國立交通大學

電機與控制工程學系

碩士論文

適用於802.11n之Radix-4低密度同位檢查碼解碼器

A Radix-4 LDPC Decoder for 802.11n

研究生：陳宇文

指導教授：蔡尚澕 博士

中華民國九十七年十二月

A Radix-4 LDPC Decoder for 802.11n

Yu-Wen Chen

Advisor: Dr. Shang-Ho Tsai
Department of Electrical and Control Engineering
National Chico Tung University

December 3, 2008

Abstract

In this thesis, a new decoding algorithm called Radix-4 LDPC decoder is used to increase the throughput and achieve better BER performance. Moreover, a three-size (1944,972), (1296,648), and (648,324) LDPC decoder applied to IEEE 802.11n standard is implemented. The partially parallel scheme is used to decrease chip area as well as routing resource. The LDPC decoder was implemented with TSMC CMOS 18um process. The proposed decoder can achieve 292∼50Mbps decoding throughput rate under clock frequency of 62.5MHz. The core size is 17.9 mm^2 and average power consumption with a 1.62V voltage supply is 145mW.

Contents

1 Introduction **1**

2 Low-Density Parity-Check Codes **3**

2.1 Categories of error correction codes 3

2.2 Categories of LDPC 4

2.3 Encoder of LDPC code 6

2.4 Decoder of LDPC code 8

2.4.1 Sum-Product algorithm(SPA) 9

2.4.2 Log-Likelihood Ratio Sum-Product algorithm (LLR) . . . 13

2.4.3 Min-Sum with a correct factor (Min-Sum-Correct) 15

2.4.4 Min-Sum algorithm 16

2.4.5 Simulation result 17

3 Proposed Algorithm and Architecture **23**

3.1 Latency reduction 23

3.1.1 Reordering of the parity check matrix 23

3.1.2 Overlapped operation of bit node update and check node update 26

3.2 Proposed algorithm 28

3.3 The comparison of Radix-4 and Min-Sum-Correct 31

3.4 LUT circuit 35

3.5 Fixed point analysis 37

3.6 Proposed architecture 37

3.6.1 The unit for the check node update 38

3.6.2 The unit for the bit node update 41

4 VLSI implementation 43

4.1 Design flow 43

4.1.1 System model 43

4.1.2 RTL code 43

4.1.3 BIST 45

4.1.4 Synthesis 45

4.1.5 Gate-level simulation 45

4.1.6 DFT 46

4.1.7 ATPG 46

4.1.8 APR 46

4.1.9 DRC and LVS 46

4.1.10 Post-layout simulation 47

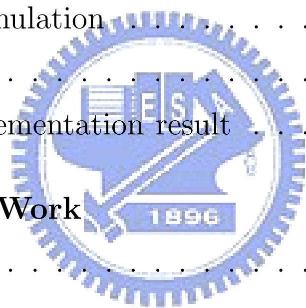
4.2 Chip layout 47

4.3 Comparison and implementation result 48

5 Conclusion and Future Work 51

5.1 Conclusion 51

5.2 Future work 51



List of Figures

2.1 A digital communication system. 3
2.2 Categories of error correction codes. 4
2.3 Codeword in systematic block code. 4
2.4 Parity check and corresponding Tanner Graph. 5
2.5 Quasi-cyclic LDPC code in 802.11n. 6
2.6 A identity matrix shifted right by 1 bit. 6
2.7 Six sub-matrices in parity check matrix **H**. 7
2.8 Block diagram of the encoder architecture for the block LDPC code. 9
2.9 A decoding flow of LDPC codes. 9
2.10 Message information form Check node to bit node. 10
2.11 Message information form bit node to check node. 12
2.12 Message information from bit node to check node. 16
2.13 The general flow of check node update. 17
2.14 LLR algorithm in 802.11n in AWGN channel (Z=81bits). 18
2.15 Min-Sum-Correct algorithm in 802.11n in AWGN channel (Z=81bits). 19
2.16 Min-Sum algorithm in 802.11n in AWGN channel (Z=81bits). . . . 19
2.17 Comparison of different algorithm with one iteration. 20
2.18 Comparison of different algorithm with ten iterations. 20
2.19 The value of check node update in different algorithm. 21
2.20 The value of check node update in different algorithm. 21
2.21 The value of check node update in different algorithm. 22
2.22 The value of check node update in different algorithm. 22

3.1 Reordering the row of the parity check matrix: (a) original matrix, and (b) reordered matrix. 24
3.2 Reordering the column of the parity check matrix: (a) original matrix, and (b) reordered matrix. 24
3.3 Reordering the parity check matrix (a)original matrix, and (b)reordered matrix. 25
3.4 The original parity check matrix of IEEE 802.11n standard. . . . 25
3.5 The reordered parity check matrix of IEEE 802.11n standard. . . 26
3.6 The reordered parity check matrix of IEEE 802.11n standard. . . 27
3.7 Timing diagram: (a) original (b) overlapped. 28
3.8 Message information from bit nodes to a check node. 28
3.9 The flow diagram of check node update using Radix-4 algorithm. . 30
3.10 Performance comparison of the proposed Radix-4 algorithm and the conventional LLR. 30
3.11 The value of check node update in different algorithm. 31
3.12 The value of check node update in different algorithm. 32
3.13 The value of check node update in different algorithm. 32
3.14 The value of check node update in different algorithm. 33
3.15 The unit for check update with 4 bit node in LLR algorithm. . . . 33
3.16 The unit for check update with 4 bit node in Radix-4 algorithm. . 34
3.17 Comparison of using LLR and Radix-4. 36
3.18 The performance of Radix-4 algorithm with fixed-point. 37
3.19 The overall architecture of the proposed LDPC decoder. 38
3.20 Table shows how many nonzero (elements that are not "-") elements in rows. 39
3.21 Case 1: A check node connected to 7 bit nodes. 39
3.22 Case 2: A check node connected to 8 bit nodes. 40
3.23 The first operation unit for the check node update. 40
3.24 The second operation unit for the check node update. 41
3.25 The unit for bit node update with 12 inputs. 42

4.1 IC design flow. 44
4.2 Layout of the proposed LDPC decoder. 47
4.3 Layout of the proposed LDPC decoder. 48

List of Tables

3.1 The comparison of Radix-4 and Min-Sum-Correct. 34

3.2 The decoding latency. 35

3.3 Quantization table for $\log(1 + e^{-|x|})$. 35

3.4 Piece-wise linear function for $\log(1 + e^{-|x|})$. 36

3.5 The proposed piece-wise linear function for $\log(1 + e^{-|x|})$. 36

4.1 Specification of the proposed LDPC decoder. 49

4.2 The comparison of different architectures. 50

Chapter 1

Introduction

Low-density parity-check (LDPC) codes was first invented by Gallager in 1962 [1] [2]. Due to the difficulty of circuit implementation and large complexity of calculation, LDPC codes have been forgot for about forty years except for the research of codes defined on graphs by Tanner [3]. The rediscover of LDPC code was done by Mackay in 1995 [4] [5]. It was proven [6] that the LDPC codes with large block length can beat turbo codes, and achieve a capacity within 0.0045dB of the Shannon limit on AWGN channel. With the dramatic improvement of VLSI technology and the robust transmission demands of next communication standards, the research interest of LDPC is dramatically increased recently.

LDPC codes have been adopted by several communication standards, such as IEEE 802.16e standard, IEEE 802.11n standard, and the Digital Video Broadcasting - Satellite - Second Generation (DVB-S2).

The main challenge of the LDPC decoder falls in the complicated interconnections due to the large size of parity check matrix. This leads to large chip area. According to using memory or not, the architecture of LDPC decoder can be divided into two types, one is fully parallel form and the other is partially parallel form.

Fully parallel form directly maps the corresponding Tanner graph into the hardware and all the processing units are connected according to the connectivity of the graph. Thus, they can have very high throughput but have a large hardware cost. The first published LDPC decoder [7] which used fully parallel

form was designed by Blanksby and Howland in 2002. It can achieve 1Gb/s with 64 iterations. However, it also need large area due to the large amount of processing units and the complicated interconnections.

In the other hand, partially parallel form only has part unit of the full processing. Since the processing can be shared by controlling memory. The hardware complexity can be reduced with the plenty of lower throughput rate and more route complexity. In [8], the authors use the decoding unit of Turbo code to increase the throughput, up to 640Mb/s. In [9] an LDPC decoder was proposed for IEEE 802.16e standard, and the design controlling circuit can support 19 modes with rate 1/2. Among the many kinds of the decoders of turbo code, the decoding architecture called Radix-4 turbo decoding architecture [10] calculate two stage of data with one timing cycle. Hence it can reduce memory size and increase the processing speed. In this thesis, we develop a Radix-4 algorithm in LDPC decoder. We call it as Radix-4 LDPC decoding. The Radix-4 decoding can increase the throughput than the conventional decoding. Also, from the simulation results, we observe that the performance of the Radix-4 algorithm is better than conventional algorithm. Furthermore, we implement proposed algorithm via VLSI with application over IEEE 802.11n standard.

The rest of this thesis is organized as follows. Chapter 2 describes the LDPC encoder and decoding algorithm. In addition, we also show the performance of different decoding algorithm. Chapter 3 introduces the proposed algorithm and a method to reduce the decoding latency under IEEE 802.11n standard. The design flow of chip implementation and specification of this work are presented in Chapter 4. Finally, conclusion and future work are given in Chapter 5.

Chapter 2

Low-Density Parity-Check Codes

2.1 Categories of error correction codes

The error correction codes are widely used to improve transmitting quality in modern digital communication systems. The error correction codes also called channel coding are beforehand preserving method adopted to protect transmitted data from injected interference and channel response with noise, as shown in Fig 2.1. In general, error correct codes can be divided into two groups, including block codes and convolution codes, as shown in Fig 2.2. Famous block codes include Hamming codes, cyclic code, and Reed-Solomon code. For the last ten years, Turbo codes in convolutional code and LDPC codes were widely studied due to their significant performance improvement.

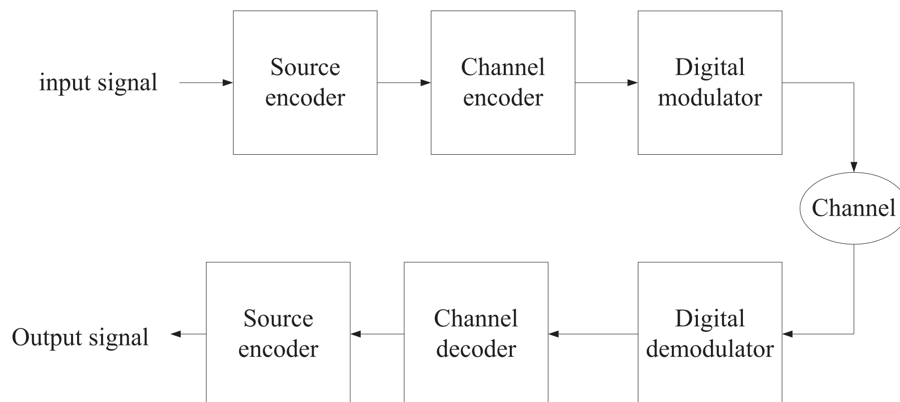


Figure 2.1: A digital communication system.

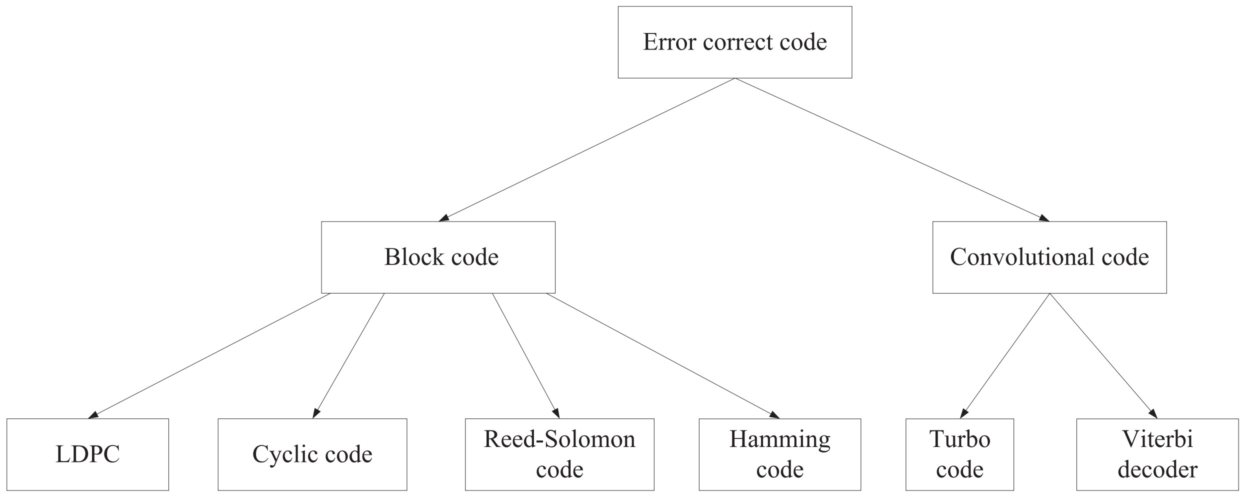


Figure 2.2: Categories of error correction codes.

An (n, k) block code represents that k-bit information data is encoded into an n-bit codeword. Block codes can be further divided into two groups according to the encoded bits in the codeword. One is the systematic block code and the other is the non-systematic block code. Systematic block code consists of original the information bits and redundant bits called parity check bits, as shown in Fig. 2.3. If the block code can not be directly divided into information bits and parity check bits, we call it non-systematic block code.

$$(\underbrace{I_1 \quad I_2 \quad I_3 \quad \cdots \quad I_{k-2} \quad I_{k-1} \quad I_k}_{k \text{ bits}} \quad | \quad \underbrace{P_1 \quad P_2 \cdots \quad P_{n-k-2} \quad P_{n-k-1} \quad P_{n-k}}_{n-k \text{ bits}})$$

Figure 2.3: Codeword in systematic block code.

2.2 Categories of LDPC

Low-density parity-check (LDPC) code is a systematic block code, because the parity check matrix is a sparse matrix, which contains mostly 0's and only a small number of 1's. The parity check matrix $\mathbf{H}$ which has N rows and M columns defines an LDPC code with N information bits and $M - N$ parity bits. The code

rate R in this case is N/M.

An LDPC codes can be divided into two groups according to the number of 1's in one row or one column. If the number of 1's in one row and one column are constant, it is called regular LDPC code. Otherwise, it is called irregular LDPC code. For a regular LDPC code, each column or row has a fixed number of 1's in the parity check matrix **H**. For example, a 6×9 parity check matrix has code rate $R = 2/3$, and can be expressed by Tanner Graph [3] as shown in Fig. 2.4. In general, an irregular LDPC code has better error-correcting performance than regular ones.

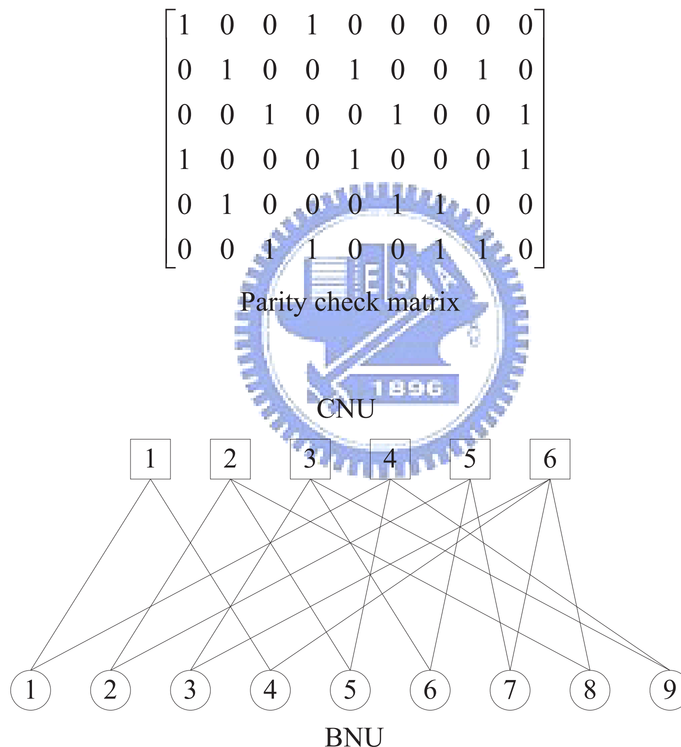


Figure 2.4: Parity check and corresponding Tanner Graph.

An LDPC code can also be classified by it's parity check matrix **H**, e.g. random and quasi-cyclic LDPC code. Quasi-cyclic LDPC codes [11] is a popular code construction in modern communication systems. The code constructions in the Standard like IEEE 802.16e and IEEE 802.11n standards are exactly quasi-

cyclic.

$$\begin{bmatrix}
0 & - & - & - & 0 & 0 & - & - & 0 & - & - & 0 & 1 & 0 & - & - & - & - & - & - & - & - & - & - \\
22 & 0 & - & - & 17 & - & 0 & 0 & 12 & - & - & - & - & 0 & 0 & - & - & - & - & - & - & - & - & - \\
6 & - & 0 & - & 10 & - & - & - & 24 & - & 0 & - & - & - & 0 & 0 & - & - & - & - & - & - & - & - \\
2 & - & - & 0 & 20 & - & - & - & 25 & 0 & - & - & - & - & - & 0 & 0 & - & - & - & - & - & - & - \\
23 & - & - & - & 3 & - & - & - & 0 & - & 9 & 11 & - & - & - & - & 0 & 0 & - & - & - & - & - & - \\
24 & - & 23 & 1 & 17 & - & 3 & - & 10 & - & - & - & - & - & - & - & - & 0 & 0 & - & - & - & - & - \\
26 & - & - & - & 8 & - & - & - & 7 & 18 & - & - & 0 & - & - & - & - & - & 0 & 0 & - & - & - & - \\
13 & 24 & - & - & 0 & - & 8 & - & 6 & - & - & - & - & - & - & - & - & - & - & 0 & 0 & - & - & - \\
7 & 20 & - & 16 & 22 & 10 & - & - & 23 & - & - & - & - & - & - & - & - & - & - & - & 0 & 0 & - & - \\
11 & - & - & - & 19 & - & - & - & 13 & - & 3 & 17 & - & - & - & - & - & - & - & - & - & 0 & 0 & - \\
25 & - & 8 & - & 23 & 18 & - & 14 & 9 & - & - & - & - & - & - & - & - & - & - & - & - & - & 0 & 0 \\
3 & - & - & - & 16 & - & - & 2 & 25 & 5 & - & - & 1 & - & - & - & - & - & - & - & - & - & - & 0
\end{bmatrix}$$

Figure 2.5: Quasi-cyclic LDPC code in 802.11n.

In the quasi-cyclic LDPC code, every element in the parity check matrix is a basic matrix with size $Z \times Z$. There are three types of Z, e.g. 27, 54, and 81 bits in IEEE 802.11n standard. As shown in Fig. 2.5, the element with mark "-" is a zero matrix and the other are cyclic-shifted matrixes. For instance, the identity matrix cyclically shifted right by 1 bit is shown in Fig. 2.6.

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 \end{bmatrix} \xrightarrow{\text{Cyclically shift right by 1 bit}} \begin{bmatrix} 0 & 1 & \cdots & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & \ddots & \ddots & 1 \\ 1 & \dots & 0 & 0 \end{bmatrix}$$

Figure 2.6: A identity matrix shifted right by 1 bit.

2.3 Encoder of LDPC code

In LDPC encoder, a codeword vector $\mathbf{v}$ was produced by multiplying a generator matrix $\mathbf{G}$ multiplied by the information vector $\mathbf{u}$, i.e.

$$\mathbf{uG} = \mathbf{v}. \tag{2.1}$$

Because LDPC code is a linear block code, it satisfies the following equation

$$\mathbf{H}\mathbf{v}^T = 0, \tag{2.2}$$

where $\mathbf{H}$ is the parity check matrix. From (2.2), we can get the relationship of $\mathbf{G}$ and $\mathbf{H}$, as show as

$$\mathbf{H}\mathbf{v}^T = \mathbf{H}(\mathbf{u}\mathbf{G})^T = \mathbf{H}\mathbf{G}^T\mathbf{u}^T = 0 \Longrightarrow \mathbf{H}\mathbf{G}^T = 0. \tag{2.3}$$

Intuitively, the encoding method is letting the generator matrix $\mathbf{G}$ multiply the information vector $\mathbf{u}$. However, it needs to store both $\mathbf{G}$ and $\mathbf{H}$ in circuit. In order to save storing unit, IEEE 802.16e standard introduces a method [12] to encoder directly from the parity check matrix $\mathbf{H}$. Hence, there is no need to use extra memory to store $\mathbf{G}$.

According to [12], an $n \times m$ parity check matrix $\mathbf{H}$ can be divide into six-sub matrices, $\mathbf{A}_{(m-g)\times(n-m)}$, $\mathbf{B}_{(m-g)\times g}$, $\mathbf{T}_{(m-g)\times(m-g)}$, $\mathbf{C}_{g\times(n-m)}$, $\mathbf{D}_{g\times g}$, and $\mathbf{E}_{g\times(m-g)}$ as shown in Fig. 2.7. Among the sub-matrices, there is a constraint which makes LDPC encoding process easier. That is sub-matrix $\mathbf{T}_{(m-g)\times(m-g)}$ must be a lower triangular square matrix with all it's diagonal elements being ones.

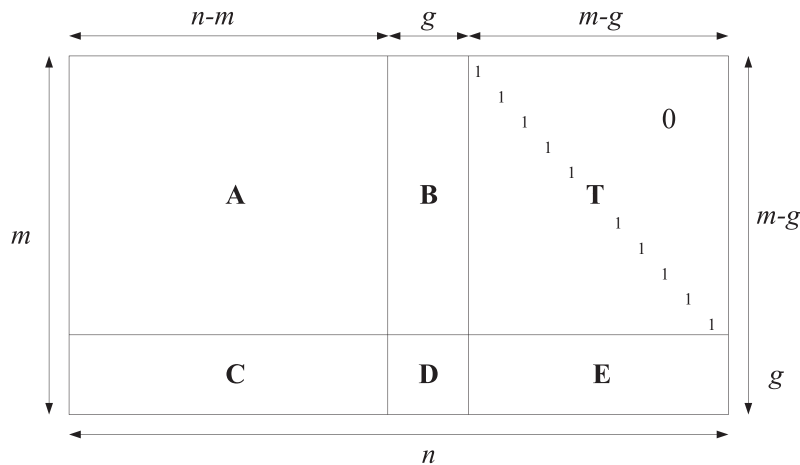


Figure 2.7: Six sub-matrices in parity check matrix $\mathbf{H}$.

Let $\mathbf{v}$ be a codeword corresponding to parity check matrix $\mathbf{H}$. It needs to satisfy the equation (2.2). Therefore, the codeword can be divide into three parts, i.e. $\mathbf{v} = [\mathbf{u}\ \mathbf{p}_1\ \mathbf{p}_2]$. Where $\mathbf{u}$ is the original information bits with length $n-m$. $\mathbf{p}_1$ is with length g and $\mathbf{p}_2$ is with length $m-g$. $\mathbf{p}_1$ and $\mathbf{p}_2$ are the parity check bits.

From the equation (2.2), we can obtain the mathematical relationship be shown in following equations

$$\mathbf{A}\mathbf{u}^T + \mathbf{B}\mathbf{p}_1^T + \mathbf{T}\mathbf{p}_2^T = 0, \tag{2.4}$$

and

$$\mathbf{C}\mathbf{u}^T + \mathbf{D}\mathbf{p}_1^T + \mathbf{E}\mathbf{p}_2^T = 0. \tag{2.5}$$

After rewriting (2.4) and (2.5), we can get

$$(\mathbf{E}\mathbf{T}^{-1}\mathbf{A} + \mathbf{C})\mathbf{u}^T + (\mathbf{E}\mathbf{T}^{-1}\mathbf{B} + \mathbf{D})\mathbf{p}_1^T = 0. \tag{2.6}$$

Define $\phi := \mathbf{E}\mathbf{T}^{-1}\mathbf{B} + \mathbf{D}$ and select the appropriate sub-matrices $\mathbf{B}_{(m-g)\times g}$, $\mathbf{T}_{(m-g)\times(m-g)}$, $\mathbf{D}_{g\times g}$, and $\mathbf{E}_{g\times(m-g)}$ to make the matrix ϕ be the identity matrix. Then we can get

$$\mathbf{p}_1^T = (\mathbf{E}\mathbf{T}^{-1}\mathbf{A} + \mathbf{C})\mathbf{u}^T, \tag{2.7}$$

and

$$\mathbf{p}_2^T = \mathbf{T}^{-1}(\mathbf{A}\mathbf{u}^T + \mathbf{B}\mathbf{p}_1^T). \tag{2.8}$$

As a result, we can produce easily the output vector $\mathbf{v}$, as shown in Fig. 2.8.

2.4 Decoder of LDPC code

In general, the LDPC decoding can divided into hard decision and soft decision. Hard decision means that the message exchange between bit node and check node is a bit, like the bit flipping [2]. On the contrary, soft decision means that the message exchange between bit node and check node is a soft information, like the belief propagation [2] or called Sum-Product algorithm. The performance of

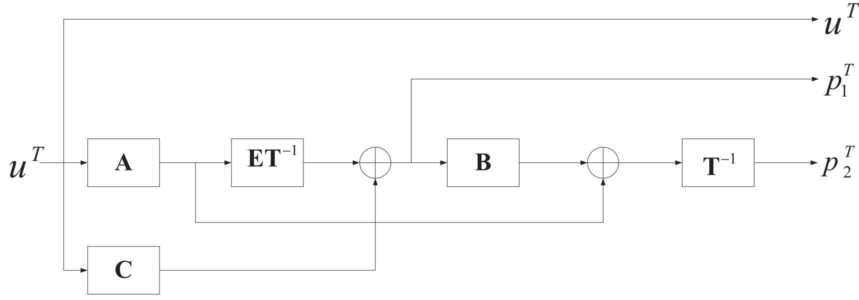


Figure 2.8: Block diagram of the encoder architecture for the block LDPC code.

soft decision is usually better than hard decision however with more complexities. The sections which will be introduced below are based on soft decision.

2.4.1 Sum-Product algorithm(SPA)

By the Tanner Graph as shown in Fig. 2.3 and concept of message passing, we can understand the decoding procedure of LDPC codes more easily. The decoding flow is illustrated in Fig. 2.9.

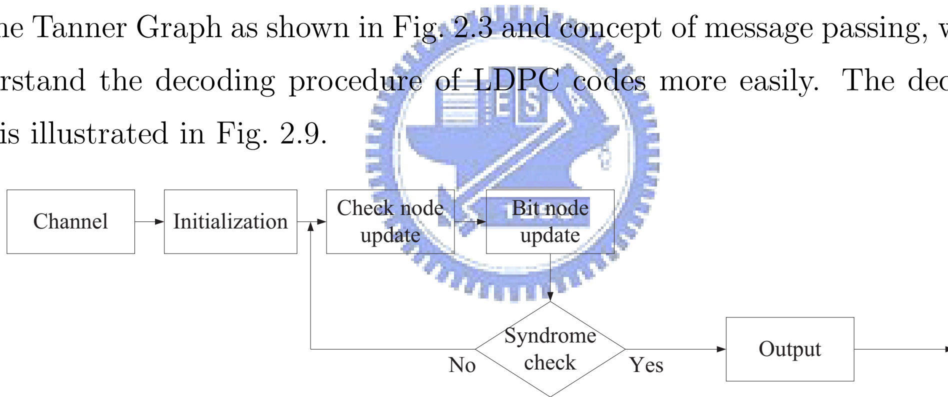


Figure 2.9: A decoding flow of LDPC codes.

For the following mathematical deriving, define that $q_{i\rightarrow j}$ be the probability message from bit node B_i to check node C_j, $r_{j\rightarrow i}$ be the probability message from the check node C_j to the bit node B_i, v_i be the ith bit in the codeword $\mathbf{v}$, y_i be the ith received data, and σ^2 be the noise variance.

(1) Initialization: The probability message $q_{i\rightarrow j}$ is initialized as the received data from channel. Let the transmit data x_i be BPSK, we have

$$q_{i\rightarrow j}(x_i=+1) \quad = \quad P(x_i=+1|y_i)=\frac{P(y_i|x_i=+1)P(x_i=+1)}{P(y_i)}$$

$$= \frac{\frac{1}{2\sigma^2} e^{-\frac{(y_i-1)^2}{2\sigma^2}} \frac{1}{2}}{\frac{1}{2}\frac{1}{2\sigma^2}\left(e^{-\frac{(y_i-1)^2}{2\sigma^2}} + e^{\frac{(y_i+1)^2}{2\sigma^2}}\right)} = \frac{1}{1+e^{-\frac{2y_i}{\sigma^2}}}, \tag{2.9}$$

and

$$q_{i\to j}(x_i = -1) = \frac{e^{-\frac{(2y_i)}{\sigma^2}}}{1+e^{-\frac{2y_i}{\sigma^2}}}. \tag{2.10}$$

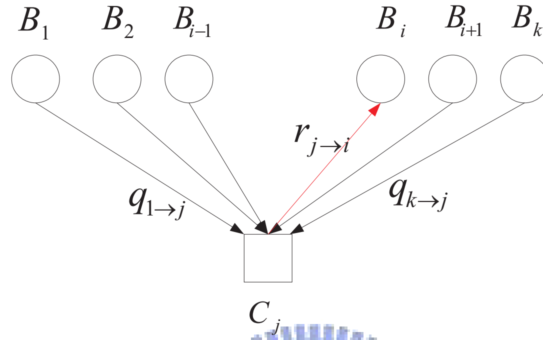


Figure 2.10: Message information form Check node to bit node.

(2) Check node update: In this step, the check nodes perform updating by collecting message from connected bit node. As shown in Fig. 2.10, we will derive the relation between $q_{k\to j}$ and $r_{j\to i}$ below.

From (2.2) and Fig. 2.10, we can obtain

$$B_1 \oplus B_2 \oplus \cdots \oplus B_{i-1} \oplus B_i \oplus B_{i+1} \oplus \cdots \oplus B_k = 0. \tag{2.11}$$

Following (2.11), we can get

$$P(r_{j\to i} = 1) = P(B_1 \oplus B_2 \oplus \cdots \oplus B_{i-1} \oplus B_{i+1} \oplus \cdots \oplus B_k = 1), \tag{2.12}$$

and

$$P(r_{j\to i} = 0) = P(B_1 \oplus B_2 \oplus \cdots \oplus B_{i-1} \oplus B_{i+1} \oplus \cdots \oplus B_k = 0). \tag{2.13}$$

Obtain (2.12) and (2.13). Let $k = 2$, $P(B_1 = 1) = a_1$, and $P(B_2 = 1) = a_2$. (2.12) and (2.13) can be rewritten as

$$P(B_1 \oplus B_2 = 1) = a_1(1-a_2) + (1-a_1)a_2, \tag{2.14}$$

and

$$P(B_1 \oplus B_2 = 0) = a_1 a_2 + (1 - a_1)(1 - a_2). \tag{2.15}$$

(2.14) and (2.15) can be rewritten again as

$$P(B_1 \oplus B_2 = 1) = \frac{1 - (1 - 2a_1)(1 - 2a_2)}{2} = \frac{1 - \prod_{i=1}^{2}(1 - 2a_i)}{2}, \tag{2.16}$$

and

$$P(B_1 \oplus B_2 = 0) = \frac{1 + (1 - 2a_1)(1 - 2a_2)}{2} = \frac{1 + \prod_{i=1}^{2}(1 - 2a_i)}{2}. \tag{2.17}$$

Assume (2.12) and (2.13) are true for the condition that $k = n$, we can obtain

$$P(B_1 \oplus B_2 \oplus \cdots \oplus B_n = 1) = \frac{1 - \prod_{i=1}^{n}(1 - 2a_i)}{2} \triangleq M_n, \tag{2.18}$$

and

$$P(B_1 \oplus B_2 \oplus \cdots \oplus B_n = 0) = \frac{1 + \prod_{i=1}^{n}(1 - 2a_i)}{2} = 1 - M_n. \tag{2.19}$$

Let $P(B_{n+1} = 1) = a_{n+1}$ on condition that $k = n + 1$, (2.16) and (2.17) can be deduced as

$$\begin{aligned} P(B_1 \oplus B_2 \oplus \cdots \oplus B_n \oplus B_{n+1} = 1) &= M_n(1 - a_{n+1}) + (1 - M_n)a_{n+1} \\ &= \frac{1 - (1 - 2a_{n+1})(1 - 2M_n)}{2} \\ &= \frac{1 - \prod_{i=1}^{n+1}(1 - 2a_i)}{2}, \end{aligned} \tag{2.20}$$

and

$$\begin{aligned} P(B_1 \oplus B_2 \oplus \cdots \oplus B_n \oplus B_{n+1} = 0) &= M_n a_{n+1} + (1 - M_n)(1 - a_{n+1}) \\ &= \frac{1 + (1 - 2a_{n+1})(1 - 2M_n)}{2} \\ &= \frac{1 + \prod_{i=1}^{n+1}(1 - 2a_i)}{2}. \end{aligned} \tag{2.21}$$

By Mathematical Induction, we can get

$$P(r_{j\to i}=1) = P(B_1 \oplus B_2 \oplus \cdots \oplus B_{i-1} \oplus B_{i+1} \oplus \cdots \oplus B_k = 1) = \frac{1-\prod_{i'\in W(j)\backslash\{i\}}(1-2Q_{i'\to j})}{2}, \tag{2.22}$$

and

$$P(r_{j\to i}=0) = P(B_1 \oplus B_2 \oplus \cdots \oplus B_{i-1} \oplus B_{i+1} \oplus \cdots \oplus B_k = 0) = \frac{1+\prod_{i'\in W(j)\backslash\{i\}}(1-2Q_{i'\to j})}{2}, \tag{2.23}$$

where $Q_{i'\to j} \triangleq P(q_{i'\to j}=1)$, $W(j)$ is the set of bit nodes connected to the check node C_j, and $W(j)\backslash\{i\}$means the subset excluding the ith bit node.

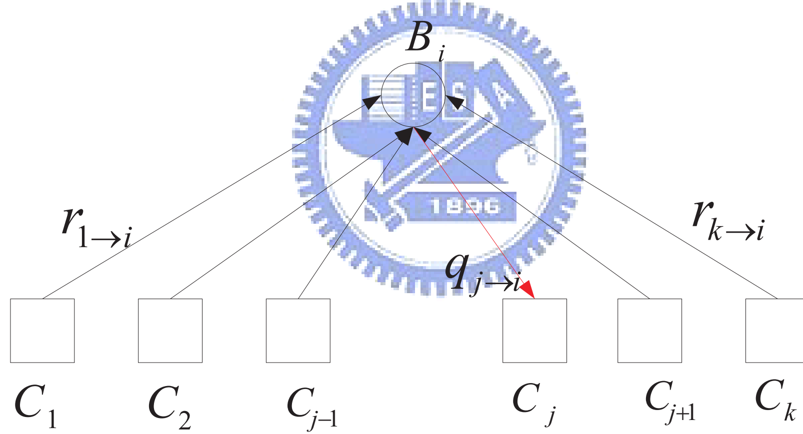


Figure 2.11: Message information form bit node to check node.

(3) Bit node update: Similarly, the bit nodes perform updating by collecting message from the connected check nodes. Next let us derive the relation between $q_{j\to i}$ and $r_{k\to i}$ below.

As shown in Fig. 2.11, there are k check nodes connected bit node B_i. Thus we can get

$$P(q_{j\to i}=1) = P(v_i=1)\prod_{j'\in M(i)\backslash\{j\}} P(r_{i\to j'}=1), \tag{2.24}$$

and

$$P(q_{j\to i}=0)=P(v_i=0)\prod_{j'\in M(i)\setminus\{j\}}P(r_{i\to j'}=0). \tag{2.25}$$

where $M(i)$ is the set of check nodes connected to the bit node B_i, and $M(i)\setminus\{j\}$ means the subset excluding the jth check node.

(4) Syndrome check: Let

$$P(v_i=1)=P(x_i=+1|y_i),$$

and

$$P(v_i=0)=P(x_i=-1|y_i).$$

The a posteriori probability for each codeword bit can be computed as

$$P^{post}(v_i=1)=P(v_i=1)\prod_{j\in M(i)}P(r_{i\to j}=1). \tag{2.26}$$

and

$$P^{post}(v_i=0)=P(v_i=0)\prod_{j\in M(i)}P(r_{i\to j}=0), \tag{2.27}$$

The estimated bit $\hat{v}_i$ is set to 1 if $P^{post}(v_i=1)>0.5$, otherwise it is set to 0. Then the syndrome check equation (2.2) is used to verify whether the estimated sequence $\hat{\mathbf{v}}=[\hat{v}_1\ \hat{v}_2\ \cdots\ \hat{v}_n]$ is a valid codeword.

The decoding process stops when the syndrome check equation is satisfied, otherwise the decoding iteration will start. If the number of iterations attains our predetermined goal without finding a right codeword, it fails to decode and still pass to be the valid codeword. This is the reason which make bit error.

2.4.2 Log-Likelihood Ratio Sum-Product algorithm (LLR)

In order to reduce decoding complexity, the decoding operations can be performed in term of log-likelihood ratios [13]. The log-likelihood ratio (LLR) of a binary random variable U which can be defined as

$$L(U)=\log\frac{P(U=0)}{P(U=1)}, \tag{2.28}$$

where $P(U = 0)$ and $P(U = 1)$ mean the probability of a binary random variable U being 0 and 1 respectively. Therefore, the decoding flow can be modified as follows.

(1) Initialization: By combining (2.9), (2.10), and (2.28), the initial message can be modified as

$$L_{v_i} = \log \frac{P(v_i = 0)}{P(v_i = 1)} = \log \frac{\frac{e^{-\frac{(2y_i)}{\sigma^2}}}{1+e^{-\frac{2y_i}{\sigma^2}}}}{\frac{1}{1+e^{-\frac{2y_i}{\sigma^2}}}} = \log \left(e^{\frac{-2y_i}{\sigma^2}} \right) = \frac{-2y_i}{\sigma^2}, \tag{2.29}$$

where y_i be the received data and σ^2 be the noise variance.

(2) Check node update: By combining (2.22), (2.23), and (2.28), we can obtain

$$L_{r_{j\to i}} = \log \frac{P(r_{j\to i} = 0)}{P(r_{j\to i} = 1)} = \log \left(\frac{1 + \prod_{i' \in W(j)\backslash\{i\}} (1 - 2Q_{i'\to j})}{1 - \prod_{i' \in W(j)\backslash\{i\}} (1 - 2Q_{i'\to j})} \right). \tag{2.30}$$

By using the hyperbolic tangent function

$$\tanh\left(\frac{x}{2}\right) = \frac{e^x - 1}{e^x + 1}, \tag{2.31}$$

and the arc-hyperbolic tangent function

$$\tanh^{-1}(y) = \frac{1}{2} \log \frac{1+y}{1-y}, \tag{2.32}$$

Eq (2.30) can be rewritten as follows,

$$\begin{aligned}
L_{r_{j\to i}} &= \log \frac{1 + \prod_{i' \in W(j)\backslash\{i\}} (1 - 2Q_{i'\to j})}{1 - \prod_{i' \in W(j)\backslash\{i\}} (1 - 2Q_{i'\to j})} = \log \frac{1 + \prod_{i' \in W(j)\backslash\{j\}} \left(1 - 2\frac{1}{1 + e^{-\frac{2y_{i'}}{\sigma^2}}}\right)}{1 - \prod_{i' \in W(j)\backslash\{j\}} \left(1 - 2\frac{1}{1 + e^{-\frac{2y_{i'}}{\sigma^2}}}\right)} \\
&= \log \frac{1 + \prod_{i' \in W(j)\backslash\{j\}} \left(\frac{e^{-\frac{2y_{i'}}{\sigma^2}} - 1}{e^{-\frac{2y_{i'}}{\sigma^2}} + 1} \right)}{1 - \prod_{i' \in W(j)\backslash\{j\}} \left(\frac{e^{-\frac{2y_{i'}}{\sigma^2}} - 1}{e^{-\frac{2y_{i'}}{\sigma^2}} + 1} \right)} = \log \frac{1 + \prod_{i' \in W(j)\backslash\{j\}} \tanh\left(\frac{-y_{i'}}{\sigma^2}\right)}{1 - \prod_{i' \in W(j)\backslash\{j\}} \tanh\left(\frac{-y_{i'}}{\sigma^2}\right)} \\
&= 2\tanh^{-1}\left(\prod_{i' \in W(j)\backslash\{j\}} \tanh\left(\frac{1}{2} L_{q_{i'\to j}}\right) \right).
\end{aligned} \tag{2.33}$$

(3) Bit node update: By combining (2.24), (2.25), and (2.28), we can obtain

$$L_{q_{i\to j}} = \log \frac{P(v_i=0)\prod_{j'\in M(i)\setminus\{j\}} P(r_{j'\to i}=0)}{P(v_i=1)\prod_{j'\in M(i)\setminus\{j\}} P(r_{j'\to i}=1)} = L_{v_i} + \sum_{j'\in M(i)\setminus j} L_{r_{j'\to i}}. \quad (2.34)$$

(4) Syndrome check: The final posteriori probability of deciding 0 or 1 is

$$L_{v_i}^{post} = \log \frac{P^{post}(v_i=0)}{P^{post}(v_i=1)} = L_{v_i} + \sum_{j\in M(i)} L_{r_{i\to j}}. \quad (2.35)$$

Deciding the result is 0 or 1 according $L_{v_i}^{post}$ by

$$\begin{cases} \hat{v}_i \Rightarrow 0, \text{if } L_{v_i}^{post} \geq 0,\ \forall i \in \{1,2,\cdots\cdots,n\} \\ \hat{v}_i \Rightarrow 1, \text{if } L_{v_i}^{post} < 0,\ \forall i \in \{1,2,\cdots\cdots,n\}. \end{cases} \quad (2.36)$$

If (2.2) is satisfied or arrival the number of setting iteration, the decoding process is finished. Otherwise it goes into the next iteration.

2.4.3 Min-Sum with a correct factor (Min-Sum-Correct)

By observing the flow of the above decoding algorithm, we can find the computational complexity was most on the check node update due to the hyperbolic tangent functions. Based on the calculation reducing or rewriting, there are some development of derivative algorithm like Min-Sum-Correct [14], Min-Sum [15], and normalized Min-Sum algorithm [16].

We consider a check node with 3 bit nodes, as shown in Fig. 2.12. By combining (2.22), (2.23), and (2.28), we can obtain

$$\begin{aligned} L_{r_{1\to 3}} &= \log \frac{1 + (\frac{e^{L_{q_{1\to 1}}}-1}{e^{L_{q_{1\to 1}}}+1} \cdot \frac{e^{L_{q_{2\to 1}}}-1}{e^{L_{q_{2\to 1}}}+1})}{1 - (\frac{e^{L_{q_{1\to 1}}}-1}{e^{L_{q_{1\to 1}}}+1} \cdot \frac{e^{L_{q_{2\to 1}}}-1}{e^{L_{q_{2\to 1}}}+1})} \\ &= \log(\frac{1 + e^{L_{q_{1\to 1}}} e^{L_{q_{2\to 1}}}}{e^{L_{q_{1\to 1}}} + e^{L_{q_{2\to 1}}}}) \\ &= \log(1 + e^{L_{q_{1\to 1}} + L_{q_{2\to 1}}}) - \log(e^{L_{q_{1\to 1}}} + e^{Lq_{2\to 1}}) \\ &= \max(0, L_{q_{1\to 1}} + L_{q_{2\to 1}}) + \log(1 + e^{-|L_{q_{1\to 1}} + L_{q_{2\to 1}}|}) \\ &\quad - \max(L_{q_{1\to 1}}, L_{q_{2\to 1}}) - \log(1 + e^{-|L_{q_{1\to 1}} - L_{q_{2\to 1}}|}) \\ &= \text{sign}(L_{q_{1\to 1}})\text{sign}(L_{q_{2\to 1}}) \min(|L_{q_{1\to 1}}|, |L_{q_{2\to 1}}|) + g(L_{q_{1\to 1}}, L_{q_{2\to 1}}), \end{aligned} \quad (2.37)$$

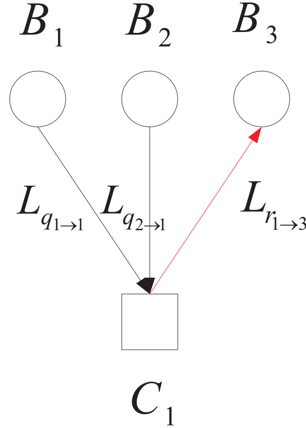


Figure 2.12: Message information from bit node to check node.

where $g(L_{q_{1\to1}}, L_{q_{2\to1}}) = \log(1 + e^{-|L_{q_{1\to1}}+L_{q_{2\to1}}|}) - \log(1 + e^{-|L_{q_{1\to1}}-L_{q_{2\to1}}|})$ is a correct term.

We can obtain the general check nod updates flow as shown in Fig. 2.13, a check node is connected to many bit nodes. The notation $\oslash$ is the computing unit which execute the formula in (2.37).

The only different between Min-Sum-Correct algorithm and LLR algorithm is the mathematical representation in step of check node update. Therefore the Min-Sum-Correct algorithm flow can be obtained by replace equation (2.37) with (2.33). In general, most research in LDPC decoding is on how to achieve good performance or low complex, like Min-Sum algorithm that we will introduce below.

2.4.4 Min-Sum algorithm

The Min-Sum algorithm is derived from Min-Sum-Correct algorithm. By following (2.34), if the term $g(L_{q_{1\to1}}, L_{q_{2\to1}})$ is skipped, we can obtain

$$L_{r_{1\to3}} = \text{sign}(L_{q_{1\to1}})\text{sign}(L_{q_{2\to1}}) \min(|L_{q_{1\to1}}|, |L_{q_{2\to1}}|). \quad (2.38)$$

From (2.38), a sub-optimal expression for general case can be obtained as follows

$$L_{r_{j\to i}} \approx (\prod_{B_{i'}\in W(j)\setminus B_i} \text{sign}(L_{q_{i'\to j}})) \min_{B_{i'}\in W(j)\setminus B_i} (|L_{q_{i'\to j}}|). \quad (2.39)$$

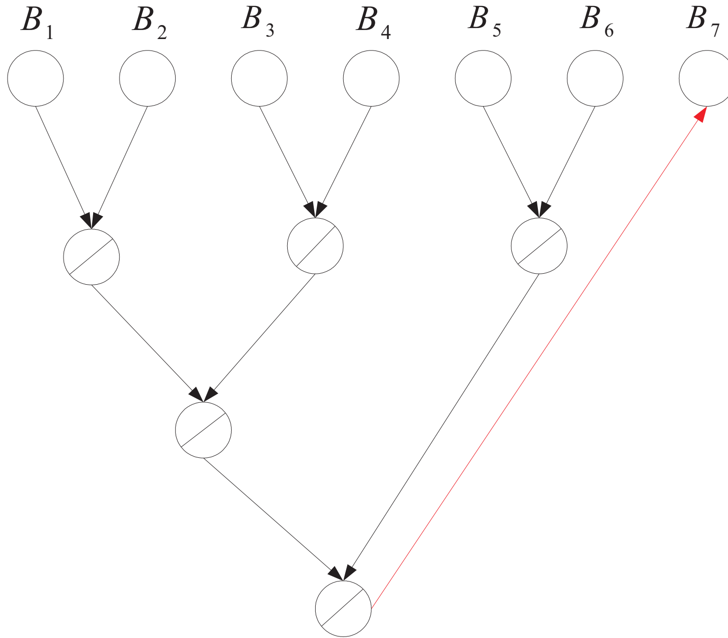


Figure 2.13: The general flow of check node update.

By skipping the correct term, there is a penalty of performance degradation. Later we will show this by simulation result. Like Min-Sum-Correct algorithm, the Min-Sum algorithm is same with LLR algorithm except check node update.

2.4.5 Simulation result

In this subsection, we will show the simulation that the relationship is between BER (bit error rate) and SNR (signal to noise ratio) by algorithm above mentioned.

In Fig. 2.14, there are performances of LLR algorithm with different iterations. We can find that the better performance can achieved by the more iterations. Similarly, Fig. 2.15 and Fig. 2.16 show the simulation results with different iterations by Min-Sum-Correct algorithm and Min-Sum algorithm respectively. By the way, Fig. 2.17 and Fig. 2.18 mean the performance comparison with one iteration and ten iterations respectively. We can find the curve of LLR algorithm and the curve of Min-Sum-Correct algorithm are almost overlapped, the reason

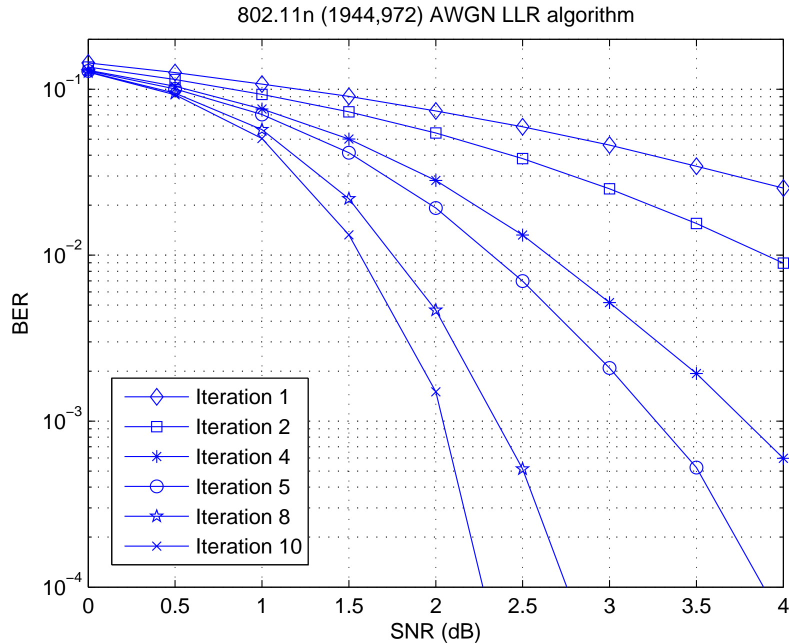


Figure 2.14: LLR algorithm in 802.11n in AWGN channel (Z=81bits).

is that their calculation of check node update are same in fact.

In addition, we find in Fig. 2.17 that with one iteration the performance with Min-Sum is better than that with LLR algorithm. [16] and [20] mentioned that the value of the check node update in Min-Sum algorithm is larger than it in LLR algorithm, where larger values implies better reliability. However the above phenomenon only appear at the 1st iteration as shown in Fig. 2.19. The reliability of the Min-Sum becomes worse as the iteration number grows as shown in Figs. 2.20- 2.22, where the SNR is 4. Hence, with a reasonable large iteration number the performance of the LLR outperforms that of Min-sum.

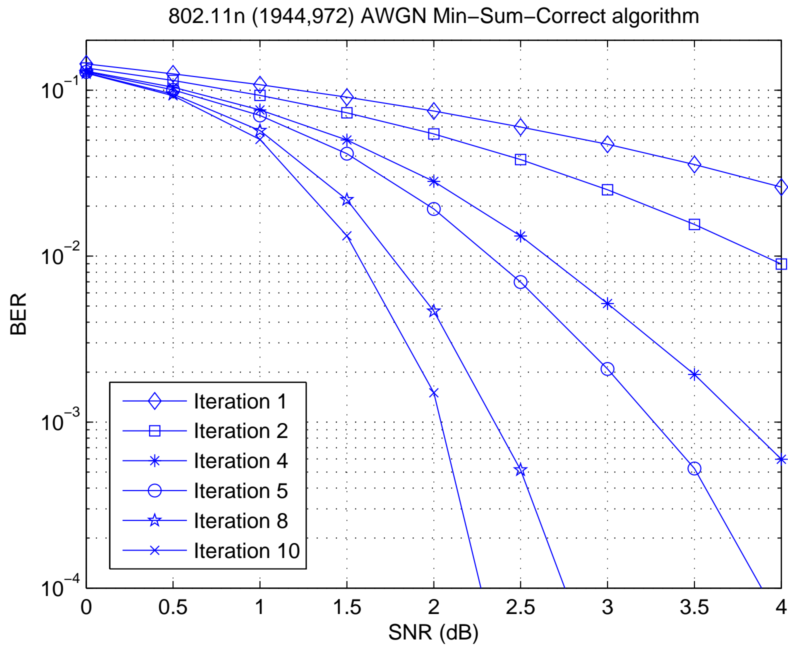


Figure 2.15: Min-Sum-Correct algorithm in 802.11n in AWGN channel (Z=81bits).

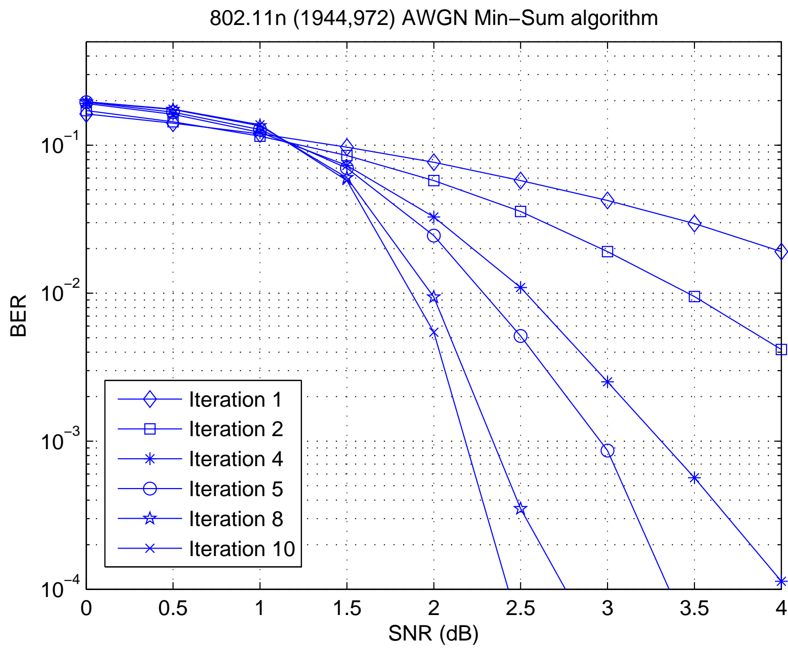


Figure 2.16: Min-Sum algorithm in 802.11n in AWGN channel (Z=81bits).

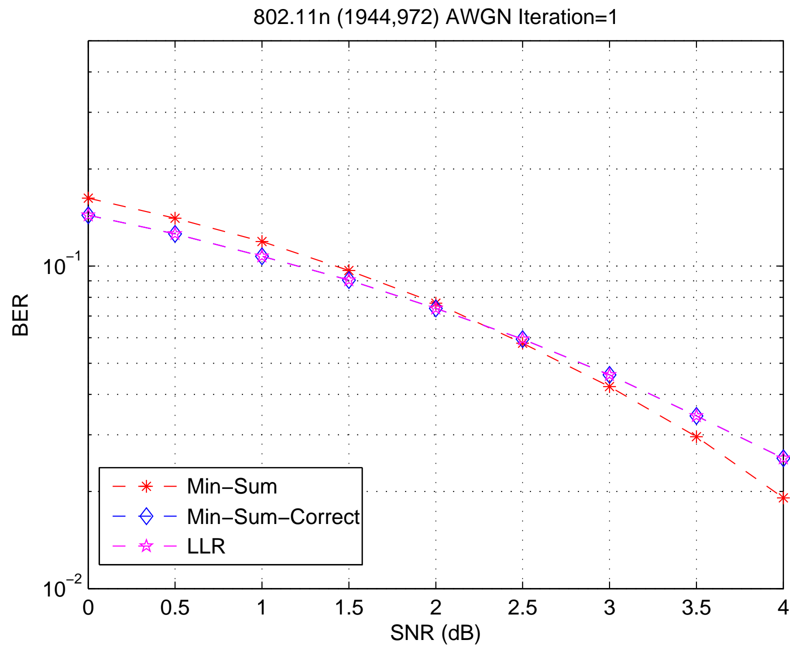


Figure 2.17: Comparison of different algorithm with one iteration.

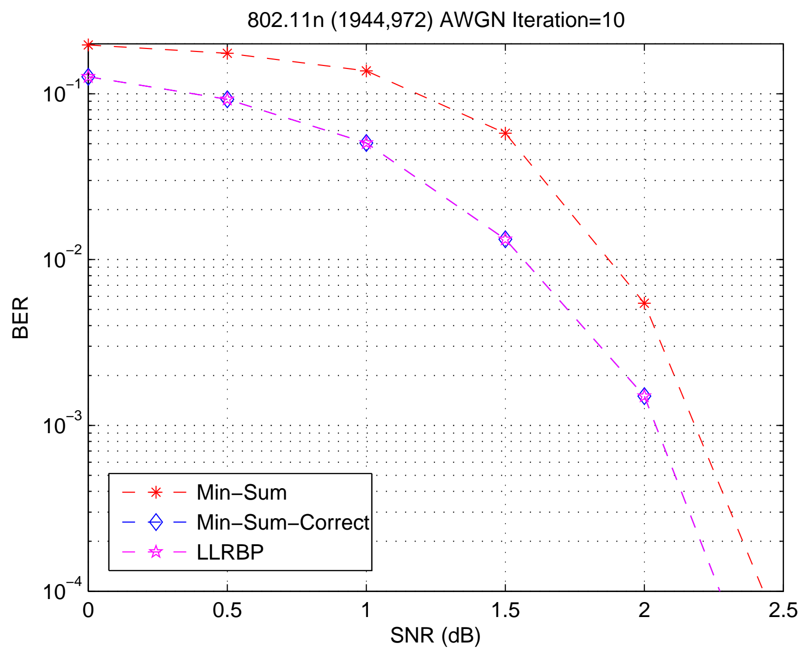


Figure 2.18: Comparison of different algorithm with ten iterations.

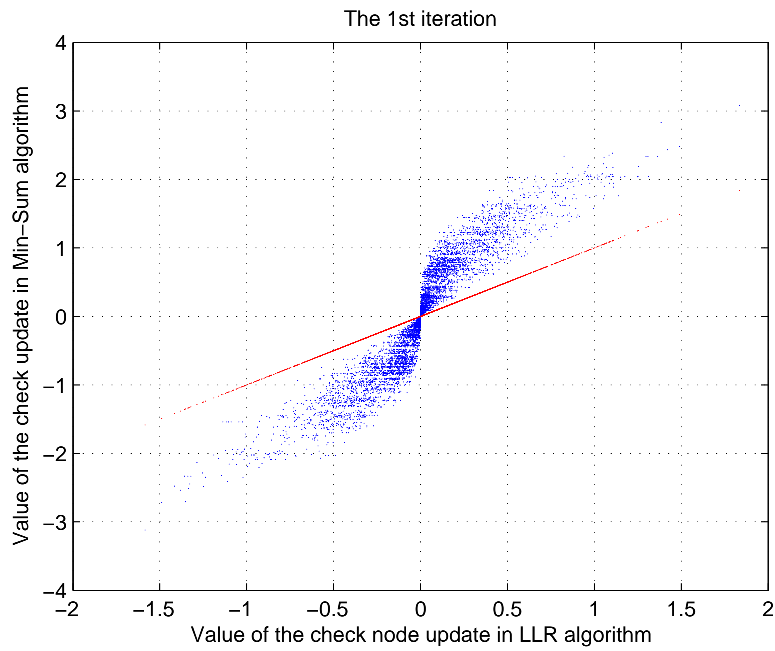


Figure 2.19: The value of check node update in different algorithm.

ES

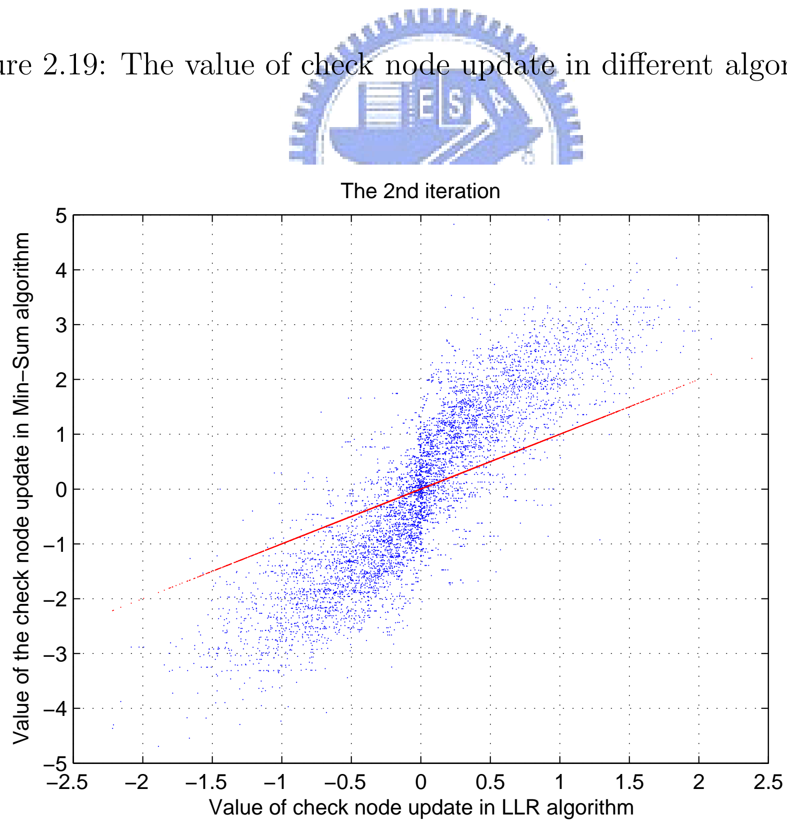


Figure 2.20: The value of check node update in different algorithm.

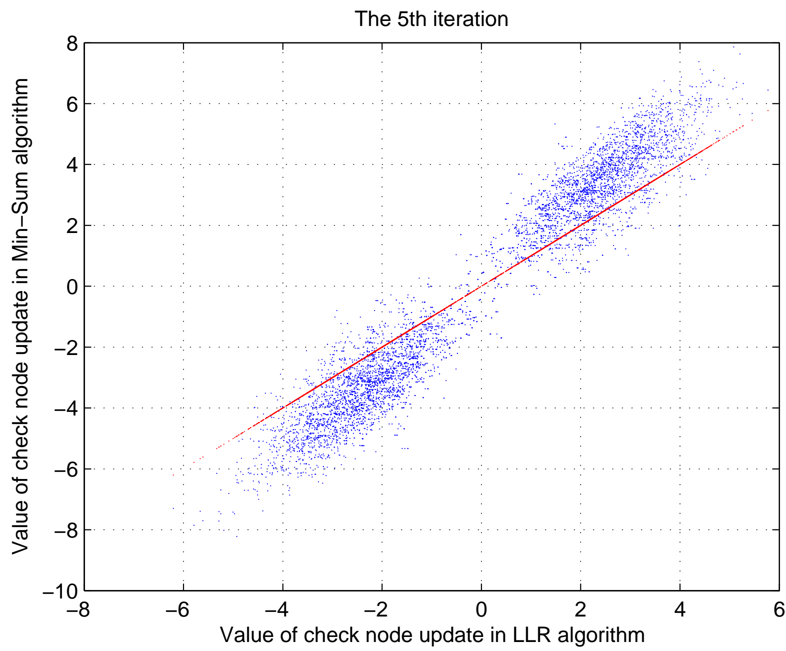


Figure 2.21: The value of check node update in different algorithm.

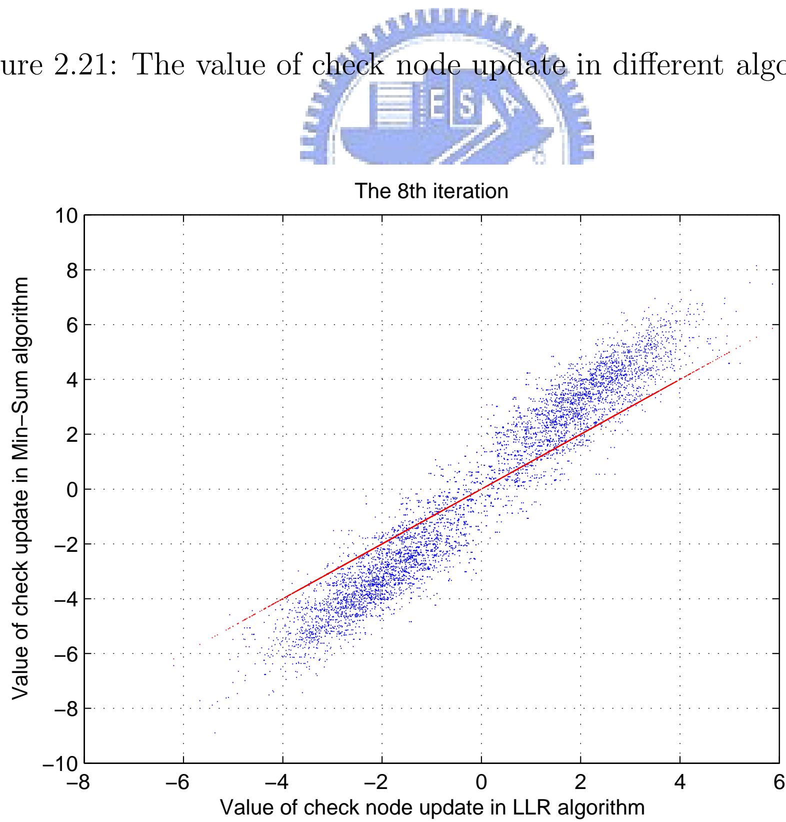


Figure 2.22: The value of check node update in different algorithm.

Chapter 3

Proposed Algorithm and Architecture

3.1 Latency reduction

In general, the LDPC decoding time increases as the size of parity check matrix increase. However, the size of parity check matrix size is usually large in standard. Thus we use a method consisting of two steps to reduce the decoding time introduced in the following subsections.

3.1.1 Reordering of the parity check matrix

In LDPC decoding process, the bit node update does not start until that the check node update completes. If the size of the parity check matrix is large, e.g. IEEE 802.11n standard, the decoding time is also large. To save overall decoding time, a easy way was proposed in [17] to reduce the decoding time by reordering the parity check matrix. Reordering the rows and the columns in the parity matrix does not change the decoding process as mentioned in Chapter 2 and does not affect the BER performance. Next let us explain the reason. From Fig. 3.1, the parity check equation (2.2) is still satisfied in case (b), so we can get same decoding result from the original parity check matrix and reordered parity check matrix. We can obtain that the row reordering has no any change on the decoding process. In another hand, the columns reordering just affect the codeword sequence. From Fig. 3.2, the codeword sequence in case (b) is changed

to satisfy the parity check equation (2.2). Hence overall decoding process does not change.

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = 0 \qquad \begin{bmatrix} d_1 & d_2 & d_3 & d_4 \\ a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = 0$$

(a) (b)

Figure 3.1: Reordering the row of the parity check matrix: (a) original matrix, and (b) reordered matrix.

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = 0 \qquad \begin{bmatrix} a_3 & a_4 & a_2 & a_1 \\ b_3 & b_4 & b_2 & b_1 \\ c_3 & c_4 & c_2 & c_1 \\ d_3 & d_4 & d_2 & d_1 \end{bmatrix}\begin{bmatrix} v_3 \\ v_4 \\ v_2 \\ v_1 \end{bmatrix} = 0$$

(a) (b)

Figure 3.2: Reordering the column of the parity check matrix: (a) original matrix, and (b) reordered matrix.

By combining both the row and the column reordering, we only need to consider the effect of column reordering. From Fig. 3.3, the relationship between the original codeword and the reordered codeword is just a element permutation according to column permutation of the parity check matrix.

The main advantage of the reordering is that the idle time can be reduced by performing overlapped operation between check node update and bit node update. For example, by using the original parity check matrix **H** shown in Fig. 3.4. If we want to perform bit node update, we have to wait until the last row of the matrix being calculated, and than we can start performing bit node update. Because there is a necessary information (red part) for the bit node update of the 1st column in the last row. On the contrary, by using the

$$
\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix}
\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = 0
\qquad
\begin{bmatrix} d_3 & d_4 & d_2 & d_1 \\ c_3 & c_4 & c_2 & c_1 \\ a_3 & a_4 & a_2 & a_1 \\ b_3 & b_4 & b_2 & b_1 \end{bmatrix}
\begin{bmatrix} v_3 \\ v_4 \\ v_2 \\ v_1 \end{bmatrix} = 0
$$

(a) (b)

Figure 3.3: Reordering the parity check matrix (a)original matrix, and (b)reordered matrix.

reordered parity check matrix as shown in Fig. 3.5, we only need to wait until the 8th row to finish, we can start bit node update. On the other hand, it is same that the performing check node update before bit node updating. If we want to perform check node update in the 4th row, we have to wait until the 17th column of the matrix being calculated, and than we can start performing check node update. Because there is a necessary information (green part) for the check node update of the 4th row in the last column. General speaking, reordering the parity matrix can increase decoding speed without degrading performance and adding extra hardware complexity.

$$
\begin{bmatrix}
57 & - & - & - & 50 & - & 11 & - & 50 & - & 79 & - & 1 & 0 & - & - & - & - & - & - & - & - & - & - \\
3 & - & 28 & - & 0 & - & - & - & 55 & 7 & - & - & - & 0 & 0 & - & - & - & - & - & - & - & - & - \\
30 & - & - & - & 24 & 37 & - & - & 56 & 14 & - & - & - & - & 0 & 0 & - & - & - & - & - & - & - & - \\
62 & 53 & - & - & 53 & - & - & 3 & 35 & - & - & - & - & - & - & 0 & 0 & - & - & - & - & - & - & - \\
40 & - & - & 20 & 66 & - & - & 22 & 28 & - & - & - & - & - & - & - & 0 & 0 & - & - & - & - & - & - \\
0 & - & - & - & 8 & - & 42 & - & 50 & - & - & 8 & - & - & - & - & - & 0 & 0 & - & - & - & - & - \\
69 & 79 & 79 & - & - & - & 56 & - & 52 & - & - & - & 0 & - & - & - & - & - & 0 & 0 & - & - & - & - \\
65 & - & - & - & 38 & 57 & - & - & 72 & - & 27 & - & - & - & - & - & - & - & - & 0 & 0 & - & - & - \\
64 & - & - & - & 14 & 52 & - & - & 30 & - & - & 32 & - & - & - & - & - & - & - & - & 0 & 0 & - & - \\
- & 45 & - & 70 & 0 & - & - & - & 77 & 9 & - & - & - & - & - & - & - & - & - & - & - & 0 & 0 & - \\
2 & 56 & - & 57 & 35 & - & - & - & - & - & 12 & - & - & - & - & - & - & - & - & - & - & - & 0 & 0 \\
24 & - & 61 & - & 60 & - & - & 27 & 51 & - & - & 16 & 1 & - & - & - & - & - & - & - & - & - & - & 0
\end{bmatrix}
$$

Figure 3.4: The original parity check matrix of IEEE 802.11n standard.

	NEW	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
NEW	*OLD*	7	11	13	12	24	23	22	21	5	1	9	2	10	6	4	14	3	8	20	19	18	17	16	15
1	1	11	79	1	–	–	–	–	–	50	57	50	–	–	–	–	0	–	–	–	–	–	–	–	–
2	9	–	–	–	32	–	–	0	0	14	64	30	–	–	52	–	–	–	–	–	–	–	–	–	–
3	11	–	12	–	–	0	0	–	–	35	2	–	56	–	–	57	–	–	–	–	–	–	–	–	–
4	10	–	–	–	–	–	0	0	–	0	–	77	45	9	–	70	–	–	–	–	–	–	–	–	–
5	12	–	–	1	16	0	–	–	–	60	24	51	–	–	–	–	–	61	27	–	–	–	–	–	–
6	8	–	27	–	–	–	–	0	–	38	65	72	–	–	57	–	–	–	–	0	–	–	–	–	–
7	7	56	–	0	–	–	–	–	–	–	69	52	79	–	–	–	–	79	–	0	0	–	–	–	–
8	6	42	–	–	8	–	–	–	–	8	0	50	–	–	–	–	–	–	–	–	0	0	–	–	–
9	5	–	–	–	–	–	–	–	–	66	40	28	–	–	–	20	–	–	22	–	–	0	0	–	–
10	4	–	–	–	–	–	–	–	–	53	62	35	53	–	–	–	–	–	3	–	–	–	0	0	–
11	3	–	–	–	–	–	–	–	–	24	30	56	–	14	37	–	–	–	–	–	–	–	–	0	0
12	2	–	–	–	–	–	–	–	–	0	3	55	–	7	–	–	0	28	–	–	–	–	–	–	0

Figure 3.5: The reordered parity check matrix of IEEE 802.11n standard.

3.1.2 Overlapped operation of bit node update and check node update

In general, the LDPC decoder can be divided into two groups, including fully-parallel and partially-parallel architectures. The fully-parallel form can achieve large throughput however with large hardware. On the contrary, the partially-parallel form has lower throughput and large decoding latency but with lower complexity. In addition, the partially-parallel form can perform overlapped operations of bit node update and check node update when we reorder the parity check matrix.

In order to maximize the period of the overlapping, the reordered parity check matrix in Fig. 3.5 can be redrawn as Fig. 3.6, where the BNU is the unit for the bit node update, and CNU is the unit for the check node update. There are eight BNUs and four CNUs in the proposed architecture. Let us introduce the timing schedule of the decoding process. Assume we use a subblock size Z 81×81. Fours rows are completed in one clock cycle in the step of check node update. It needs 81 clock cycles to complete the check node update from the 1st row to the 4th row in Fig 3.6. The same clock cycles are needed for check node update from the 5th row to the 8th row, and from the 9th row to the 12th row. Hence it needs 243 clock cycles to complete one check node update. In addition, it

	BNU 1								BNU 2								BNU 3							
CNU 1	11	79	1	–	–	–	–	–	50	57	50	–	–	–	–	0	–	–	–	–	–	–	–	–
	–	–	–	32	–	–	0	0	14	64	30	–	–	52	–	–	–	–	–	–	–	–	–	–
	–	12	–	–	0	0	–	–	35	2	–	56	–	–	57	–	–	–	–	–	–	–	–	–
	–	–	–	–	–	0	0	–	0	–	77	45	9	–	70	–	–	–	–	–	–	–	–	–
CNU 2	–	–	1	16	0	–	–	–	60	24	51	–	–	–	–	–	61	27	–	–	–	–	–	–
	–	27	–	–	–	–	0	–	38	65	72	–	–	57	–	–	–	–	0	–	–	–	–	–
	56	–	0	–	–	–	–	–	–	69	52	79	–	–	–	–	79	–	0	0	–	–	–	–
	42	–	–	8	–	–	–	–	8	0	50	–	–	–	–	–	–	–	–	0	0	–	–	–
CNU 3	–	–	–	–	–	–	–	–	66	40	28	–	–	–	20	–	–	22	–	–	0	0	–	–
	–	–	–	–	–	–	–	–	53	62	35	53	–	–	–	–	–	3	–	–	–	0	0	–
	–	–	–	–	–	–	–	–	24	30	56	–	14	37	–	–	–	–	–	–	–	–	0	0
	–	–	–	–	–	–	–	–	0	3	55	–	7	–	–	0	28	–	–	–	–	–	–	0

Figure 3.6: The reordered parity check matrix of IEEE 802.11n standard.

also needs 81 clock cycles to complete the bit node update form the 1st column to the 4th column. The same clock cycles are needed for bit node update from the 5th column to the 8th column and from 9th column to the 12th column. Totally it also needs 243 clock cycle to complete one bit node update. Because the parity matrix is reordered, we can make the lower left and upper-right parts of the parity check matrix be zero matrices as shown in Fig. 3.5. In this case, the bit node update from the 1st column to the 8th column can be performed easier right the after check note update of the first 8 rows. There is null information between the 9th row and the 12th row for performing bit node update from the 1st column to the 8th column; Similarly, the check node update from the 1st row to the 4th row can be performed easier right the after the bit node update of the first 16 columns. There are null information between the 17th column and the 24th column for check node update from the 1st row to the 4th row. For example, it needs $81 \times 12 = 972$ clock cycles for original decoding flow 2 iteration. On the other hand, it just needs $81 \times 9 = 729$ clock cycles for overlapped decoding flow with 2 iterations, as shown in Fig. 3.7.

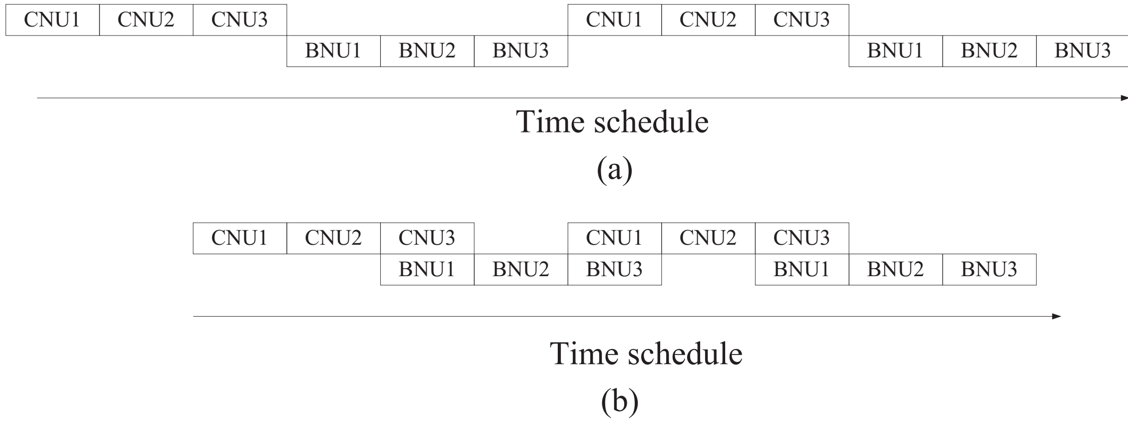


Figure 3.7: Timing diagram: (a) original (b) overlapped.

3.2 Proposed algorithm

In Chapter 2, we introduced the commonly used LDPC decoding algorithms and showed the simulation result of the corresponding performance. In this section, we will introduce the Radix-4 algorithm.

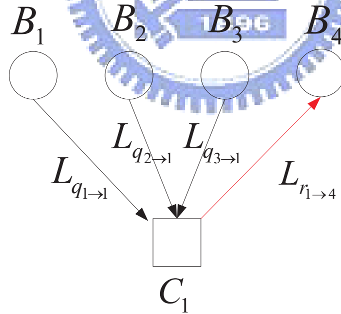


Figure 3.8: Message information from bit nodes to a check node.

Similar to the derivation of Min-Sum-Correct algorithm, first we consider a check node with 4 bit nodes, as shown in Fig. 3.8. By combining (2.22), (2.23), and (2.28), we can obtain

$$L_{r_{1\to 4}} \quad = \quad \log \frac{1 + (\frac{e^{L_{q1\to 1}}-1}{e^{L_{q1\to 1}}+1} \cdot \frac{e^{L_{q2\to 1}}-1}{e^{L_{q2\to 1}}+1} \cdot \frac{e^{L_{q3\to 1}}-1}{e^{L_{q3\to 1}}+1})}{1 - (\frac{e^{L_{q1\to 1}}-1}{e^{L_{q1\to 1}}+1} \cdot \frac{e^{L_{q2\to 1}}-1}{e^{L_{q2\to 1}}+1} \cdot \frac{e^{L_{q3\to 1}}-1}{e^{L_{q3\to 1}}+1})}$$

$$
\begin{aligned}
&= \log\left(\frac{e^{(L_{q_{1\to1}}+L_{q_{2\to1}}+L_{q_{3\to1}})}+e^{L_{q_{1\to1}}}+e^{L_{q_{2\to1}}}+e^{L_{q_{3\to1}}}}{e^{(L_{q_{1\to1}}+L_{q_{2\to1}})}+e^{(L_{q_{1\to1}}+L_{q_{3\to1}})}+e^{(L_{q_{2\to1}}+L_{q_{3\to1}})}+1}\right)\\
&= \log\left(e^{(L_{q_{1\to1}}+L_{q_{2\to1}}+L_{q_{3\to1}})}+e^{L_{q_{1\to1}}}+e^{L_{q_{2\to1}}}+e^{L_{q_{3\to1}}}\right)\\
&\quad -\log\left(e^{(L_{q_{1\to1}}+L_{q_{2\to1}})}+e^{(L_{q_{1\to1}}+L_{q_{3\to1}})}+e^{(L_{q_{2\to1}}+L_{q_{3\to1}})}+1\right). \quad (3.1)
\end{aligned}
$$

We follow the approximations in [10] to obtain a new approximation as follows:

$$\log(e^a+e^b+e^c+e^d)\approx \max(a,b,c,d)+\log(1+e^{-(\max(a,b,c,d)-\max_2(a,b,c,d))}), \quad (3.2)$$

where $\max_2(a,b,c,d)$ is the second biggest value in the set of (a,b,c,d). If we use the approximate equation (3.2) to simply (3.1), we can obtain

$$
\begin{aligned}
L_{r_{1\to4}} &= \log\left(e^{L_{q_{1\to1}}+L_{q_{2\to1}}+L_{q_{3\to1}}}+e^{L_{q_{1\to1}}}+e^{L_{q_{2\to1}}}+e^{L_{q_{3\to1}}}\right)\\
&\quad -\log\left(e^{L_{q_{1\to1}}+L_{q_{2\to1}}}+e^{L_{q_{1\to1}}+L_{q_{3\to1}}}+e^{L_{q_{2\to1}}+L_{q_{2\to3}}}\right)\\
&= \max\left(L_{q_{1\to1}}+L_{q_{2\to1}}+L_{q_{3\to1}},L_{q_{1\to1}},L_{q_{2\to1}},L_{q_{3\to1}}\right)+g_1(L_{q_{1\to1}},L_{q_{2\to1}},L_{q_{3\to1}})\\
&\quad -\max\left(L_{q_{1\to1}}+L_{q_{2\to1}},L_{q_{2\to1}}+L_{q_{3\to1}},L_{q_{1\to1}}+L_{q_{3\to1}},0\right)\\
&\quad -g_2(L_{q_{1\to1}},L_{q_{2\to1}},L_{q_{3\to1}}), \quad (3.3)
\end{aligned}
$$

where

$$
\begin{aligned}
&g_1(L_{q_{1\to1}},L_{q_{2\to1}},L_{q_{3\to1}})=\log\left(1+e^{-(\max(\alpha)-\max_2(\alpha))}\right), \quad (3.4)\\
&\text{with } \alpha=(L_{q_{1\to1}}+L_{q_{2\to1}}+L_{q_{3\to1}},L_{q_{1\to1}},L_{q_{2\to1}},L_{q_{3\to1}}),
\end{aligned}
$$

and

$$
\begin{aligned}
&g_2(L_{q_{1\to1}},L_{q_{2\to1}},L_{q_{3\to1}})=\log\left(1+e^{-(\max(\beta)-\max_2(\beta))}\right), \quad (3.5)\\
&\text{with } \beta=(L_{q_{1\to1}}+L_{q_{2\to1}},L_{q_{2\to1}}+L_{q_{3\to1}},L_{q_{1\to1}}+L_{q_{3\to1}},0).
\end{aligned}
$$

We can also obtain the general check node updates flow diagram as shown in Fig. 3.9, where a check node is connected to many bit nodes. The notation $\ominus$ is the computing unit which execute the formula in (3.3).

The performance comparison of the proposed algorithm and the conventional LLR is shown in Fig. 3.10. In simulation result, we can find the performances of the Radix-4 algorithm are better than the performance of LLR algorithm. By

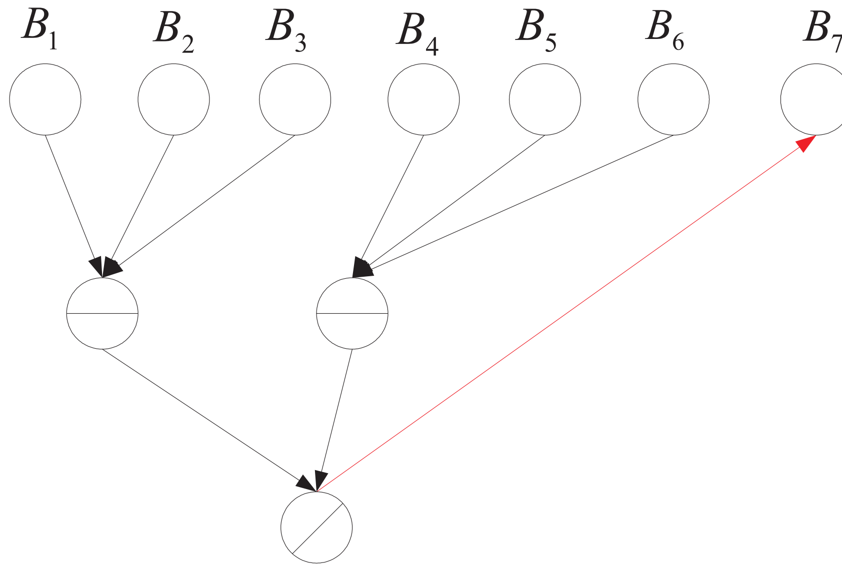


Figure 3.9: The flow diagram of check node update using Radix-4 algorithm.

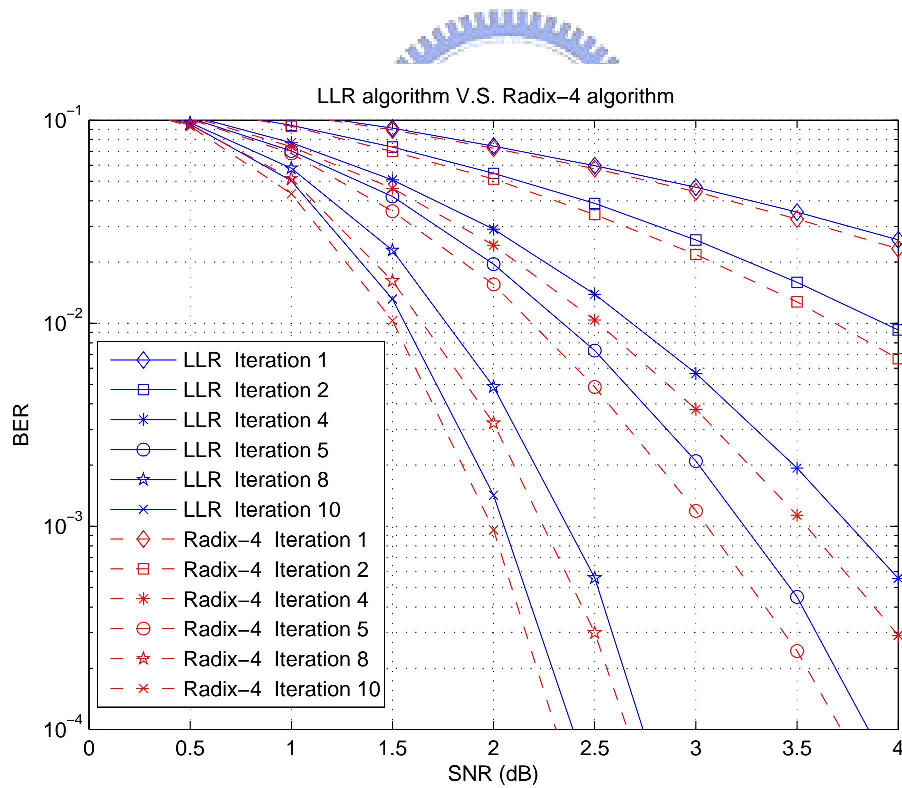


Figure 3.10: Performance comparison of the proposed Radix-4 algorithm and the conventional LLR.

following [20], we show the value of the check update with LLR algorithm and the proposed algorithm as in Figs. 3.11- 3.14, where the SNR is 4. We find that unlike Min-Sum algorithm the proposed algorithm can keep the quite large when value iteration unmber grows. This may be the reason that the performances of the proposed Radix-4 algorithm is better than that of LLR algorithm. The Radix-4 LDPC decoding has more complexity than the LLR decoding. The unit for check node update in LLR algorithm and Radix-4 algorithm can be implemented as show in Fig. 3.15 and Fig. 3.16 respectively.

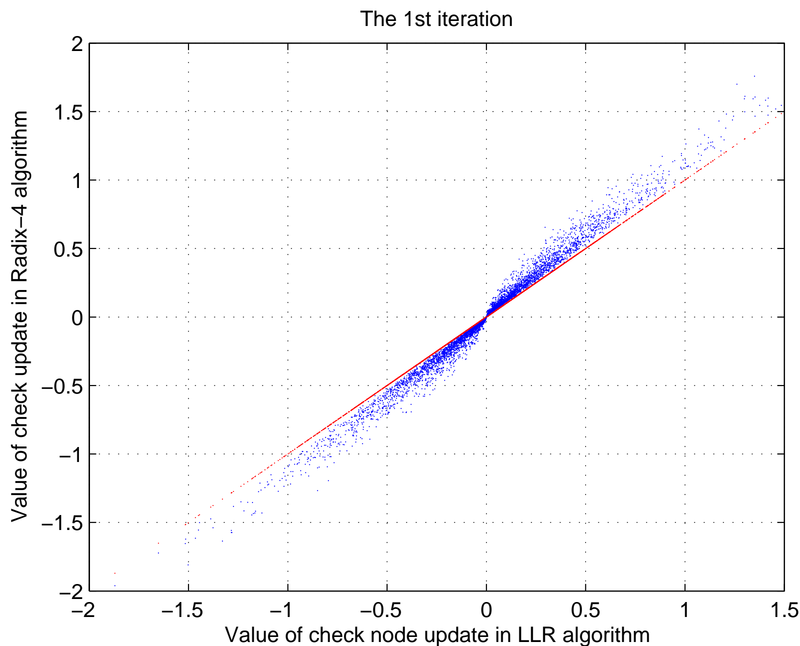


Figure 3.11: The value of check node update in different algorithm.

3.3 The comparison of Radix-4 and Min-Sum-Correct

From Fig. 2.13 and Fig. 3.9, we can obtain the individual number of state with seven nonzero elements in a row, as shown in table 3.1. It shows the number of state is less by Radix-4 algorithm than by Min-Sum-Correct algorithm. In

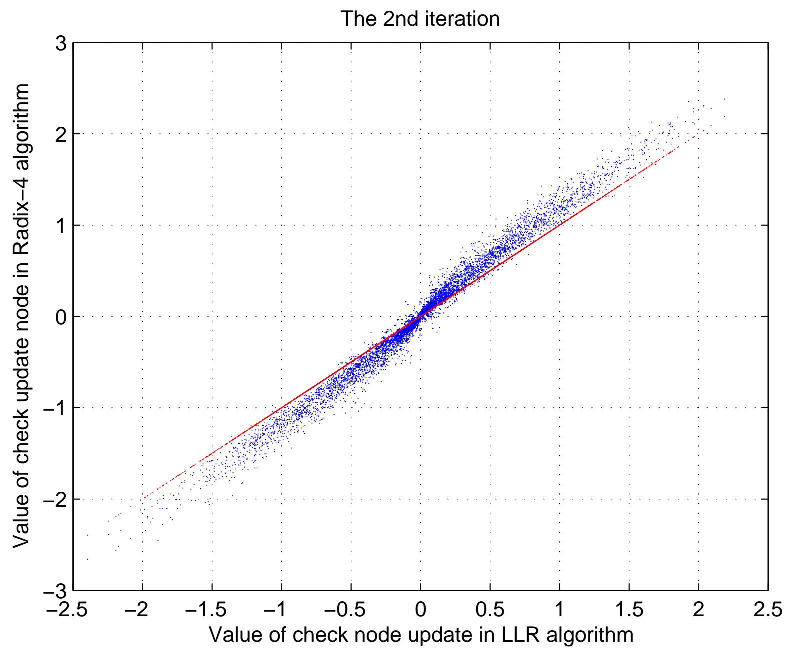


Figure 3.12: The value of check node update in different algorithm.

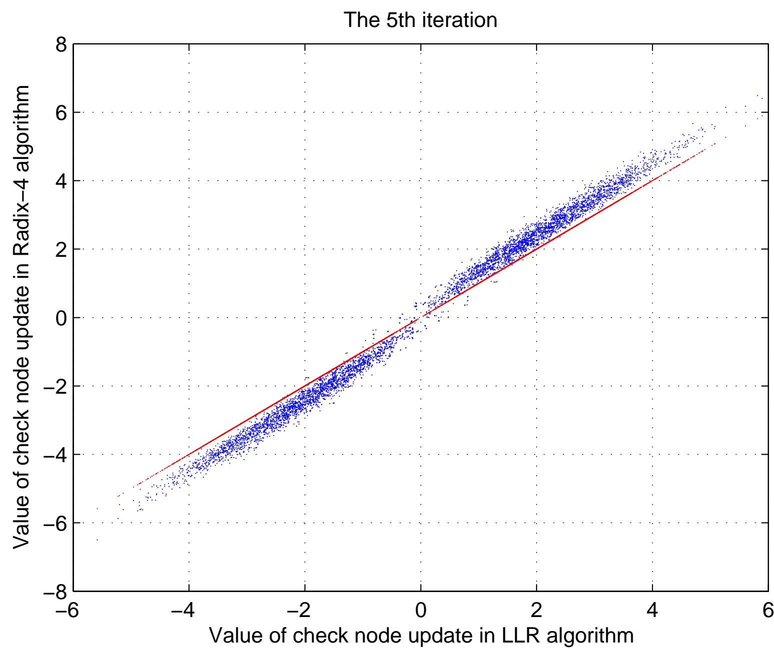


Figure 3.13: The value of check node update in different algorithm.

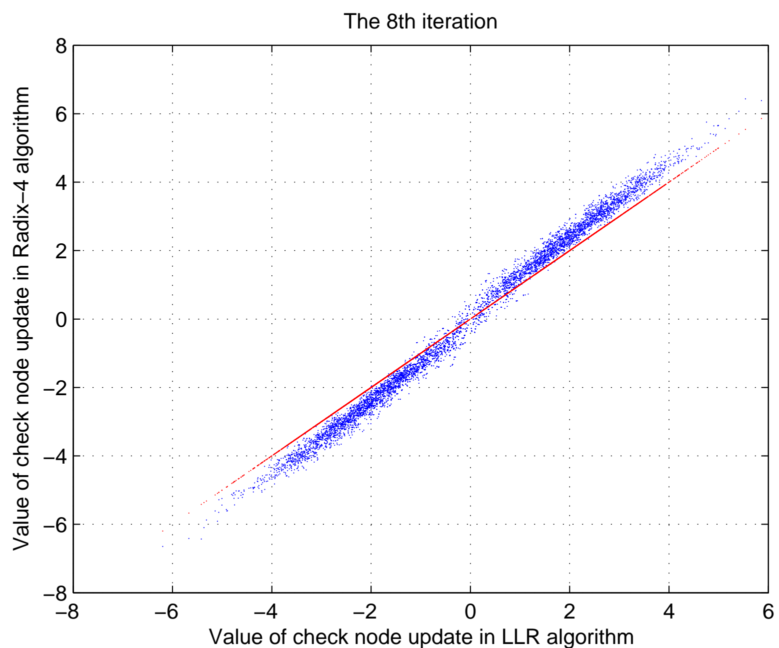


Figure 3.14: The value of check node update in different algorithm.

1896

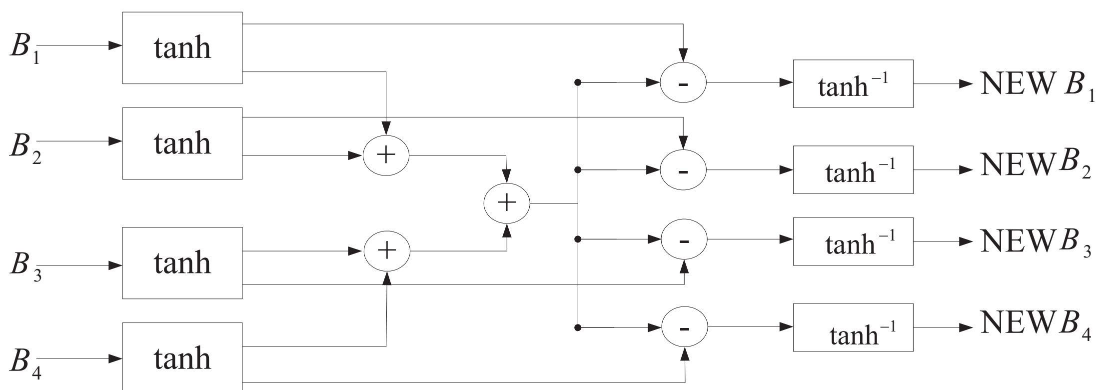


Figure 3.15: The unit for check update with 4 bit node in LLR algorithm.

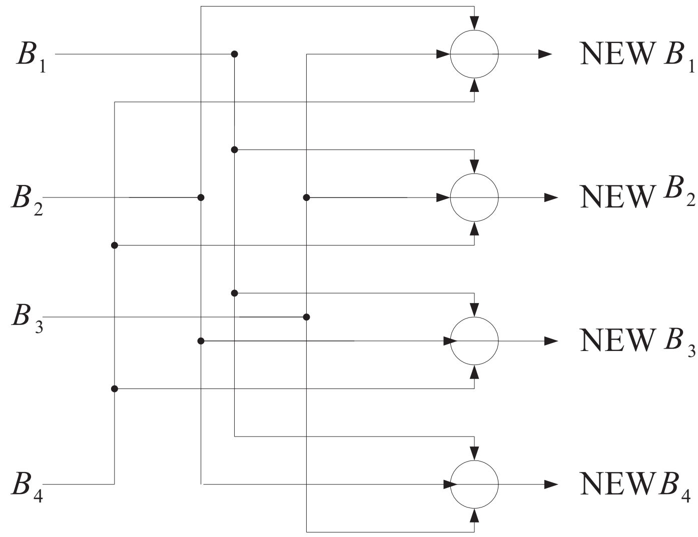


Figure 3.16: The unit for check update with 4 bit node in Radix-4 algorithm.

addition, we can obtain the overall decoding latency with partially parallel scheme by Radix-4 algorithm and Min-Sum-Correct algorithm, as show in Table 3.2. For example, assume subblock size is 81 bits with 10 nonzero elements in a row and 10 iterations. We can obtain the overlapped clock cycle by Radix-4 algorithm is (81+3)*(1+4*10)=3403 and overlapped clock cycle by Min-Sum-Correct is (81+5)*(1+4*10)=3525, where 81 is the size of basic matrix, 3 and 5 are needed stage, and 10 is iteration number. The decoding latency by Radix-4 algorithm is shorter than it by Min-Sum-Correct algorithm.

Number of the nonzero element in a row	7	8	10	20
Number of stage by Min-Sum-Correct (X)	3	3	4	5
Number of stage by Radix-4 (Y)	2	2	2	3

Table 3.1: The comparison of Radix-4 and Min-Sum-Correct.

	Radix-4	Min-Sum-Correct
Non-overlapped (clock cycles)	(Z+Y)*(Iteration*6)	(Z+X)*(Iteration*6)
Non-overlapped (clock cycles)	(Z+Y)*(1+Iteration*4)	(Z+X)*(1+Iteration*4)

X: Number of stage by Min-Sum-Correct
Y: Number of stage by Radix-4

Table 3.2: The decoding latency.

3.4 LUT circuit

Observing Min-Sum-Correct and the Radix-4 algorithm, we need to perform function of $\log(1 + e^{-|x|})$. In general, the special function is usually implemented by look-up table. The authors in [14] proposed two approximation methods, i.e. coarse quantization and piece-wise linear approximation as shown in Table 3.3 and Table 3.4. To reduce the implementation effort in VLSI design, we modified Table 3.4 as in Table 3.5. The simulation of different approximation methods and original function are shown in Fig. 3.17.

|x|	$\log(1+e^{-\|x\|})$	|x|	$\log(1+e^{-\|x\|})$
[0,0.196)	0.65	[1.05,1.508)	0.25
[0.196,0.433)	0.55	[1.508,2.252)	0.15
[0.433,071)	0.45	[2.252,4.5)	0.05
[0.71,1.05)	0.35	[4.5,+∞)	0.0

Table 3.3: Quantization table for $\log(1 + e^{-|x|})$.

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

Table 3.4: Piece-wise linear function for $\log(1+e^{-|x|})$.

|x|	$\log(1+e^{-\|x\|})$	|x|	$\log(1+e^{-\|x\|})$
[0,0.5)	$-\|x\|\times 2^{-1}+0.6875$	[2.0,3.0)	$-\|x\|\times 2^{-4}+0.25$
[0.5,1.5)	$-\|x\|\times 2^{-2}+0.575$	[3.0,4.5)	$-\|x\|\times 2^{-5}+0.125$
[1.5,2.0)	$-\|x\|\times 2^{-3}+0.375$	[4.5,+∞)	0.0

Table 3.5: The proposed piece-wise linear function for $\log(1+e^{-|x|})$.

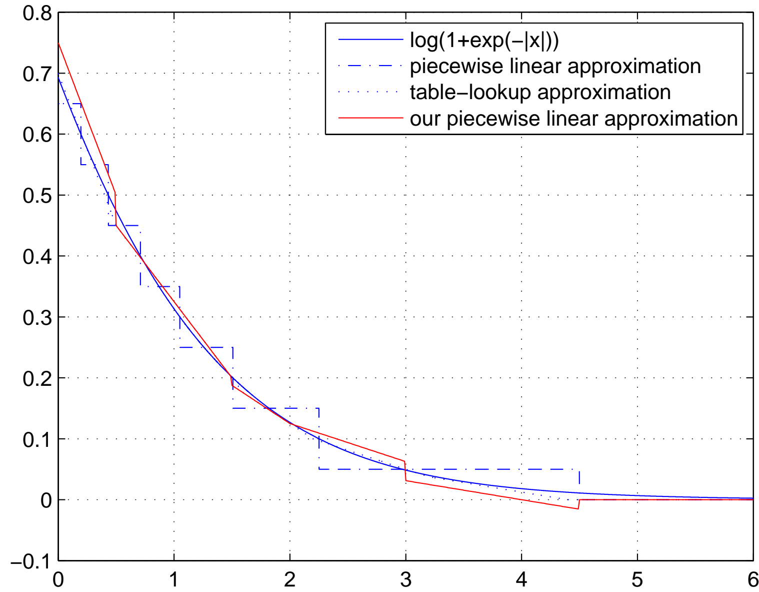


Figure 3.17: Comparison of using LLR and Radix-4.

3.5 Fixed point analysis

In VLSI implemented flow, the quantification effect is must considered. In order to show quantification effect, we need to run proposed algorithm with fixed point. In order to reduce complexity, we choose 4 bits for integer part and 4 bits for decimal. Fig. 3.18 show the floating point and fixed point simulation.

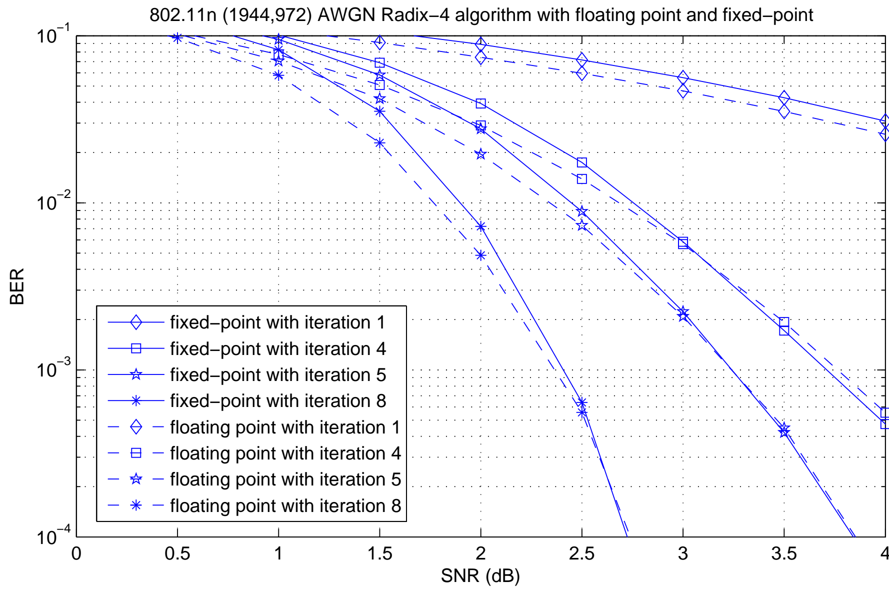


Figure 3.18: The performance of Radix-4 algorithm with fixed-point.

3.6 Proposed architecture

The VLSI design for the proposed LDPC decoders is shown in Fig. 3.19. The 8 bit input data consisted of 4 bits respectively for integer and decimal part. First the received date is passed input buffer and then fed into the one port memory bank and the two port memory bank simultaneously. The one port memory bank stores the initial channel value for to decide the valid codeword for syndrome check, and the two port memory bank is used to store the probability message exchanged between the unit for the check node update and that for bit node update. The date fed into the two port memory bank from input will be

passed to the unit for the check node update to perform check node update. After the check node update, the computing result will be rewritten in the two port memory bank. Then the data stored in the two port memory bank is fed into the unit for the bit node update to perform the bit node update. Similarly, after the bit node update, the computing result will be rewritten in the two port memory bank too. By performing storing and writing mentioned above, we say we can finish the iteration for one time. After iterative computing, the date in the two port memory bank and the one port memory bank are summed to decide valid codeword bits.

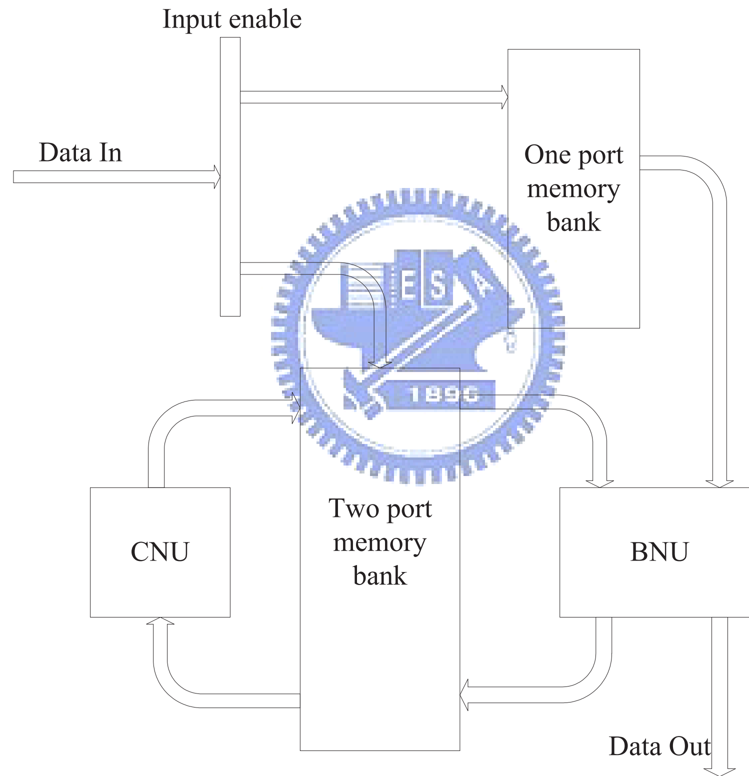


Figure 3.19: The overall architecture of the proposed LDPC decoder.

3.6.1 The unit for the check node update

To implement the proposed algorithm for code rate 1/2 in IEEE 802.11n standard, we need to consider two cases, as shown in Fig. 3.20. The last column represents

how many nonzero elements (elements that are not "-") in the corresponding row. There are only two different numbers in the last column, i.e. 7 and 8. Thus the we can obtain the flow diagram of the check node update as shown in Fig. 3.21 and Fig. 3.22. From two figures, we find that two kinds of operation units are needed, i.e. $\oslash$ and $\ominus$. $\oslash$ (the first unit for the check update) is an unit to perform formula (2.37) and $\ominus$ (the second unit for the check update) is an unit to perform formula (3.3).

	NEW	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
NEW	*OLD*	7	11	13	12	24	23	22	21	5	1	9	2	10	6	4	14	3	8	20	19	18	17	16	15	
1	1	11	79	1	–	–	–	–	–	50	57	50	–	–	–	–	0	–	–	–	–	–	–	–	–	7
2	9	–	–	–	32	–	–	0	0	14	64	30	–	–	52	–	–	–	–	–	–	–	–	–	–	7
3	11	–	12	–	–	0	0	–	–	35	2	–	56	–	–	57	–	–	–	–	–	–	–	–	–	7
4	10	–	–	–	–	–	0	0	–	0	–	77	45	9	–	70	–	–	–	–	–	–	–	–	–	7
5	12	–	–	1	16	0	–	–	–	60	24	51	–	–	–	–	–	61	27	–	–	–	–	–	–	8
6	8	–	27	–	–	–	–	0	–	38	65	72	–	–	57	–	–	–	–	0	–	–	–	–	–	7
7	7	56	–	0	–	–	–	–	–	–	69	52	79	–	–	–	–	79	–	0	0	–	–	–	–	8
8	6	42	–	–	8	–	–	–	–	8	0	50	–	–	–	–	–	–	–	–	0	0	–	–	–	7
9	5	–	–	–	–	–	–	–	–	66	40	28	–	–	–	20	–	–	22	–	–	0	0	–	–	7
10	4	–	–	–	–	–	–	–	–	53	62	35	53	–	–	–	–	–	3	–	–	–	0	0	–	7
11	3	–	–	–	–	–	–	–	–	24	30	56	–	14	37	–	–	–	–	–	–	–	–	0	0	7
12	2	–	–	–	–	–	–	–	–	0	3	55	–	7	–	–	0	28	–	–	–	–	–	–	0	7

Figure 3.20: Table shows how many nonzero (elements that are not "-") elements in rows.

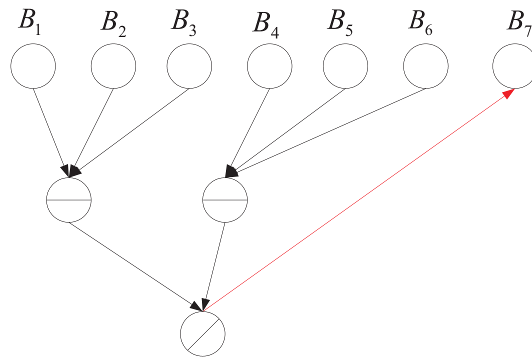


Figure 3.21: Case 1: A check node connected to 7 bit nodes.

Fig. 3.23 is The first operation unit for the check node update. There are 2 input in Fig. 3.23. We take the MSB of input A and B to perform xor operation,

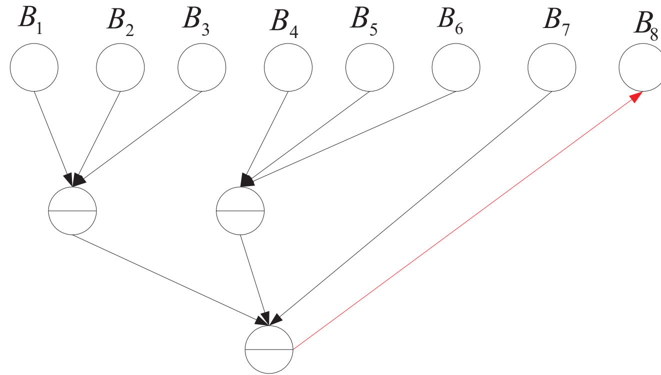


Figure 3.22: Case 2: A check node connected to 8 bit nodes.

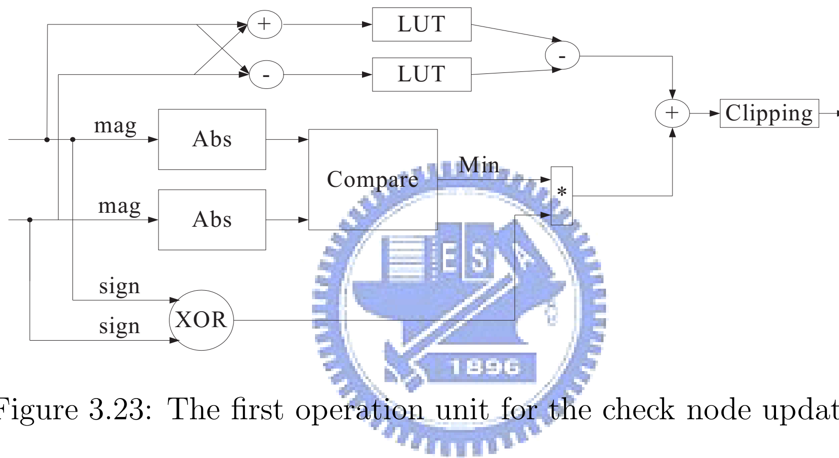


Figure 3.23: The first operation unit for the check node update.

1896

and the last 7 bits to obtain the absolute value of the input. Then, compare the absolute value of the input, and take the smaller value to combine with the output of xor operation. In addition, we take input A and B to add and subtract, and then perform table look-up and subtract. Final, the real output is that we sum the output of circuit above mentioned. The clipping component is used to clip the number of bits for the input of the bit node update.

There are 3 input in Fig. 3.24. We take the three inputs to perform addition individually as shown in Fig. 3.24. Then, we can obtain the largest value MAX_{11} and the second largest value MAX_{12} in the first comparison block. We can also obtain the largest value MAX_{21} and the second largest value MAX_{22} in the second comparison block. Then, we perform the subtract and table look-up. The

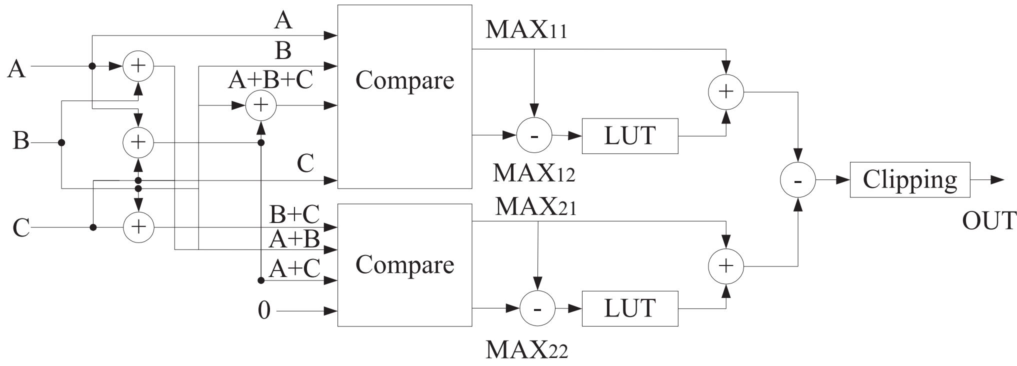


Figure 3.24: The second operation unit for the check node update.

final output can be obtained by performing add individual and then performing subtraction. The clipping component is used to clip the number of bits for the input of the bit node update.

3.6.2 The unit for the bit node update

In the step of the bit node update, we need to sum the probability message from all check nodes connected to this bit node. The number of the nonzero elements in the same column in IEEE 802.11n standard have been defined, i.e. 2, 3, 4, 11, and 12. Based on the reason mentioned above, we design units for bit node update with 12 inputs and a mux, as shown in Fig. 3.25. For Fig. 3.25, this is used for 12 inputs in standard. We sum all input and channel value, then subtract a target input individually excluding to be the input for the check node update. The FF is the flip-flop used to shorten data path and the clipping function is to clip carry bit that is the input of check node update. In addition, the MSB at point P is the hard-decision value of the decoded result. At the final stage a multiplexer is needed to select the desired output signal. This scheme enables a sharing circuit for BNU with 2, 3, 4 ,11 and 12 inputs.

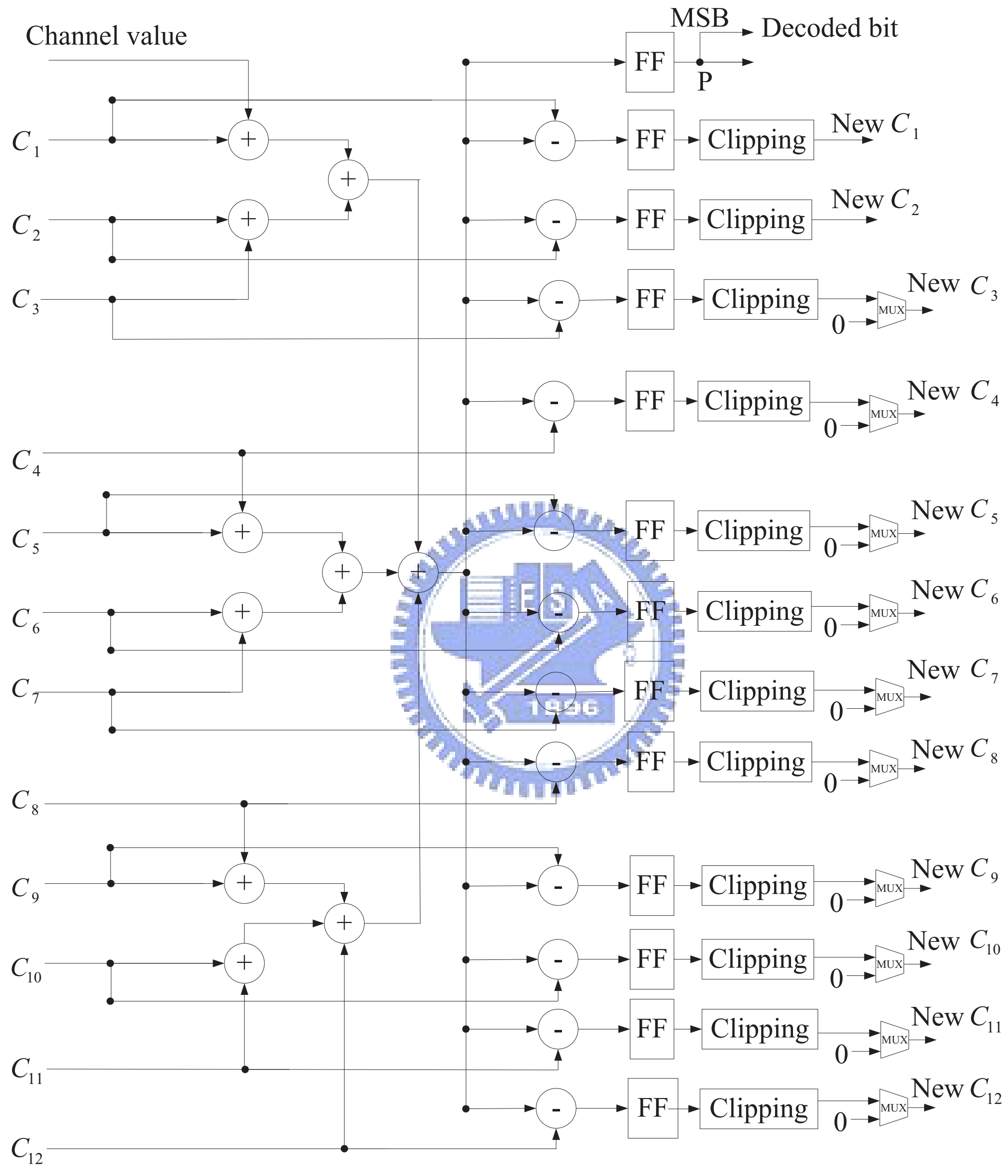


Figure 3.25: The unit for bit node update with 12 inputs.

Chapter 4

VLSI implementation

4.1 Design flow

In the section, we will introduce the design flow for the proposed LDPC decoder. The cell-based design flow is as shown in Fig. 4.1.

4.1.1 System model

We use the Matlab to build simulation environment. First the encoder is created according to the IEEE 802.11n standard, then Radix-4 algorithm with floating point and fixed point are used to observe the decoding performance. We use Radix-4 algorithm with floating point to observe the performance between BER and SNR, and Radix-4 algorithm with fixed point to choose the bit width and build test pattern for RTL code.

Using tool: Matlab.

4.1.2 RTL code

In this step, we use Verilog-HDL to describe the hardware architecture. The general design method is hierarchically method. Hence we need to divide the overall design into serval basic modules first. Then, connecting among the basic modules to complete the rough structure. Finally we need to perform bit true in order to make sure the output signals of RTL code and Matlab are same with same input signals. In addition, we have using memory in our architecture, so we

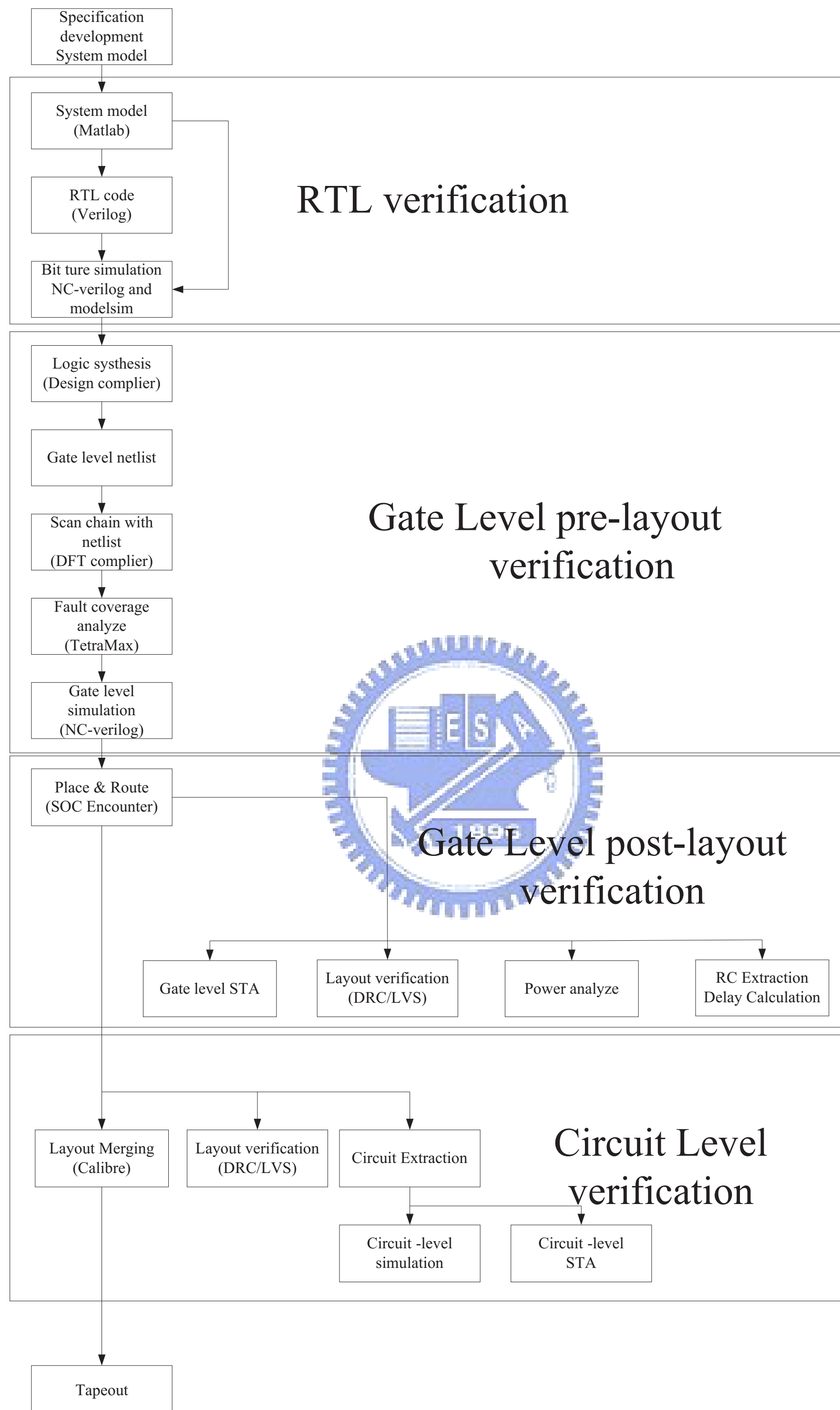


Figure 4.1: IC design flow.

use the memory compiler to generate need one port and two port register files.

Using tools: memory compiler, NC-verilog, modelsim, and Debussy nWave.

4.1.3 BIST

Because there are memory in our architecture, we need to add BIST circuit on memory control for the testability of IC. After adding BIST circuit, there are two mode in circuit, i.e. function mode and test mode. Function mode means that normal LDPC decoding can be performed, and test mode can be used test that there are have any error in memory.

Using tool: TurboBIST.

4.1.4 Synthesis

In this step, we start to synthesize our circuit. Before this step, our program is just hardware language, is not real gate. By using Synopsys Design Compiler to do the synthesis, our program can be translate as real gate. And we can get the rough area and some timing information of the gate. In our decoder design, all modules except the one port and two port register files are synthesized with TSMC 0.18um CMOS process technology.

Using tool: Design Compiler.

4.1.5 Gate-level simulation

After synthesis, we can get timing information of gate. So we can perform our circuit to check have any error with real time. We use NC-Verilog to do the gate-level simulation and use Debussy nWave to check waveform. By checking waveform, we can observe function exactitude with our predetermined clock period.

Using tools: NC-Verilog, and Debussy nWave.

4.1.6 DFT

For IC testing, we need to add mux in front of Flip-Flop and scan chains for the testability of IC. After adding mux, we can get there is any error between Flip-Flop and Flip-Flop by passing mux input signal. We use to Synopsys DFT Compiler to do scan chain insertion.

Using tool: DFT compiler.

4.1.7 ATPG

In the step, we use ATPG (automatic test pattern generator) of Synopsys TetraMax to generate test patterns for chip measurement.

Using tool: Synopsys TetraMax.

4.1.8 APR

We use SOC encounter to do automatical placement and routing (APR). Before placing and routing, we need to add power I/O and core I/O on Gate-level netlist and arrange location of input, output, I/O power, and core power on pad CIC supported. We need to consider core utilization, location of one port and two port register files, number of power ring, location and number of stripe to meet timing constraints from SDC file.

Using tool: SOC encounter.

4.1.9 DRC and LVS

In general, we usually have consider DRC (design rule checking) and LVS (layout V.S. schematic) in APR. But there is just rough check result in SOC encounter. So we need to do detail verification. We use the Calibre DRC to check whether there is any error with design rule and use the Calibre LVS to make sure that whether the layout and the schematic are identical or not.

Using tool: Calibre.

4.1.10 Post-layout simulation

In order to check function, we take the netlist and file of timing information generated by SOC encounter to run NC-Verilog. We can observe wave to find whether is any error by Debussy nWave. This is the last step to check function on myself work.

Using tools: NC-Verilog, and Debussy nWave.

4.2 Chip layout

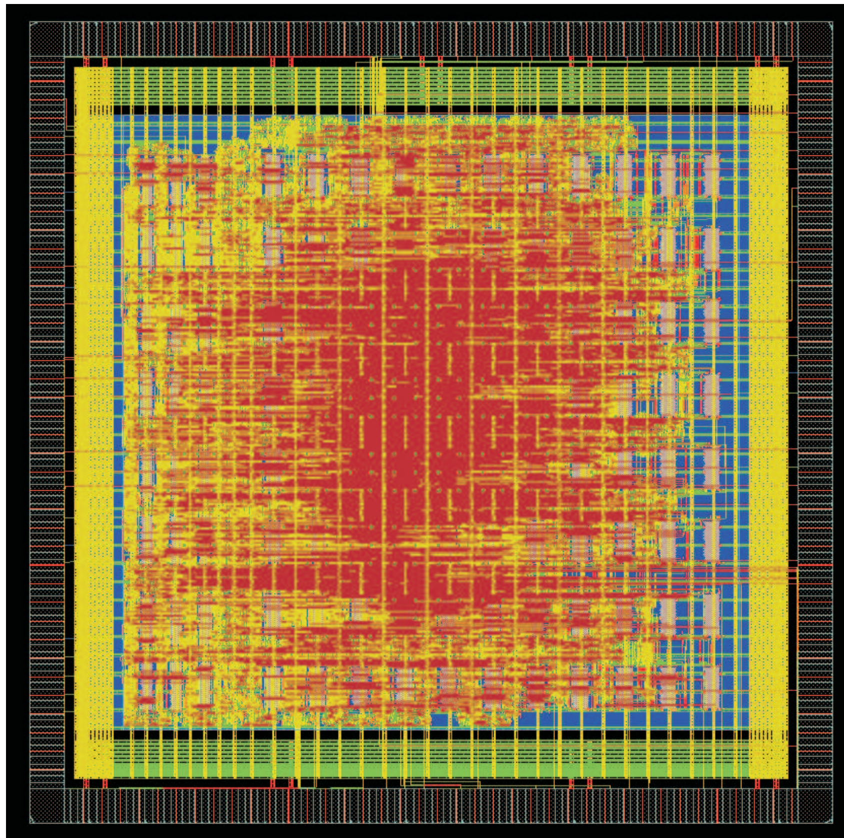


Figure 4.2: Layout of the proposed LDPC decoder.

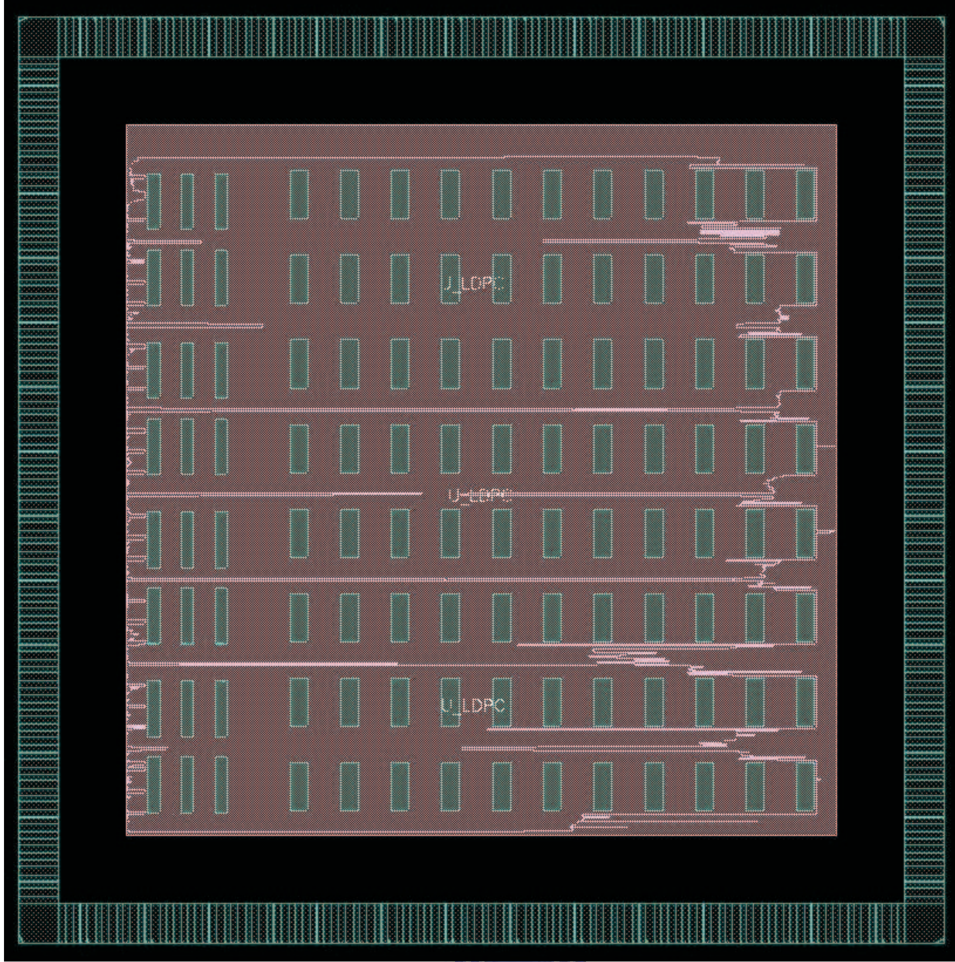


Figure 4.3: Layout of the proposed LDPC decoder.

4.3 Comparison and implementation result

A 3-mode LDPC decoder for IEEE 802.11n is implemented. With input quantization of 8 bits (4-bit integer part and 4-bit fraction part), synthesized with RTL compiler using TSMC CMOS 18um cell library, the total gate number of the proposed architecture is 780K. Using the TSMC 18um technology with 6 metal layers, the layout plot is presented as in Fig. 4.2 and Fig. 4.3 by SOC encounter for floorplaning, placement and routing. The core size is 17.9mm^2, clock frequency is 62.5MHz, and average power is 145mW. The data rate is 292$\sim$50Mbps.

Items	Specification
Technology	TSMC0.18um
Package	CQFP160
Chip Size	$30mm^2$
Core Area	$17.9mm^2$
Gate Count	780K
On-Chip Memory	(RF1SH82X8)X24 (RF2SH82X8)X88
Max Frequency	62.5MHZ
Throughput	292~50Mbp
Power Consumption	165mW

Table 4.1: Specification of the proposed LDPC decoder.

	[7]	[19]	[8]	This work
Multi-modes	NO	NO	NO	3 modes
Spec.	(1024,512)	(2048,1732)	(2048,1024)	(24*Z,12*Z) Z=27,54,81
Code Construction	Random	RS-based	Turbo-interleaved	QC_based
Decoding algorithm	LLR	LLR	Turbo	Radix-4
Technology	0.16um	0.18um	0.18um	0.18um
Parallelism	Fully(100%)	Fully(100%)	Partial(33%)	Partial(33%)
Iterations	64	32	16	1~7
Frequency	64MHz	100MHz	125MHz	62.5MHz
Area	$52.5mm^2$	$17.3mm^2$	$14.3mm^2$	$17.9mm^2$
Throughput	1Gbps	3.2Gbps	640Mbps	292~50Mbps
Power	690mW	N/A	787mW	145mW

Table 4.2: The comparison of different architectures.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we propose a decoding architecture of LDPC codes for IEEE 802.11n. We use partial parallel scheme to reduce the area and congestion. It can increased throughput by using proposed Radix-4 algorithm in our architecture, and we use a method to reduce decoding latency. In addition, we use simpler method to approximate function needed. Finally, the proposed architecture is implemented of 292∼50Mbps according post-layout simulation. The core size is 17.9mm^2, and clock frequency is 62.5MHz and average power is 145mW.

5.2 Future work

In this thesis, we only implement three mode which all are rate 1/2 LDPC decoder under for IEEE 802.11n standard. In fact, there are 12 modes which 4 kind of rate and 3 kind of basic matrix size. A 12 mode supported LDPC decoder design can be expected. In addition, the point that performance of Radix-4 algorithm is better than traditional decoding be also discuss. Furthermore, we should find a easy way to make routing easy.

Bibliography

[1] R. G. Gallager, “Low-density parity-check codes,” IRE Trans. on Information Theory, vol. IT-8, pp. 21-28, Jan. 1962.

[2] R. G. Gallager, “Low-Density Parity-Check Codes,” Cambridge, MA:MIT press, 1963.

[3] R. M. Tanner, “A recursive approach to low complexity codes,” IEEE Trans. Inform. Theory, vol. 74, no. 2, pp. 533-547, Sept. 1981.

[4] D. J. C. Mackay and R .M. Neal, “Near Shannon limit performance of low-density parity-check codes,” Electronics. Letters., vol. 32, pp. 1645-1646, Aug. 1996.

[5] D. J. C. MacKay, “Good error-correcting codes based on very sparse matrices,” IEEE Trans. Inf. Theory, 45, pp. 399V432, March 1999.

[6] S. Y. Chung, G. D. Forney, T. J. Richardson, and R. Urbanke, “On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit,” IEEE Comm. Lett., vol. COMM-5, no. 2, pp. 58-60, Feb. 2001.

[7] A. J. Blanksby and C. J. Howland, “A 690-mW 1-Gbs 1024-b rate 1/2 low-density parity-check code decoder,” IEEE Journal of Solid-State Circuits, Volume 37, Issue 3, pp. 404 V 412, March 2002.

[8] M. Mansour and N. Shanbhag, “A 640-Mb/s 2048-bit programmable LDPC decoder chip,” IEEE Journal of Solid- State Circuits, vol. 41, no.3, pp. 684-698, March 2006.

[9] Xin-Yu Shih, Cheng-Zhou Zhan, Cheng-Hung Lin, and An-Yeu Wu, “An 8.29mm^2 52mW Multi-mode LDPC Decoder Design for Mobile WiMAX

System in 0.13um CMOS Process," IEEE Jour. Solid-State Circuits, vol. 43, no. 3, pp. 672-683, Mar. 2008.

[10] M. Bicherstaff, L. Davis, C. Thomas, D. Garrett, and C. Nicol, "A 24Mb/s Radix-4 LogMAP Turbo Decoder for 3GPP Mobile Wireless," ISSCC03 Dig. Tech. Papers, pp. 150 - 151, Feb. 2003.

[11] H. Zhang and T. Zhang, "Design of VLSI Implementation-Oriented LDPC Code," Vehicular Technology Conference, vol. 1, pp. 670-673, 2003.

[12] T. J. Richardson and R. L. Urbanke, "Efficient encoding of Low-Density Parity-Check codes," IEEE Trans. Inform. Theory, vol. 47, no. 2, pp. 638-656, Feb. 2001.

[13] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," IEEE Trans. Inform. Theory, vol. 42, pp. 429-445, Mar. 1996.

[14] X. Y. Hu, E. Eleftheriou, D. M. Arnold and A. Dholakia, "Efficient implementation of the sum-product algorithm for decoding LDPC codes," Proc. IEEE GLOBECOM01, vol. 2, pp. 25-29, Nov. 2001.

[15] N. Wiberg,"Codes and Decoding on General Graphs" Ph.D. thesis, Linkoping University, Sweden, 1996.

[16] J. Chen and M. P. C. Fossorier, "Near optimum universal belief propagation based decoding of low-density parity check codes," IEEE Comm. Lett., Vol. 50, pp. 406-414, March 2002.

[17] I. C. Park and S. H. Kang, "Scheduling algorithm for partially parallel architecture of LDPC decoder by matrix permutation," Proc. IEEE International Symposium Circuits and Systems, pp. 5778-5781 Vol. 6, May 2005.

[18] K. Shimizu, T. Ishikawa, N. Togawa, T. Ikenaga and S. Goto, "Partially-parallel LDPC decoder based on high-efficiency message-passing algorithm," Proc. IEEE International Conference Computer Design: VLSI in Computers and Processors, pp. 503-510, Oct. 2005.

[19] A. Darabiha, A.C. Carusone, and F.R. Kschischang, “Multi-Gbit/sec low density parity check decoders with reduced interconnect complexity,” ISCAS 2005, Vol. 5, May 2005.

[20] W. Ryan, “Low-Density Parity-Check Codes,” June 2005.