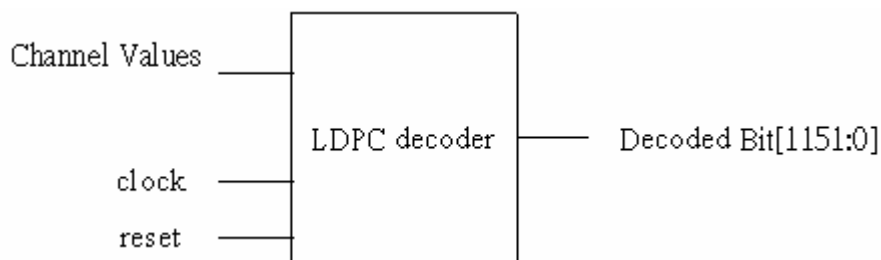


Figure 5.3 I/O pin of the decoder IP

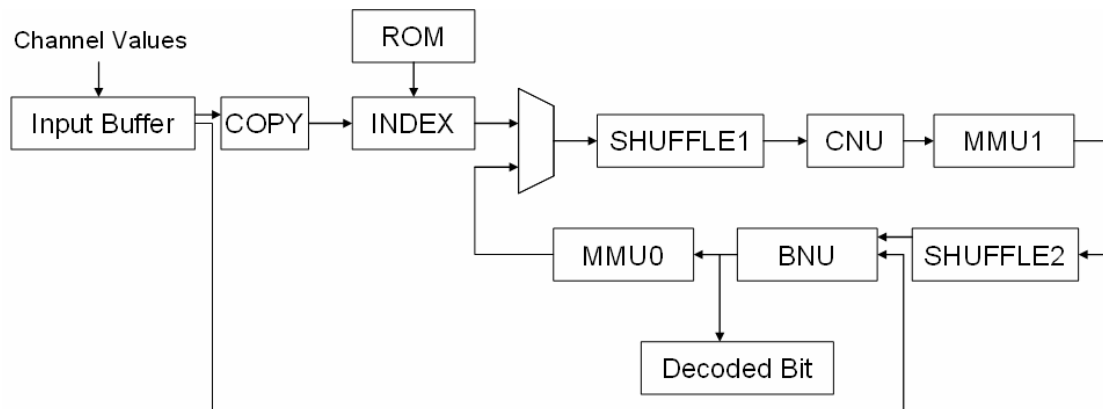


Figure 5.4 The whole LDPC decoder architecture for the block LDPC code

The I/O pin of the decoder chip is shown in Figure 5.3. Figure 5.4 shows the block diagram of the decoder architecture. The modules in it will be described explicitly in the following. We adopt partial-parallel architectures [19], so the decoder can handle 2 codewords at one time.

Input Buffer [19]

The input buffer is a storage component that receives and keeps channel values for iterative decoding. Channel values should be fed into the COPY module during initialization and BNU processing time.

COPY, INDEX, and ROM modules

The parity-check matrix H is sparse which means there are few ones in the matrix. It is not worth to save the whole parity-check matrix in the memory. So we use the module INDEX to keep the information of H . We take a simple example to explain how these modules work. Figure 5.4 shows the simple parity-check matrix.

$$H = \begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 5.5 A simple parity-check matrix example, based on shifted identity matrix.

The parity-check matrix is composed by 4 sub-matrices and the sub-matrices are right-circular-shifted matrices. The shifted numbers are expressed in Figure 5.5. Since the parity-check matrix size in this example is 8-by-8, we receive 8 channel values. The channel values are assumed to be $\vec{v} = [v_1 \ v_2 \ v_3 \ v_4 \ v_5 \ v_6 \ v_7 \ v_8]$, and then they are fed to the module “COPY”. Figure 5.6 (a) and 5.6 (b) show how modules “COPY”, “INDEX”, “ROM” work. The outputs of the module “INDEX” are $\vec{i}_1, \vec{i}_2, \vec{i}_3, \vec{i}_4$. They reserve the channel values and add the indices of the shifted numbers. The indices of the shifted numbers are stored in module “ROM.”

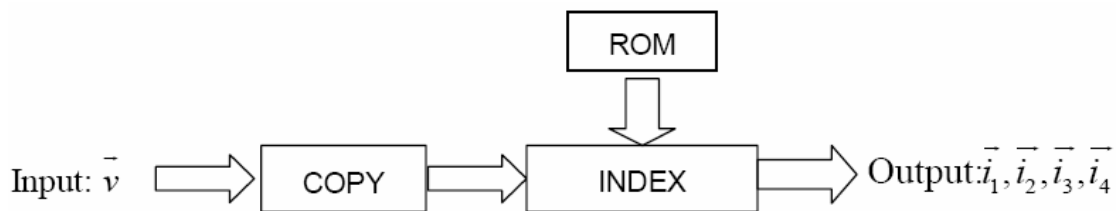


Figure 5.6 (a) The sub-modules of the whole decoder

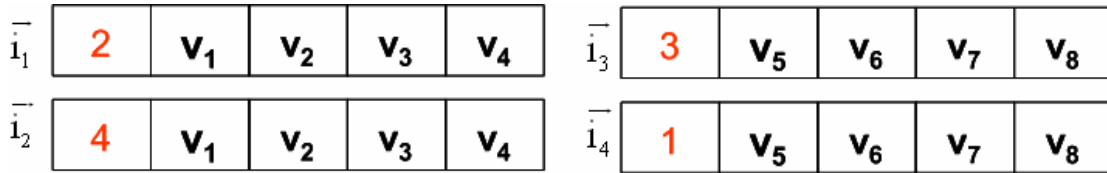


Figure 5.6 (b) The outputs of the module INDEX

The indices represent the shifted amounts and the information of H . So we place the indices in front of the channel values.

SHUFFLE1, SHUFFLE2 modules

Before sending the values to the check-node update unit, we have to shuffle left the values in order to give the correct positions when doing check-node computation and shuffle right the values before doing the bit-node computation. The amount of the shuffling value is decided by the index numbers. Figure 5.7(a) and 5.7(b) show how modules SHUFFLE1 and SHUFFLE2 work. In this example, $(v_2, v_7), (v_3, v_8), (v_4, v_5), (v_1, v_6)$ are the input pairs of the check-node update unit. Before sending the values to the bit-node update unit, we have to shuffle back the values. Thus we can have the correct answers.

$$H = [2 \ 3] = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & v_2 & 0 & 0 & 0 & 0 & v_7 & 0 \\ 0 & 0 & v_3 & 0 & 0 & 0 & 0 & v_8 \\ 0 & 0 & 0 & v_4 & v_5 & 0 & 0 & 0 \\ v_1 & 0 & 0 & 0 & 0 & v_6 & 0 & 0 \end{bmatrix}$$

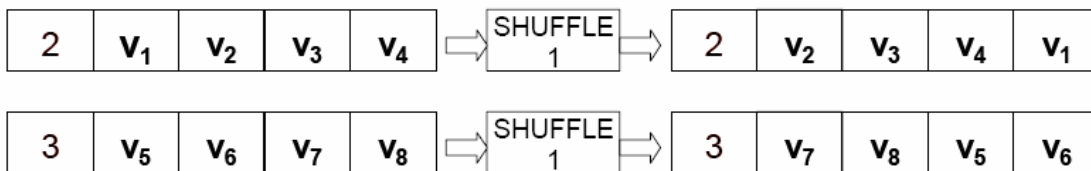


Figure 5.7(a) Values shuffling before sending to check-node update unit

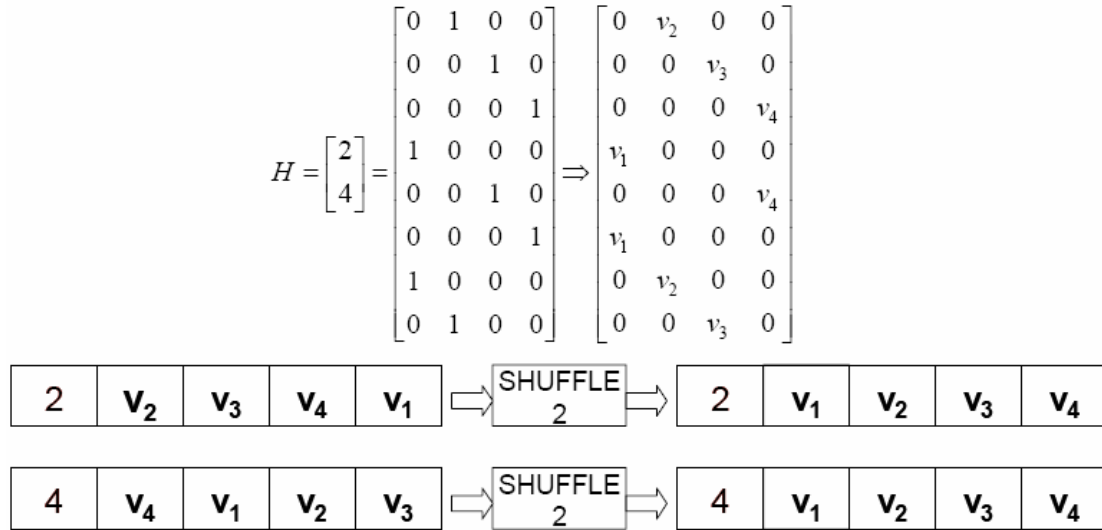


Figure 5.7(b) Values shuffling before sending to bit-node update unit

CNU[15]

Check node update units (CNUs) are used to compute the check node equation. The check-to-bit message $r_{m,l}$ for the check node m and bit node l using the incoming bit-to-check messages $q_{m,l'}$ is computed by CNU as follows

$$r_{m,l} = \prod_{l' \in L(m) \setminus l} \text{sign}(q_{m,l'}) \times \min\{q_{m,l'}\} \quad (5.1)$$

where $L(m) \setminus l$ denotes the set of bit nodes connected to the check node m except l . Figure 5.8(a) shows the architecture of the CNU using the min-sum algorithm. The check node update unit has 6 inputs and 6 outputs. In Figure 5.8(a) and 5.8(b), the output of “MIN” is the minimal value of the 2 inputs. The aim of this circuit is to find the minimal value of the other 5 inputs. This architecture is quite straightforward. Figure 5.8(b) shows the architecture of the CNU using the proposed modified min-sum algorithm.

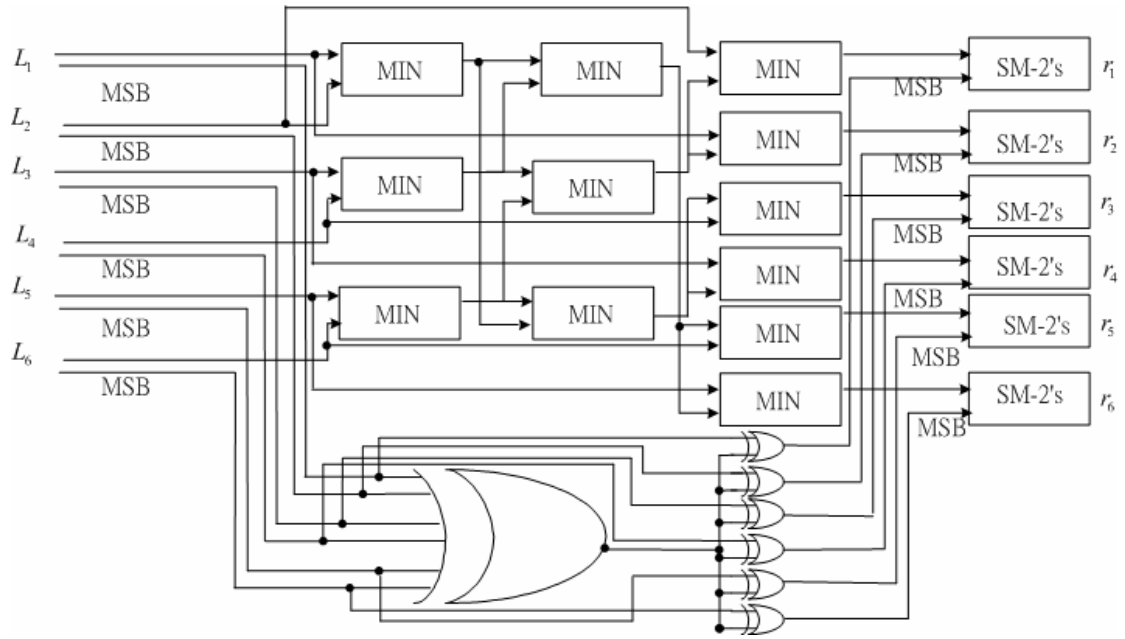


Figure 5.8(a) The architecture of CNU using min-sum algorithm

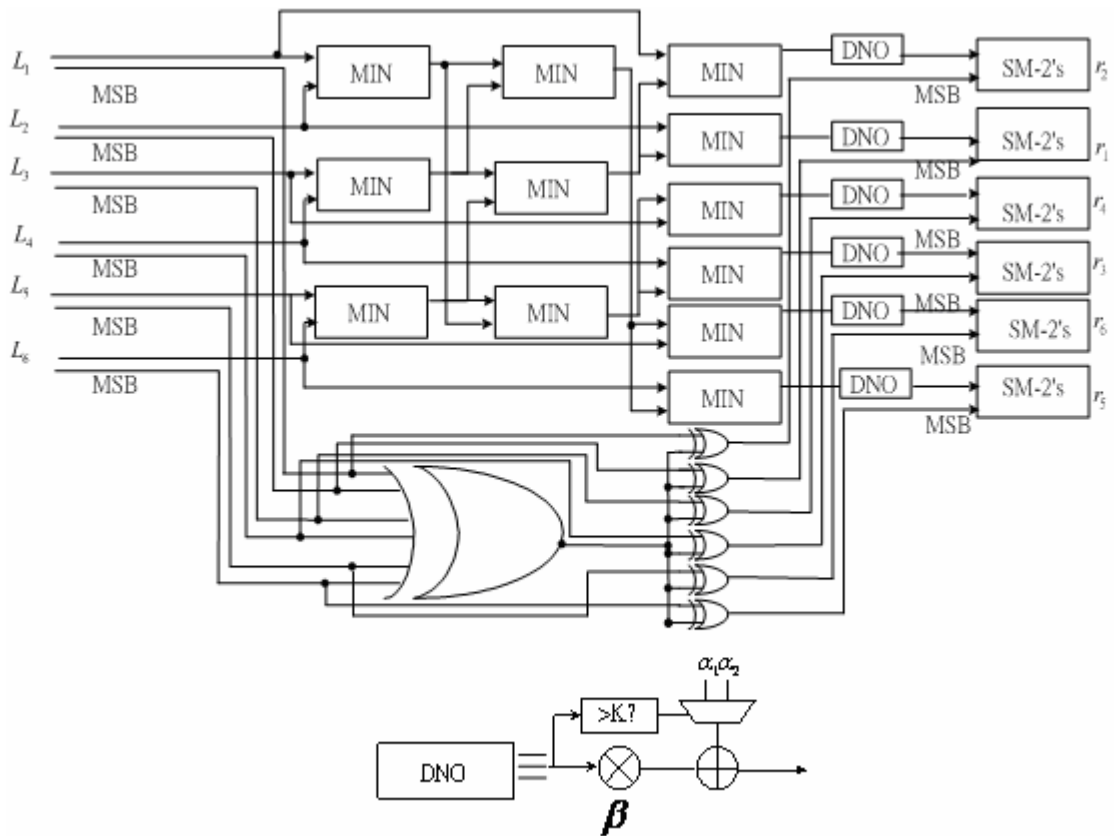


Figure 5.8(b) The architecture of CNU using modified min-sum algorithm

The other way to implement equation (5.1) is to search the minimal value and the second minimal value from inputs. Figure 5.9 shows the block diagram of the compare-select unit (CS6). The detailed architecture of CMP-6 in Figure 5.9 is illustrated in Figure 5.10, which consists of two kinds of comparators: CMP-2 and CMP-4. CMP-4 finds out the minimal and the second minimal values from the four inputs, a , b , c , and d . In addition, CMP-2 is a two input comparator which is much simpler than CMP-4.

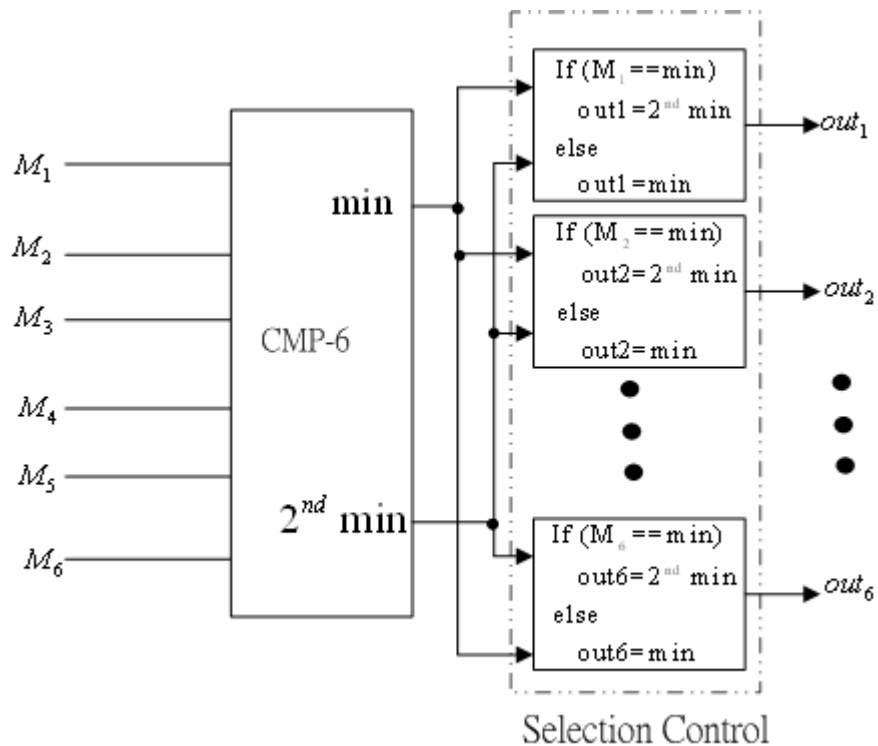


Figure 5.9 Block diagram of CS6 module

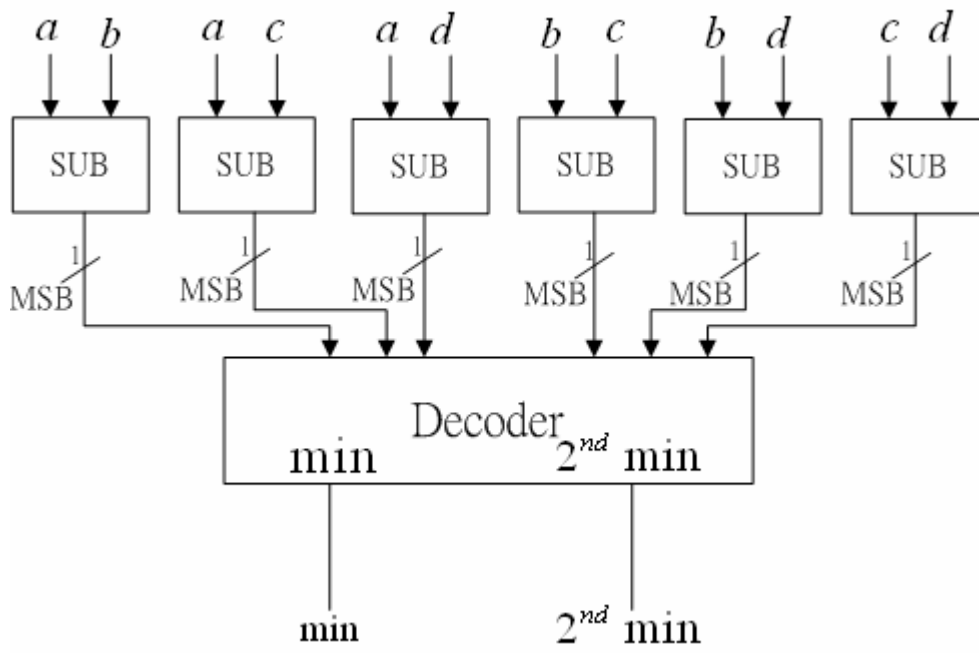


Figure 5.10(a) Block diagram of CMP-4 module

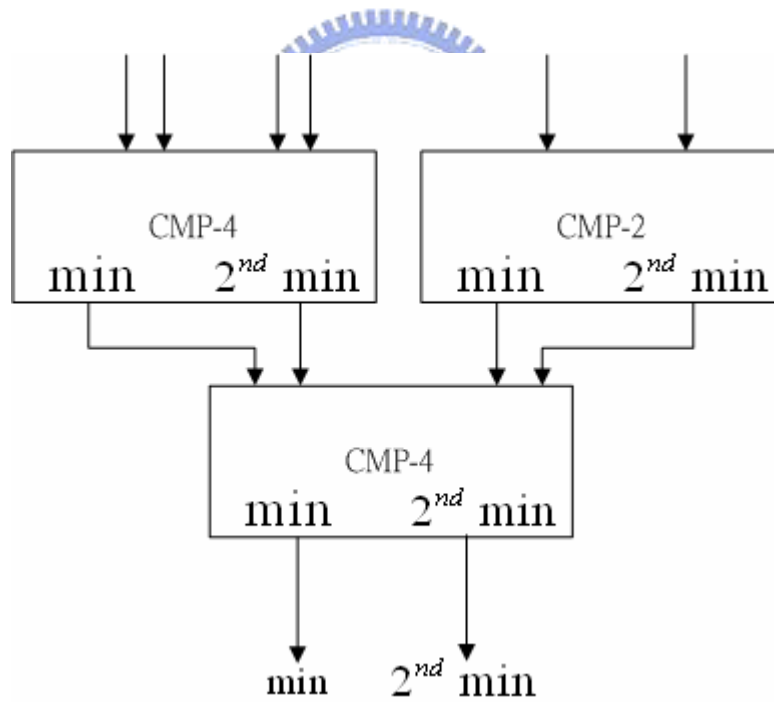


Figure 5.10(b) Block diagram of CMP-6 module

The whole architecture of the 6-input CNU is shown in Figure 5.11.

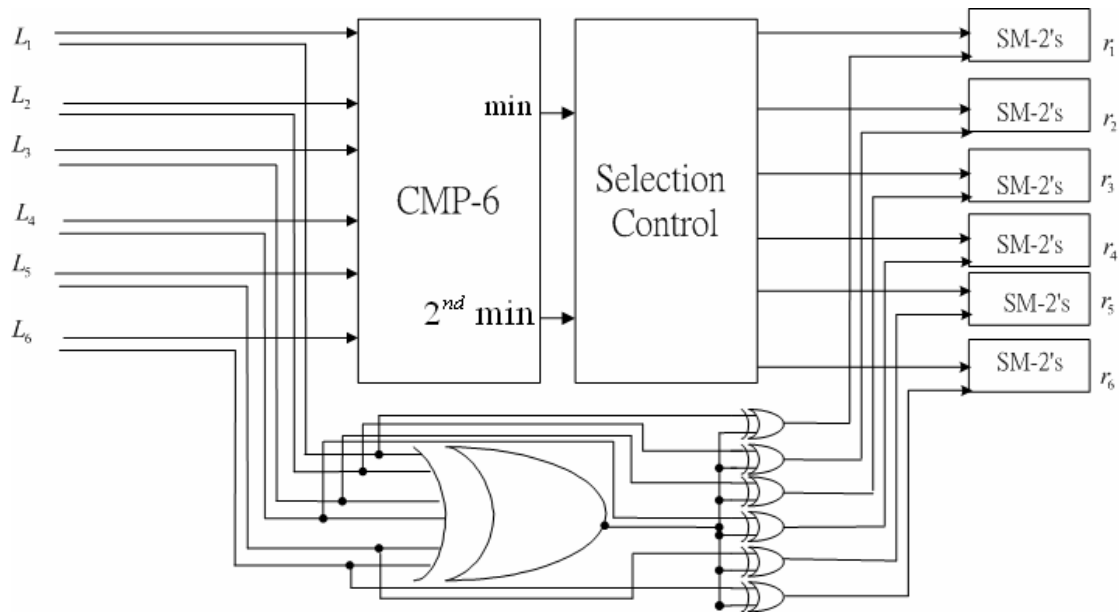


Figure 5.11 CNU architecture using min-sum algorithm



Table 5.1 compares the hardware performance of two different CNU architectures. We call the architecture in Figure 5.8(a) is direct CNU architecture and the architecture in Figure 5.11 is backhanded CNU architecture. We can find that the direct CNU architecture has only 45% size of the backhanded CNU architecture. So we choose the direct CNU architecture.

Table 5.1 Comparison of direct and backhanded CNU architectures

	Direct CNU architecture	Backhanded CNU architecture
Area (gate count)	0.52k	1.16k
Speed (MHz)	100	100
Power Consumption (mW)	4.82	10.85

BNU

Figure 5.12 shows the architecture of the bit node update unit for 4 inputs. “SM” means the sign-magnitude representation and “2’s” means the two’s complement representation. While finding the absolute minimal value of two inputs, sign-magnitude representation is more suitable for hardware implementation than two’s complement. In contrast, while adding computation, two’s complement representation is more suitable for hardware implementation than sign-magnitude representation.

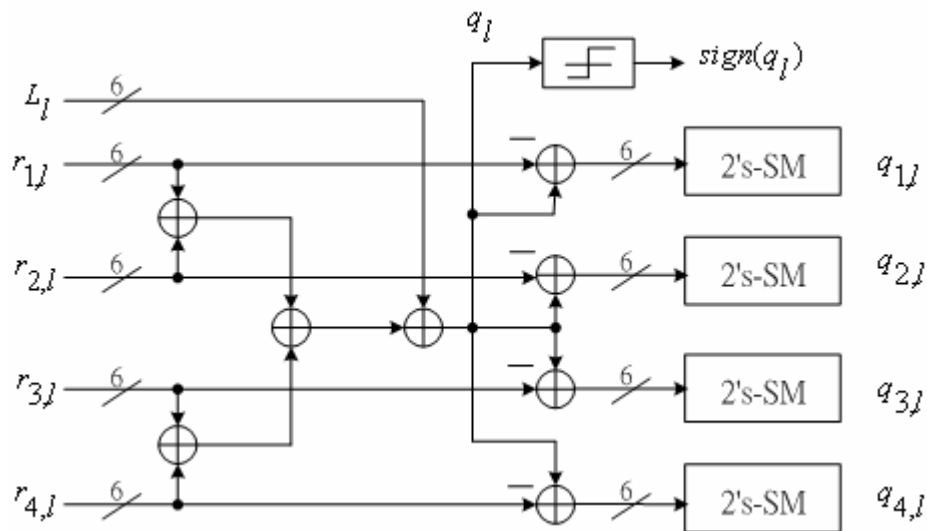


Figure 5.12 The architecture of the bit node updating unit with 4 inputs

MMU0 and MMU1 [19]

In [19], it introduces a partial-parallel decoder architecture that can increase the decoder throughput with moderate decoder area. We adopt the partial-parallel architecture in our design and make an improvement in the message memory units.

Message memory units (MMU) are used to store the message values that are generated by CNUs and BNUs. To increase the decoding throughput, two MMUs are employed to concurrently process two different codewords in the decoder. The register exchange scheme based on four sub-blocks (RE-4B) is proposed as shown in Figure 5.13(a). In MMU, sub-blocks *A*, *B*, *D* capture the outputs from CNU while sub-blocks *C* and *D* deliver the message data to SHUFFLE2. The detailed timing diagram of MMU0 and MMU1 are illustrated in Figure 5.13(b). $h_{xy}^{(0)}$ means the copied message of codeword 0 and $h_{xy}^{(1)}$ means that of codeword 1.

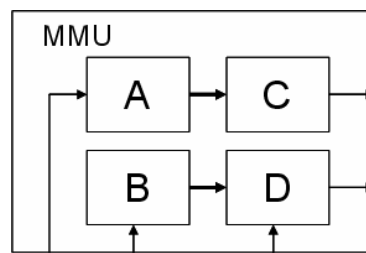


Figure 5.13(a) The architecture of RE-4B based MMU

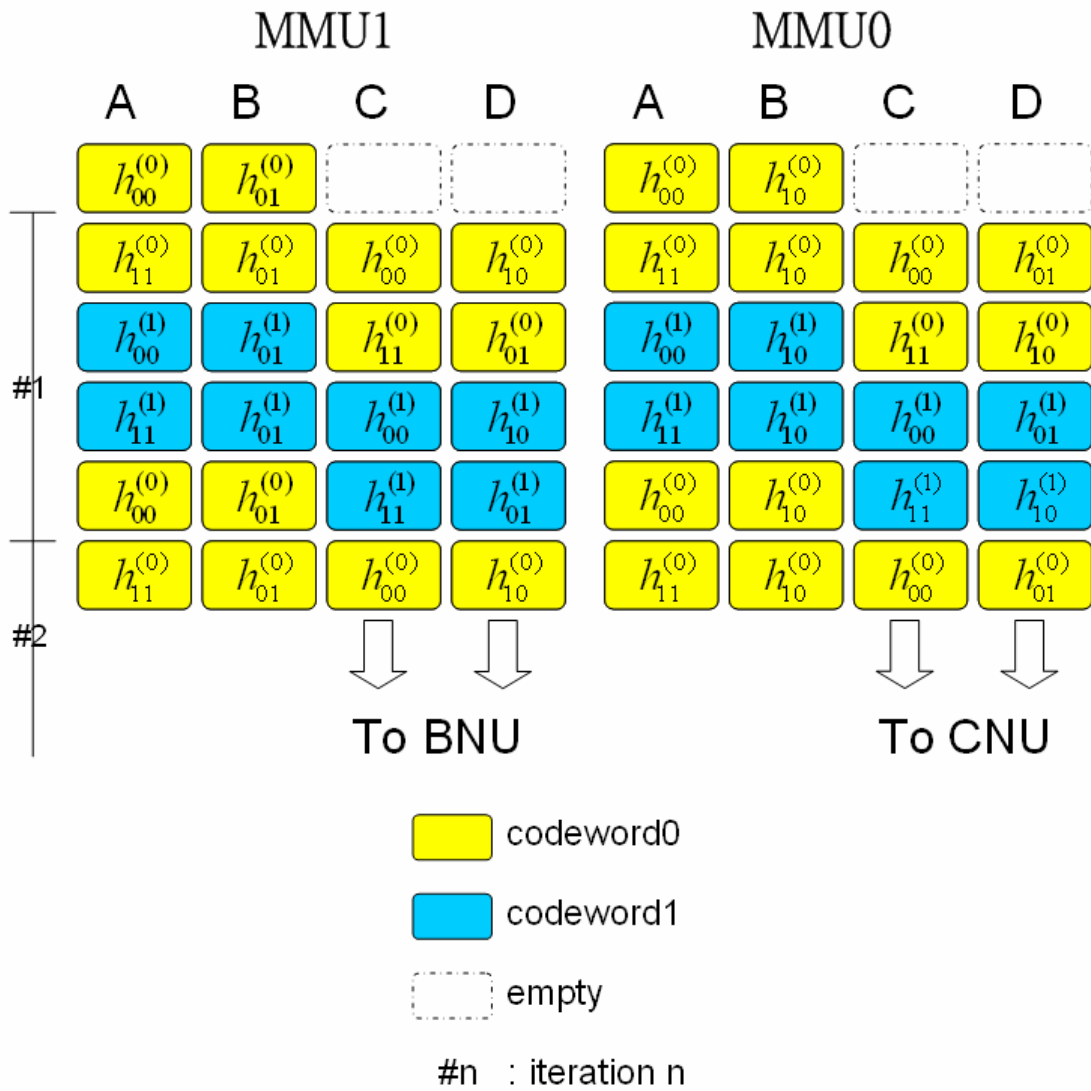


Figure 5.13(b) The timing diagram of the message memory units

While in the iterative decoding procedure, MMU0 and MMU1 pass messages to each other through SHUFFLE1, CNU, SHUFFLE2, and BNU modules. Disregarding the combinational circuit, the detailed relationship and snapshots between MMU0 and MMU1 is shown in Figure 5.14.

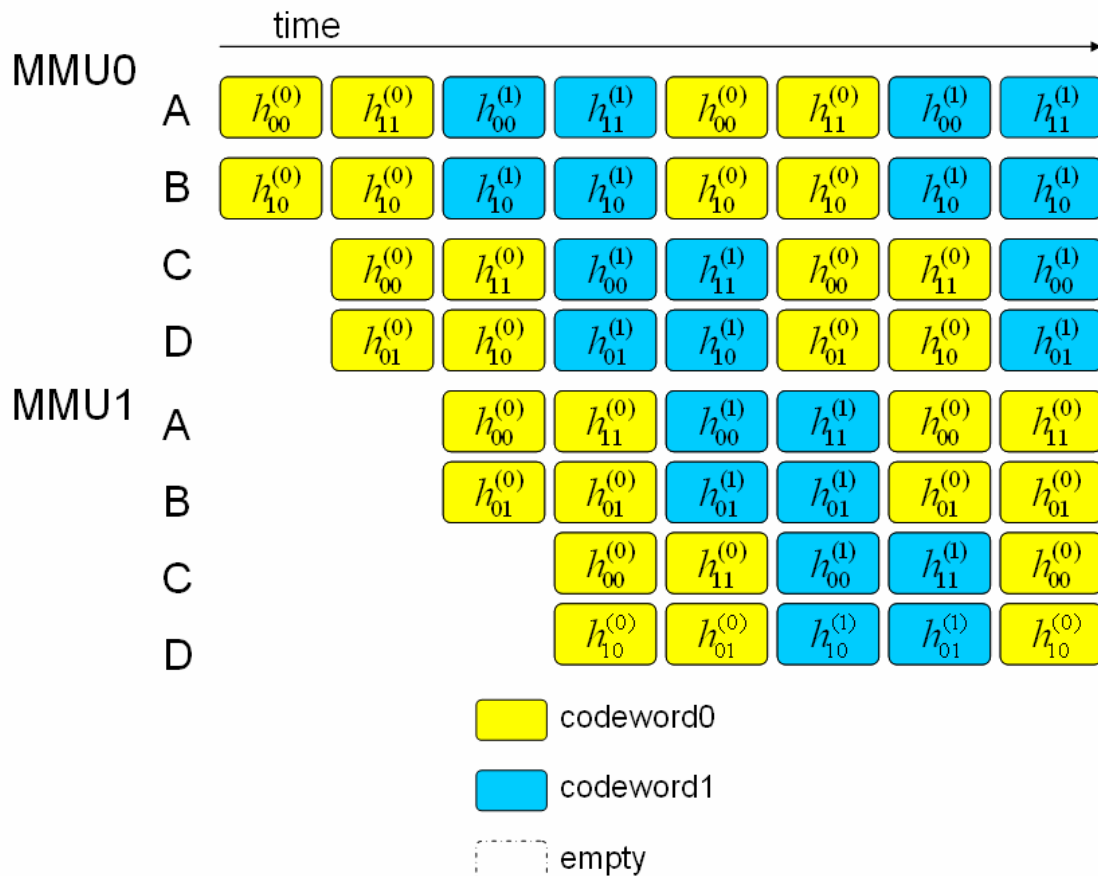


Figure 5.14 The message passing snapshots between MMU0 and MMU1

5.2 Hardware Performance Comparison and Summary

To compare the area, speed, latency, and power consumption of the architectures discussed in this section, we describe the hardware architectures in VHDL, and afterwards simulate and synthesize it using EDA tools SynopsisTM, PrimePower, and DesignAnalyzer. The process technology is UMC 0.18 μm process. Table 5.2 lists the results of CNU using min-sum algorithm and the proposed modified min-sum algorithm.

Table 5.2 Area, speed, and power consumption of the CNU using min-sum algorithm and modified min-sum algorithm

	6 input CNU	6 input CNU (modified)	7 input CNU	7 input CNU (modified)
Area (gate count)	0.52k	0.57k	0.72	0.79
Speed (MHz)	100	100	100	100
Power Consumption (mW)	4.82	4.96	6.77	7.1

As mentioned before, two different codewords are processed concurrently without any stalls. In our proposed design, BNUs and CNUs have no idle time. Hence, it leads to an efficient utilization of the functional units. The design takes four cycles to complete a decoding iteration for each codeword, including two cycles for horizontal steps in CNUs and two cycles for vertical steps in BNUs. For channel value loading, each codeword takes two extra cycles. Since the maximum iteration of the decoding procedure is 10, the total amount of cycles needed to complete the decoding of two different codewords is $2+2+10*4=44$ cycles. According to our initial synthesis results, the clock frequency is 100MHz, thus the data decoding throughput is $100*[1152*(1/2)]/44 \approx 1.31$ Gbps.

The proposed LDPC decoder is compared with other designs as listed in Table 5.3. The objective of our design is to devise a high throughput LDPC decoder with little chip area. Partial-parallel decoder architecture can meet our demand. Compared with [19], our design has lower data throughput. Because our decoder design has shorter code length and lower code rate. In our design, one codeword has 288 message bits. In [19], one codeword has 720 bits. Moreover considering the BER

performance, we choose the iteration number=10. This also reduces the data throughput. The superiority of our design is the chip area. Although we choose higher quantization bits, the chip area in our design has 82.6% of the design in [19] and 54.3% of the design in [17].

Table 5.3 Comparison of LDPC decoders

	Proposed LDPC decoder	[19]	[17]
Code length	576	1200	1024
Code rate	1/2	3/5	1/2
Quantization bits	7	6	4
Iteration number	10	8	10
Architecture	Partial-parallel	Partial-parallel	Fully-parallel
Process Technology (μm)	0.18	0.18	0.16
Clock rate (MHz)	100	83	64
Power (mW)	620	644	690
Area (gate count)	950k	1150k	1750k
Throughput (Mbps)	1310	3330	500

Chapter 6

Conclusions and Future Work

6.1 Conclusions

From this work, we summarize that using dynamic normalized-offset technique in LDPC decoder can further improve the error correction performance when compared with the conventional method. Various simulation results of LDPC decoder are investigated and the optimal choice considering the tradeoff between the hardware complexity and the performance have been discussed in this thesis.

In this thesis, with partial-parallel architecture, high-throughput and area-efficient LDPC code decoders are proposed for high-speed communication systems. A (576, 288) LDPC code in 802.16e standard has been implemented, of which the code rate is 1/2, the code length is 576 bits, and the maximum number of decoding iterations is 10. The LDPC decoder in our design can achieve a data throughput of 1.31 Gbps and the chip area is 950k gates using the UMC 0.18 μm process technology.

6.2 Future Work

The normalization factor β and the offset factor α influence the decoder BER performance quite large. Through our research, we found that our proposed dynamic normalized-offset technique and dynamic normalization technique [23] have

similar BER decoding performance. The other idea is to dynamically adjust the two factors α and β in the same time. The threshold values of α and β may be obtained through simulations. Moreover, as mentioned in Appendix A, there are a lot of different codeword lengths and code rates in 802.16e standard. Our future work is to integrate the multi-mode 802.16e LDPC decoder design.



Appendix A

LDPC Codes Specification in IEEE 802.16e

OFDMA

The LDPC code in IEEE802.16e is a systematic linear block code, where k systematic information bits are encoded to n coded bits by adding $m = n - k$ parity-check bits. The code-rate is k/n .

The LDPC code in IEEE802.16e is defined based on a parity-check matrix H of size $m \times n$ that is expanded from a binary base matrix H_b with size $m_b \times n_b$, where $m = z \cdot m_b$ and $n = z \cdot n_b$. In this standard, there are six different base matrices. One for the rate 1/2 code is depicted in Figure A.1. Two different ones for two rate 2/3 codes, type A is in Figure A.2 and type B is in Figure A.3. Two different ones for two rate 3/4 codes, type A is in Figure A.4 and type B is in Figure A.5. One for the rate 5/6 code is depicted in Figure A.6. In these base matrices, size n_b is an integer equal to 24 and the expansion factor z is an integer between 24 and 96. Therefore, we can compute the minimal code length as $n_{\min} = 24 \times 24 = 576$ bits and the maximum code length as $n_{\max} = 24 \times 96 = 2304$ bits.

For codes 1/2, 2/3B, 3/4A, 3/4B, and 5/6, the shift sizes $p(f, i, j)$ for a code size corresponding to the expansion factor z_f are derived from $p(i, j)$, which is the element at the i -th row, j -th column in the base matrices, by scaling $p(i, j)$ proportionally as

Rate 2/3 B code:

2	-1	19	-1	47	-1	48	-1	36	-1	82	-1	47	-1	15	-1	95	0	-1	-1	-1	-1	-1	-1
-1	69	-1	88	-1	33	-1	3	-1	16	-1	37	-1	40	-1	48	-1	0	0	-1	-1	-1	-1	-1
10	-1	86	-1	62	-1	28	-1	85	-1	16	-1	34	-1	73	-1	-1	-1	0	0	-1	-1	-1	-1
-1	28	-1	32	-1	81	-1	27	-1	88	-1	5	-1	56	-1	37	-1	-1	-1	0	0	-1	-1	-1
23	-1	29	-1	15	-1	30	-1	66	-1	24	-1	50	-1	62	-1	-1	-1	-1	-1	0	0	-1	-1
-1	30	-1	65	-1	54	-1	14	-1	0	-1	30	-1	74	-1	0	-1	-1	-1	-1	-1	0	0	-1
32	-1	0	-1	15	-1	56	-1	85	-1	5	-1	6	-1	52	-1	0	-1	-1	-1	-1	-1	0	0
-1	0	-1	-1	-1	13	-1	61	-1	84	-1	55	-1	78	-1	41	95	-1	-1	-1	-1	-1	-1	0

Figure A.3 Base matrix of the rate 2/3, type B code

Rate 3/4 A code:

6	38	3	93	-1	-1	-1	30	70	-1	86	-1	37	38	4	11	-1	46	48	0	-1	-1	-1	-1
62	94	19	84	-1	92	78	-1	15	-1	-1	92	-1	45	24	32	30	-1	-1	0	0	-1	-1	-1
71	-1	55	-1	12	66	45	79	-1	78	-1	-1	10	-1	22	55	70	82	-1	-1	0	0	-1	-1
38	61	-1	66	9	73	47	64	-1	39	61	43	-1	-1	-1	-1	95	32	0	-1	-1	0	0	-1
-1	-1	-1	-1	32	52	55	80	95	22	6	51	24	90	44	20	-1	-1	-1	-1	-1	-1	0	0
-1	63	31	88	20	-1	-1	-1	6	40	56	16	71	53	-1	-1	27	26	48	-1	-1	-1	-1	0

Figure A.4 Base matrix of the rate 3/4, type A code

Rate 3/4 B code:

-1	81	-1	28	-1	-1	14	25	17	-1	-1	85	29	52	78	95	22	92	0	0	-1	-1	-1	-1
42	-1	14	68	32	-1	-1	-1	-1	70	43	11	36	40	33	57	38	24	-1	0	0	-1	-1	-1
-1	-1	20	-1	-1	63	39	-1	70	67	-1	38	4	72	47	29	60	5	80	-1	0	0	-1	-1
64	2	-1	-1	63	-1	-1	3	51	-1	81	15	94	9	85	36	14	19	-1	-1	-1	0	0	-1
-1	53	60	80	-1	26	75	-1	-1	-1	86	77	1	3	72	60	25	-1	-1	-1	-1	0	0	
77	-1	-1	-1	15	28	-1	35	-1	72	30	68	85	84	26	64	11	89	0	-1	-1	-1	-1	0

Figure A.5 Base matrix of the rate 3/4, type A code

Rate 5/6 code:

1	25	55	-1	47	4	-1	91	84	8	86	52	82	33	5	0	36	20	4	77	80	0	-1	-1
-1	6	-1	36	40	47	12	79	47	-1	41	21	12	71	14	72	0	44	49	0	0	0	0	-1
51	81	83	4	67	-1	21	-1	31	24	91	61	81	9	86	78	60	88	67	15	-1	-1	0	0
50	-1	50	15	-1	36	13	10	11	20	53	90	29	92	57	30	84	92	11	66	80	-1	-1	0

Figure A.6 Base matrix of the rate 5/6 code

References

- [1] R. G. Gallager, Low-density parity-check codes, Cambridge, MA: MIT Press, 1963.
- [2] D. J. C. Mackay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electron. Lett.*, Vol. 32, pp. 1645-1646, Aug. 1996.
- [3] T. J. Richardson and R. L. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inform. Theory*, Vol. 47, pp. 638-656, Feb. 2001.
- [4] D. J. C. Mackay, S. T. Wilson, and M. C. Davey, "Comparison of constructions of irregular gallager codes," *IEEE Trans. Comm.*, Vol. 47, pp. 1449-1454, Oct. 1999.
- [5] S. J. Johnson and S. R. Weller, "A family of irregular LDPC codes with low encoding complexity," *IEEE Comm. Lett.*, Vol. 7, pp. 79-81, Feb. 2003.
- [6] J. Chen, A. Dholakia, E. Eleftheriou, and M.P.C. Fosooyer, and X.Y. Hu, "Reduced-complexity decoding of LDPC codes," *IEEE Trans. Commun.*, Vol. 53, pp. 1288-1299, July 2005.
- [7] R. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inform. Theory*, Vol. 27, pp. 533-547, Sep. 1981.
- [8] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann, "Practical loss-resilient codes," *IEEE Trans. Inform. Theory*, Vol. 47, pp. 569-584, Feb. 2001.
- [9] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, "Design of

- capacity-approaching irregular low-density parity-check codes,” IEEE Trans. Inform. Theory, Vol. 47, pp. 619-637, Feb. 2001.
- [10] D. J. C. Mackay, “Good error-correcting codes based on very sparse matrices,” IEEE Trans. Inform. Theory, Vol. 45, pp. 399-431, Mar. 1999.
- [11] F. R. Kschischang, B. J. Frey, and H. A. Loeliger, “Factor graphs and the sum-product algorithm,” IEEE Trans. Inform. Theory, Vol. 47, pp. 498-519, Feb. 2001.
- [12] H. Futaki and T. Ohtuski, “Low-density parity-check (LDPC) coded OFDM systems,” IEEE VTS, Vol. 1, pp. 82-86, Fall. 2001.
- [13] X. Y. Hu, E. Eleftheriou, D. M. Arnold, and A. Dholakia, “Efficient implementation of the sum-product algorithm for decoding LDPC codes,” IEEE GLOBECOM’01, Vol. 02, pp. 1036-1036E, Nov. 2001.
- [14] Jinghu Chen and Marc P.C. Fossorier, “Near Optimum Universal Belief Propagation Based Decoding of Low-Density Parity Check Codes,” IEEE Trans. on Commun., Vol. 50, pp. 583-587, NO.3 Mar. 2002.
- [15] Marjan Karkooti and Joseph R. Cavallaro, “Semi-parallel reconfigurable architectures for real-time LDPC decoding,” IEEE ITCC’04 Vol. 65, pp. 683-689.
- [16] Z. Wang, Y. Chen, and K. K. Parhi, “Area efficient decoding of quasi-cyclic low density parity check codes,” IEEE ICASSP’04, Vol. 5, pp. 49-52, May. 2004.
- [17] A. J. Blanksby and C. J. Howland, “A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder,” IEEE J. Solid-State Circuits, Vol. 37, pp. 404-412, Mar. 2002.

- [18] Y. Chen and D. Hocevar, "A FPGA and ASIC implementation of rate 1/2, 8088-b irregular low density parity check decoder," IEEE GLOBECOM'03, Vol. 3, pp. 113-117, Dec. 2003.
- [19] Chien-Ching Lin, Kai-Li Lin, Hsie-Chia Chang and Chen-Yi Lee, "A 3.33Gb/s (1200,720) low-density parity check code decoder," IEEE Proceedings of ESSCIRC, Grenoble, France, 2005.
- [20] T.M.N. Ngatched, M. Bossert, and A. Fahrner, "Two decoding algorithms for low-density parity check codes," IEEE ITCC'04, Vol. 32, pp. 253-257.
- [21] Yuan-Jih Chu and Sau-Gee Chen, "An efficient LDPC code structure combined with the concept of difference family," IWCMC'06, Vol. 18, pp. 355-360.
- [22] I. V. Kozintsev. Software for low-density parity-check codes. [Online] Available at: <http://www.kozintsev.net/soft.html>.
- [23] Yen-Chin Liao, Chien-Ching Lin, Chih-Wei Liu, and Hsie-Chia Chang, "A dynamic normalization technique for decoding LDPC codes," IEEE Signal Processing Systems Design and Implementation, pp. 768-772, Nov. 2005.

自 傳

邱敏杰，1982年6月15日出生，高雄縣人。2004年自國立暨南國際大學電機工程學系畢業，隨即進入國立交通大學電子研究所攻讀碩士學位。研究興趣為通訊系統與數位信號處理，碩士論文題目為低密度對偶檢查碼解碼演算法之改進以及其高速解碼器架構之設計。

