國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

IEEE802.16e OFDM 與 OFDMA 通道編

碼技術與數位訊號處理器實現之研究

Research in Channel Coding Techniques and

DSP Implementation for IEEE 802.16e

OFDM and OFDMA

研 究 生：陳勇竹

指導教授：林大衛 博士

中 華 民 國 九 十 五 年 六 月

IEEE 802.16e OFDM 與 OFDMA 通道編碼技術與數位訊號處理器實現之研究

Research in Channel Coding Techniques and DSP Implementation for IEEE 802.16e OFDM and OFDMA

研究生: 陳勇竹　　Student: Yung-Chu Chen

指導教授: 林大衛 博士　　Advisor: Dr. David W. Lin

國 立 交 通 大 學

電子工程學系　電子研究所碩士班

碩士論文

A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics
College of Electrical and Computer Engineering
National Chiao Tung University
in Partial Fulfillment of Requirements
for the Degree of
Master of Science
in
Electronics Engineering
June 2006
Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

IEEE 802.16e OFDM 與 OFDMA 通道編碼技術與數位訊號處理器實現之研究

研究生：陳勇竹　　指導教授：林大衛 博士

國立交通大學

電子工程學系 電子研究所碩士班

摘要

IEEE802.16 無線通訊標準中，於系統的傳送端訂定了前向誤差改正編碼的機制，藉此減低通訊頻道中雜訊失真的影響。通道編碼是本論文的重點。

本篇論文的前半部份重點在於，實現 IEEE 802.16e OFDM 所訂定的前向誤差改正編碼系統於數位訊號處理器(DSP)上，並且針對 DSP 平台的特性以及前向誤差改正編碼的演算法進行程式的改進。在此篇論文中，我們將標準中制訂的四個必備的前向誤差改正編碼系統，實現在以德州儀器公司所發展的 DSP 為核心的平台上。由於我們關注的重點在於程式的執行效率，因此簡短地介紹過我們使用的前向誤差改正編碼的演算法以及 DSP 平台的架構與軟體最佳化技巧後，我們將逐步地闡述如何在 DSP 平台上最佳化我們的程式。最後，前向誤差改正編碼的編碼器部份，經過改進後，於 DSP 模擬器上，可以到每秒 8013K 位元的處理速度，而解碼器的部份可以達到每秒 769K 位元的處理速度。

本論文後半部份強調 IEEE 802.16e OFDMA 中低密度奇偶校驗碼複雜度的降低。我們介紹一些分析低密度奇偶校驗碼的工具後，逐步地闡述低密度奇偶校驗碼傳統的解碼演算法，並且介紹一些降低解碼複雜度的演算法。最後我們在加成性白色高斯通道下模擬了各種調變與各種解碼演算法，並把模擬之結果與一些數學分析的結果做比較。模擬的結果顯示這幾個降低複雜度的演算法和傳統的解碼表現相當接近，甚至更好。若從性能，延遲時間，運算複雜度，延遲時間，及需要的記憶體的角度來看，我們可以彈性的挑選適當的解碼演算法來使用，以取得之間的平衡。

Research in Channel Coding Techniques and DSP Implementation for IEEE 802.16e OFDM and OFDMA

Student: Yung-Chu Chen Advisor: Dr. David W. Lin

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

Abstract

In the IEEE 802.16e wireless communication standard, a Forward Error Correction (FEC) mechanism is presented at the transmitter side to reduce the noisy channel effect. The focus is on the channel coding.

The focus of the fist part of this thesis is DSP implementation of the FEC schemes defined in IEEE 802.16e OFDM standard and modifying FEC algorithms to match the architecture of DSP platform. We have implemented four required FEC schemes defined in the standard on the Texas Instruments (TI) TMS320C6416 digital signal processor (DSP). After a brief review of the algorithms, we describe the DSP hardware architecture and its software optimization techniques. We then explain how we optimize the FEC programs on the DSP platform step by step since the speed performance is our major concern. At the end, the improved FEC encoder can achieve a data processing rate of 8013 kbits/sec and the improved FEC decoder can achieve a processing rate of 769 kbits/sec on the TI C64xx DSP simulator.

The focus of second part is the complexity-reduction for low-density parity-check (LDPC) codes defined in IEEE 802.16e OFDMA. We describe some tools to analyze the LDPC codes. We then explain the conventional decoding algorithm, and some reduced-complexity decoding algorithms. Finally, we simulate the LDPC codes for all kinds of modulation and decoding algorithms in AWGN and compare the simulation results with analytical results. Simulation results show that these reduced-complexity decoding algorithms for LDPC codes achieve a performance very close to that of conventional algorithm, or even better. We can flexibility select the appropriate decoding scheme from performance, computational-complexity, latency, and memory-requirement perspectives.

誌謝

這篇論文能夠順利完成，最要感謝的人是我的指導教授 林大衛 博士。在這二年的研究生涯中，不論是學業上或生活上，處處感受到老師的用心，尤其是修改論文時相當的用心。除了豐富的學識和研究，老師親切、認真的待人處事態度，也是我景仰、學習的目標。

另外要感謝的，是實驗室的吳俊榮學長和洪崑健學長。謝謝你們熱心地幫我解決了許多通訊方面相關的疑問。

感謝通訊電子與訊號處理實驗室(commlab)，提供了充足的軟硬體資源，讓我在研究中不虞匱乏。感謝 91 級 eras(子瀚)、 hclin(筱晴)、長毛(建統)，92 級 klinsman(昱升)、andlight(漢光)、ching(汝芩)、u8811021(思浩)、osban (承毅)、Richard Tung(景中)、gem(盈閔)、buggy(志楹)、enz(瑛姿)，93 級 Gauss(鴻志、pay3)、odom(旻弘)、allenlai(阿蛋)、ssdai(世炘)、stan(崇文少爺、林金龍)、lotus(國偉)、shiryu(治傑)、蔡蟲(崇諺)、jackyboss(國洋)、Jerome(建志)、jerry(宜寬)等實驗室成員，平日和我一起唸書，一起討論，也一起打混，讓我的研究生涯充滿歡樂又有所成長。期待大家畢業之後都能有不錯的發展。

最後，要感謝的是我的家人，他們的支持讓我能夠心無旁騖的從事研究工作。

謝謝所有幫助過我、陪我走過這一段歲月的師長、同儕與家人。謝謝！

誌於 2006.7 風城交大

勇竹

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scope of the Work

Digital wireless transmission with multimedia contents is a trend in the next generation of consumer electronics field. Due to this demand high data transmission rate and mobility are needed. Thus the OFDM modulation technique for wireless communication has been the main stream in the recent years. IEEE has completed several standards such as IEEE 802.11 series for LANs (local area networks) and IEEE 802.16 series for MANs (metropolitan area networks) based on OFDM technique. Our study is based on the IEEE 802.16e standard, which specifies the air interface of mobile broadband wireless access systems providing multiple access.

One major problem with wireless communication is that the transmission channel is not noiseless. The transmitted signals are easily interfered and distorted by different types of noise sources such as the crowd traffic, bad weather, the obstacle of buildings, etc. Multimedia service contains broad range of contents such as audio, video, still image, and the traditional speech. These services would exhibit untolerable quality if they cannot detect and recover the errors introduced from the noisy channel. To improve the robustness of the wireless communication against the noisy channel condition, the FEC (forward-error-

correcting coding) mechanism is usually a must to overcome the channel errors for almost every commercial communication standard, including the IEEE 802.16e.

This work studies two parts of IEEE802.16e. One is the implementation of the FEC schemes under OFDM on a digital signal processor (DSP). And the other part is mainly the complexity-reduced decoding algorithms for the FEC schemes under OFDMA for future implementation on DSP.

The second part of work is part of a group project that gears at studying and construction (using DSPs) of IEEE802.16e-based transmission system prototype for mobile broadband communication. The intended span of the group project is from August 2005 to July 2008. Our study constitutes part of the first year's work.

The channel coding scheme in IEEE802.16e for OFDM employs concatenated coding with shortened punctured Reed-Solomon code as outer code and punctured convolutional code as inner code. In addition, bit interleaver and M-ary QAM modulation are used after the concatenated code, whereas the channel coding scheme in IEEE802.16e for OFDMA, we consider the LDPC codes, bit interleaver and M-ary QAM modulation.

1.2 Organization of This Thesis

This thesis is organized as follows.

- Chapter 2 introduces the FEC schemes of IEEE 802.16e and introduces some tools to analyze the LDPC codes.
- Chapter 3 describes the DSP implementation environment.
- Chapter 4 introduces the DSP implementation and optimization of the OFDM FEC schemes.

- Chapter 5 presents some decoding algorithms and simulation results and compares different decoding algorithms from simulation.
- Chapter 6 contains the conclusion and point out some future work.

Chapter 2

Overview of IEEE 802.16e FEC Specifications

2.1 FEC Specifications for WirelessMAN-OFDM [1]

The channel coding scheme used in IEEE 802.16e OFDM, as shown in Fig. 2.1, is a concatenated code employing the Reed-Solomon (RS) code as the outer code and convolutional code (CC) as the inner code. Input data streams are divided into RS blocks, and then each RS block is encoded by convolutional code. The block-by-block coding makes the whole concatenated code a block-based coding scheme.

The convolutional code is used to "clean up" the channel for the RS code, which in turn corrects the burst errors emerging from the convolutional decoder. In this way, the bit error

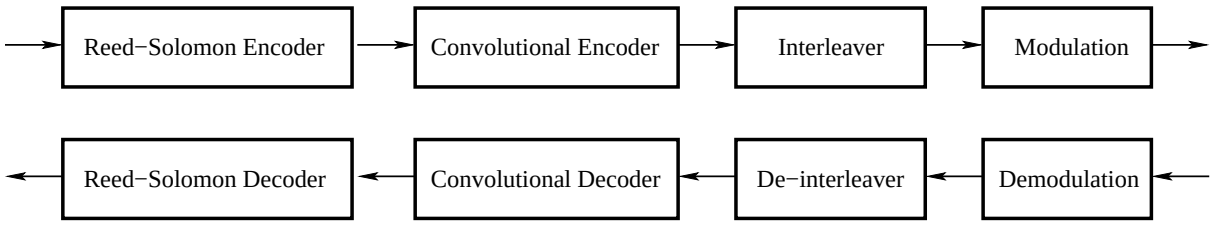


Figure 2.1: Channel coding structure in transmitter (top path) and decoding in receiver (bottom path).

Table 2.1: Mandatory Channel Coding Schemes for each Modulation Method

Modulation	Uncoded Block Size (Bytes)	Overall Code Rate	Coded Block Size (Bytes)	RS Code	CC Code Rate
BPSK	12	1/2	24	$(12, 12, 0)$	1/2
QPSK	24	1/2	48	$(32, 24, 4)$	2/3
QPSK	36	3/4	48	$(40, 36, 2)$	5/6
16QAM	48	1/2	96	$(64, 48, 8)$	2/3
16QAM	72	3/4	96	$(80, 72, 4)$	5/6
64QAM	96	2/3	144	$(108, 96, 6)$	3/4
64QAM	108	3/4	144	$(120, 108, 6)$	5/6

rate (BER) can decrease exponentially [3]. In addition, between the convolutional coder and the modulator is a bit interleaver, which protects the convolutional code from severe impact of burst errors and increases overall coding performance. This approach has been termed "bit-interleaver coded modulation (BICM)" in the literature [4].

To make the system more flexibly adaptable to the channel condition, there are seven coding-modulation schemes defined in IEEE 802.16e, as shown in Table 2.1. The different coding rates are made by shortening and puncturing the native RS code and with puncturing of the native convolutional code. The shortening and puncturing mechanisms in RS coding create different block sizes and different error-correction capability RS codes through one RS coder. The puncturing mechanism in CC coding can provide variable code rates through one CC coder.

2.1.1 Reed-Solomon Code Specification

The Reed-Solomon code in IEEE802.16e is derived from a systematic RS ($N = 255$, $K = 239$, $T = 8$) code on GF(2^8), where N is number of overall bytes after encoding, K is number

of data bytes before encoding, and T is number of data bytes which can be corrected. The following polynomials are used for the systematic code:

$$\text{Field generator polynomial:} \quad p(x) = x^8 + x^4 + x^3 + x^2 + 1. \tag{2.1}$$

$$\begin{aligned}\text{Code generator polynomial:} \quad g(x) &= (x + \lambda^0)(x + \lambda^1) \cdots (x + \lambda^{2T-1}),\ \lambda = 0\text{x}2, \\ &= g_{15}x^{15} + g_{14}x^{14} + \cdots + g_1 x + g_0. \end{aligned} \tag{2.2}$$

This code is then shortened and punctured to enable variable block sizes and variable error-correction capability. The modified RS code is denoted as (N', K', T') and the generator polynomial for RS code is given by

$$g(x) = (x + \lambda^0)(x + \lambda^1) \cdots (x + \lambda^{2T-1}). \tag{2.3}$$

When a block is shortened to K' data bytes, the first $239 - K'$ bytes of the encoder block are filled with 0s. When a codeword is punctured to permit T' bytes to be corrected, only the first $2T'$ of the total 16 parity bytes are employed.

2.1.2 Encoding of the Reed-Solomon Code [5]

We use the (64,48,8) RS code to explain the encoding process. Let the information data to the (255,239,8) systematic RS be represented as:

$$\begin{aligned} I(x) &= I_{238}x^{238} + I_{237}x^{237} + \cdots + I_{37}x^{37} + I_{36}x^{36} + I_{35}x^{35} + I_{34}x^{34} + \cdots + I_1 x + I_0 \\ &= (I_{238}, I_{237}, \cdots, I_{37}, I_{36}, I_{35}, I_{34}, \cdots, I_1, I_0). \end{aligned} \tag{2.4}$$

Then the resulting codeword is given by

$$\begin{aligned} C(x) &= I(x) \cdot x^{16} + R(x) \\ &= (I_{238}, I_{237}, \cdots, I_{37}, I_{36}, I_{35}, I_{34}, \cdots, I_1, I_0, R_{15}, \cdots, R_5, R_4, \cdots, R_1, R_0) \end{aligned} \tag{2.5}$$

where

$$\begin{aligned} R(x) &= I(x) \cdot x^{16} \bmod g(x) \\ &= R_{15}x^{15} + \cdots + R_5 x^5 + R_4 x^4 + \cdots + R_1 x + R_0 \\ &= (R_{15}, \cdots, R_5, R_4, \cdots, R_1, R_0). \end{aligned} \tag{2.6}$$

When shortened and punctured to (64,48,8), the first 191= $(239-48)$ information bytes are assigned 0, i.e., $I_{238} = I_{237} = \cdots = I_{48} = 0$, and the first 16= $(2 \cdot 8)$ bytes of $R(x)$ will be employed in the codeword. Now the information data of (64,48,8) will be

$$\begin{aligned} I'(x) &= I_{47}x^{47} + I_{46}x^{46} + \cdots + I_1 x + I_0 \\ &= (I_{47}, I_{46}, \cdots, I_1, I_0), \end{aligned} \tag{2.7}$$

and the codeword will be

$$\begin{aligned} C'(x) &= I'(x) \cdot x^{16} + R'(x) \\ &= (I_{47}, I_{46}, \cdots, I_1, I_0, R_{15}, \cdots, R_1, R_0) \end{aligned} \tag{2.8}$$

where

$$\begin{aligned} R'(x) &= \text{first 16 bytes of } (I'(x) \cdot x^{16} \bmod g(x)) \\ &= R_{15}x^{15} + \cdots + R_1 x^1 + R_0 x^0 \\ &= (R_{15}, \cdots, R_1, R_0). \end{aligned} \tag{2.9}$$

A systematic RS encoder is depicted in Fig. 2.2.

2.1.3 Convolutional Code Specification

Each RS block is encoded by a binary convolutional encoder, which has native rate of 1/2, a constraint length equal to 7, and the generator polynomials for the two output bits are 171_{OCT} and 133_{OCT}. The generator is depicted in Fig. 2.3.

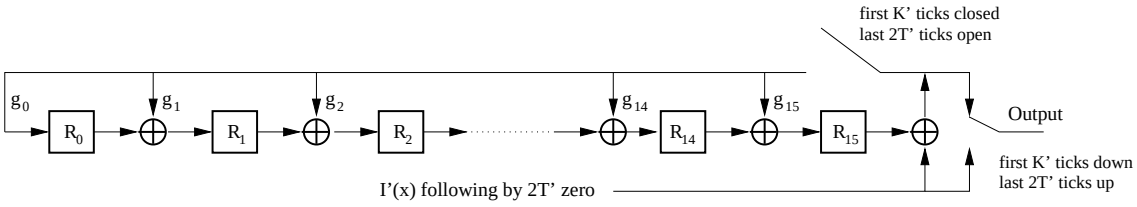


Figure 2.2: Shortened and punctured Reed-Solomon encoder (from [5]).

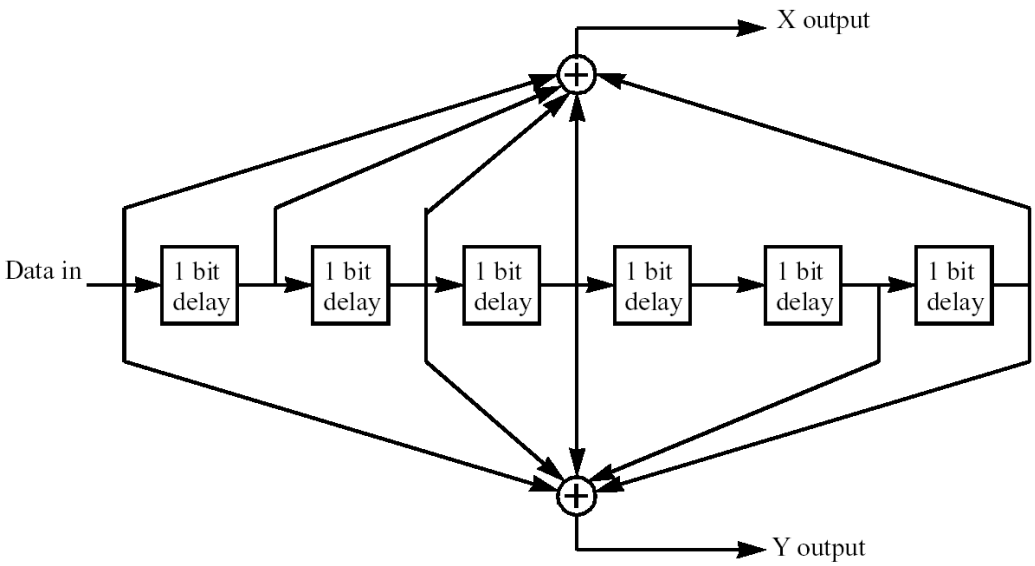


Figure 2.3: Convolutional encoder of rate 1/2 (from [1]).

This convolutional code is then punctured to allow different rates, which is known as rate-compatible punctured convolutional coding (RCPC). A single 0x00 tail byte is appended to the end of each RS output data block to initialize the CC encoder's memory.

2.1.4 Encoding of Punctured Convolutional Code

The convolutional code encoding structure is shown in Fig. 2.3. It consist of one input bit, six memory elements (shift registers), and two output bits generated by first performing the AND operations on the generator polynomial coefficients and the contents of the memory elements padded with the input bit, then performing the operation of XOR on each bits generated by the previous AND operation. Then we do the puncturing. The puncturing patterns and serialization order of the convolutional code in IEEE802.16e are defined in Table 2.2. In this table, "1" means a transmitted bit and "0" denotes a removed bit, whereas X and Y are in reference to Fig. 2.3. Note that the D_{free} has been changed from that of the native convolutional code with rate 1/2, which is equal to 10 [6, Chapter 8].

2.1.5 Interleaver

The encoded data bits are interleaved by a block interleaver with a block size corresponding to the number of coded bits per the specified allocation, N_{cbps} (see Table 2.3). The interleaver is defined by a two-step permutation. The first ensures that adjacent coded bits are

Table 2.2: The Inner Convolutional Code with Puncturing Configuration

	Code Rates			
Rate	1/2	2/3	3/4	5/6
D_{free}	10	6	5	4
X	1	10	101	10101
Y	1	11	110	11010
XY	X_1Y_1	$X_1Y_1Y_2$	$X_1Y_1Y_2X_3$	$X_1Y_1Y_2X_3Y_4X_5$

Table 2.3: Bit Interleaved Block Sizes and Modulos

Modulation	Coded Bits per Bit Interleaved Block (N_{cbps})	Coded Bits per Carrier (N_{cpc})	Modulo Used (d)
BPSK	192	1	12
QPSK	384	2	12
16QAM	768	4	12
64QAM	1152	6	12

mapped onto non-adjacent carriers. The second insures that adjacent coded bits are mapped alternately onto less or more significant bits of the constellation, thus avoiding long runs of lowly reliable bits.

Let $s = \text{ceil}(N_{cpc}/2)$, k be the index of the coded bit before the first permutation, m the index after the first and before the second permutation and j the index after the second permutation, just prior to modulation mapping. The first permutation is defined by

$$m = (\frac{N_{cbps}}{d}) \cdot \ k_{mod(d)} + floor(\frac{k}{d}), \quad k = 0, 1, \cdots, N_{cbps} - 1, \tag{2.10}$$

and the second permutation by

$$j = s \cdot \ floor(\frac{m}{s}) + (m + N_{cbps} - floor(\frac{d \cdot m}{N_{cbps}}))_{mod(s)}, \quad m = 0, 1, \cdots, N_{cbps} - 1. \tag{2.11}$$

The de-interleaver, which performs the inverse operation, is also defined by two permutations. Let j be the index of the received bit before the first permutation, m be the index after the first and before the second permutation, and k be the index after the second permutation, just prior to delivering the coded bits to the convolutional decoder. The first permutation is defined by

$$m = s \cdot floor(\frac{j}{s}) + (j + floor(\frac{d \cdot j}{N_{cbps}}))_{mod(s)}, \quad j = 0, 1, \cdots, N_{cbps} - 1, \tag{2.12}$$

and the second permutation by

$$k = d \cdot m - (N_{cbps} - 1) \cdot floor(\frac{d \cdot m}{N_{cbps}}), \quad m = 0, 1, \cdots, N_{cbps} - 1. \tag{2.13}$$

2.1.6 Modulation

After bit interleaving, the data bits are entered serially to the constellation mapper. BPSK, QPSK and Gray-mapped 16-QAM are supported, whereas the support of Gray-mapped 64-QAM is optional. The constellations as shown in Fig. 2.4 shall be normalized by multiplying the constellation points with the indicated factor c to achieve equal average power. The constellation-mapped data shall be subsequently modulated onto the allocated data carriers.

2.2 FEC Specifications for WirelessMAN-OFDMA [7]

One of the channel coding scheme used in IEEE802.16e OFDMA is using low-density parity-check (LDPC) code. The input data are first encoded by the LDPC encoder. The encoder output is then interleaved by the bit interleaver described in Section 2.2.3. To make the system more flexibly adaptable to the channel condition, there are three different modulation types which would be depicted in Section 2.2.4.

LDPC codes are a special case of error correcting codes that have recently been receiving a lot of attention because of their very high throughput and very good decoding performance. Inherent parallelism of the message passing decoding algorithm for LDPC codes makes them very suitable for hardware implementation. The LDPC codes can be used in any digital environment that high data rate and good error correction are important.

Gallager [8] proposed LDPC codes in the early 1960s, but his work received no attention until after the invention of turbo codes in 1993, which used the same concept of iterative decoding. In 1996, MacKay and Neal [9], [10] re-discovered LDPC codes. Chung *et al.* [11]

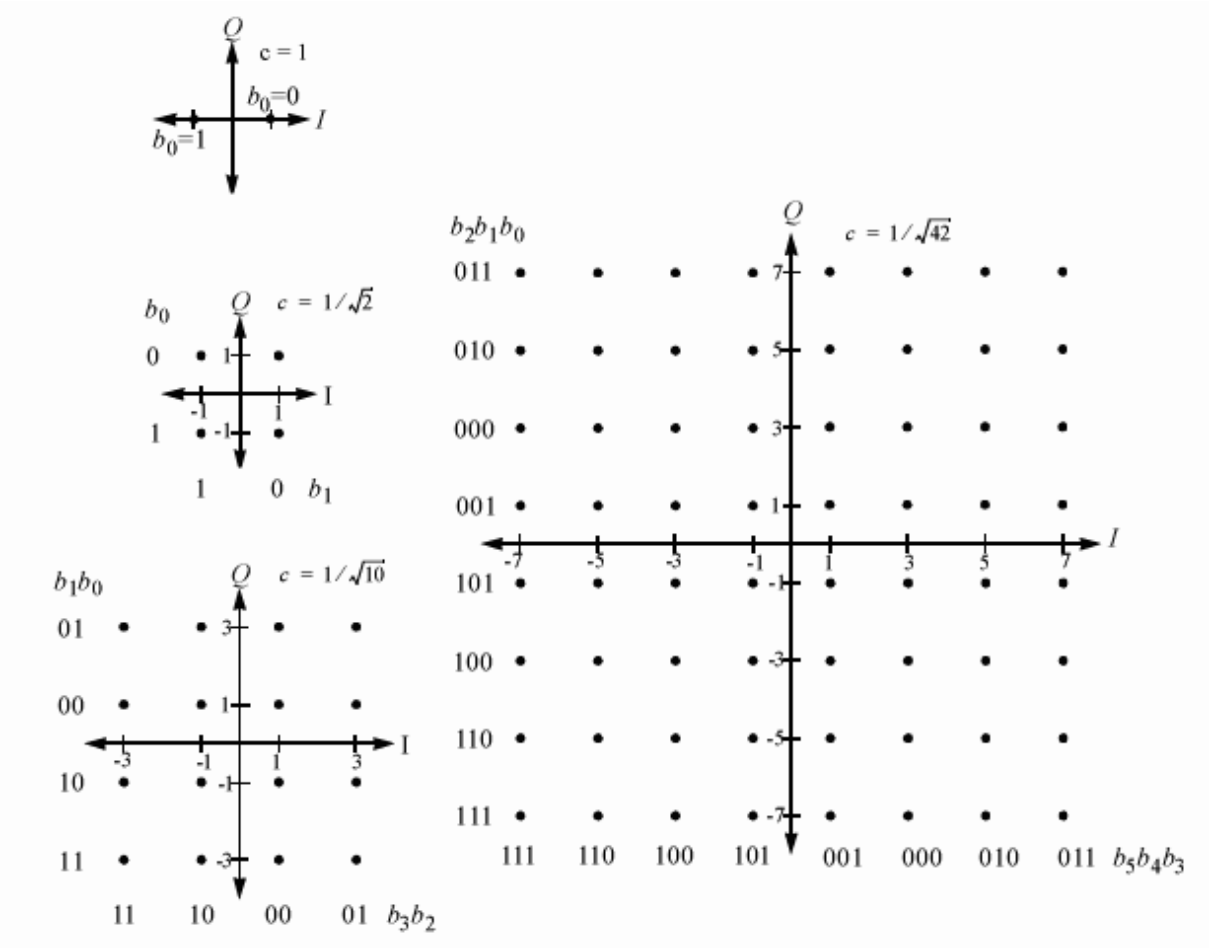


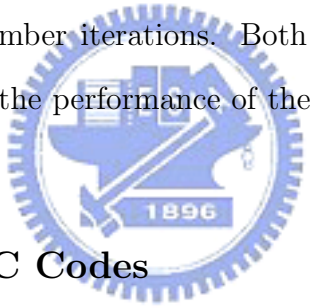
Figure 2.4: BPSK, QPSK, 16-QAM, and 64-QAM constellations (from [1]).

showed that a rate-1/2 LDPC code with block length of 10^7 in binary input additive white Gaussian noise (AWGN) can achieve a threshold of just 0.0045 dB away from Shannon limit.

LDPC codes have several advantages over turbo codes: First, the sum-product decoding algorithm for these codes has inherent parallelism which can be harvested to achieve a greater speed of decoding. Second, unlike turbo codes, decoding error is a detectable event which results in a more reliable system. Third, very low complexity decoders, such as the modified minimum-sum algorithm that closely approximate the sum-product in performance, can be designed for these codes.

Since our focus is on wireless communications, we would like to have low-power architectures and speed of decoding as it is needed for the IEEE 802.16e standard.

Complexity in iterative decoding has two parts. First, complexity of the computations in each iteration. Second, the number iterations. Both of these are manageable in practice. There is a trade-off between the performance of the decoder, complexity and speed of decoding.



2.2.1 Overview of LDPC Codes

LDPC codes are a class of linear block codes corresponding to a sparse parity check matrix H. The term "low-density" means that the number of 1s in each row or column of H is small compared to the block length n. In other words, the density of 1s in the parity check matrix which consists of only 0s and 1s is very low and sparse. Given k information bits, the set of LDPC codewords c in the code space C of length n spans the null space of the parity check matrix H in which $cH^T = 0$.

For a (W_c, W_r) LDPC code, each column of the parity check matrix H has W_c ones and each row has W_r ones; this is called *regular*. If degrees per row or column are not constant, then the code is *irregular*. Some of the irregular codes have shown better performance than

regular ones. But irregularity results in more complex hardware and inefficiency in terms of re-usability of functional units. In the IEEE 802.16e standard irregular codes have been considered to achieve better performance. Code rate R is equal to k/n, which means that $n-k$ redundant bits have been added to the message so as to correct the errors.

LDPC codes can be represented effectively by a bipartite graph called a Tanner graph [12], [13]. A bi-partite graph is a graph (nodes or vertices are connected by undirected edges) whose nodes may be separated into two classes, and where edges may only be connecting two nodes not residing in the same class. The two classes of nodes in a Tanner graph are bit nodes and check nodes. The Tanner graph of a code is drawn according to the following rule: Check node f_j , $j = 1, \cdots, n-k$, is connected to bit node x_i, $i = 1, \cdots, n$, whenever element h_{ji} in H (parity check matrix) is a *one*. Figure. 2.5 shows a Tanner graph made for a simple parity check matrix H. In this graph each bit node is connected to *two* check nodes (bit degree = 2) and each check node has a degree of *four*.

Let $d_{v_{max}}$ and $d_{c_{max}}$ denote the maximum variable node and check node degree respectively, and let λ_i and ρ_i represent the fraction of edges emanating from variable and check nodes of degree and $d(v) = i$ and $d(c) = i$ respectively. Then we can define

$$\lambda(x) = \sum_{i=2}^{d_{v_{max}}} \lambda_i x^{i-1} \tag{2.14}$$

as the variable node degree distribution, and

$$\rho(x) = \sum_{i=2}^{d_{c_{max}}} \rho_i x^{i-1} \tag{2.15}$$

as the check node degree distribution.

Definition: Degree of a node is the number of branches that is connected to that node.

Definition: A cycle of length l in a Tanner graph is a path comprised of l edges which closes back on itself. The Tanner graph in Fig. 2.5 has a cycle of length four which has been shown by dashed lines.

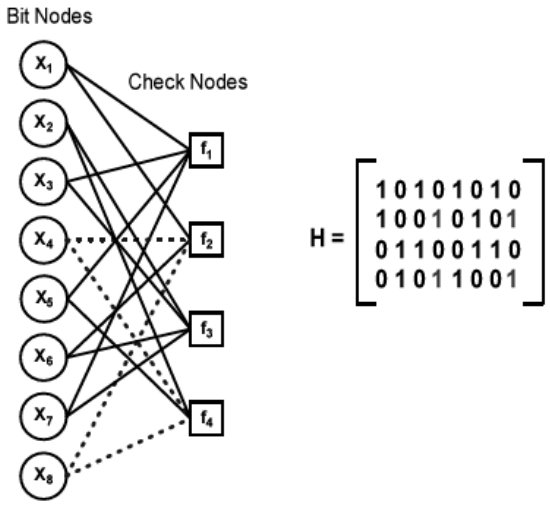


Figure 2.5: Tanner graph of a parity check matrix (from [7]).

Definition: The girth of a Tanner graph is the minimum cycle length of the graph. The shortest possible cycle in a bi-partite graph is clearly a length-4 cycle.

Short cycles have negative impact on the decoding performance of LDPC codes. Hence we would like to have large girths.

2.2.2 LDPC Codes Specification in IEEE 802.16e OFDMA

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

The LDPC codes in IEEE802.16e are defined based on a parity check matrix H of size $m \times n$ that is expanded from a binary base matrix H_b of 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 depicted in Fig. 2.6, two different ones for two rate 2/3 codes, type A in Fig. 2.7 and type B in Fig. 2.8, two different ones for two rate 3/4 codes, type A in Fig. 2.9 and type B in Fig.

Rate 1/2:

-1	94	73	-1	-1	-1	-1	-1	55	83	-1	-1	7	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	27	-1	-1	-1	22	79	9	-1	-1	-1	12	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	24	22	81	-1	33	-1	-1	-1	0	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1
61	-1	47	-1	-1	-1	-1	-1	65	25	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1
-1	-1	39	-1	-1	-1	84	-1	-1	41	72	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	46	40	-1	82	-1	-1	-1	79	0	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1
-1	-1	95	53	-1	-1	-1	-1	-1	14	18	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1
-1	11	73	-1	-1	-1	2	-1	-1	47	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1
12	-1	-1	-1	83	24	-1	43	-1	-1	-1	51	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1
-1	-1	-1	-1	-1	94	-1	59	-1	-1	70	72	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1
-1	-1	7	65	-1	-1	-1	-1	39	49	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0
43	-1	-1	-1	-1	66	-1	41	-1	-1	-1	26	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

Figure 2.6: Base model of the rate-1/2*code* (from [2]).

2.10, one for the rate 5/6 code depicted in Fig. 2.11. 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 $n_{min} = 24\times24 = 576$ bits and the maximum code length as $n_{max} = 24\times96 = 2304$ bits.

For codes $\frac{1}{2}$, $\frac{2}{3}$B, $\frac{3}{4}$A, $\frac{3}{4}$B, and $\frac{5}{6}$, the shift sizes $p(f,i,j)$ for a code size corresponding to 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

$$p(f,i,j) = \begin{cases} p(i,j), & p(i,j) \leq 0, \\ \lfloor \frac{p(i,j)z_f}{z_o} \rfloor, & p(i,j) > 0. \end{cases} \tag{2.16}$$

For code $\frac{2}{3}$A, the shift sizes $p(f,i,j)$ are derived by using a modulo function as

$$p(f,i,j) = \begin{cases} p(i,j), & p(i,j) \leq 0, \\ mod(p(i,j), z_f), & p(i,j) > 0. \end{cases} \tag{2.17}$$

A base matrix entry $p(f,i,j) = -1$ indicates a replacement with a $z \times z$ all-zero matrix and an entry $p(f,i,j) \geq 0$ indicates a replacement with a $z\times z$ permutation matrix. The permutation matrix represents a circular right shift of $p(f,i,j)$ positions. This entry $p(f,i,j)$ = 0 indicates a $z\times z$ identity matrix.

Rate 2/3 A code:

3	0	-1	-1	2	0	-1	3	7	-1	1	1	-1	-1	-1	-1	1	0	-1	-1	-1	-1	-1	-1
-1	-1	1	-1	36	-1	-1	34	10	-1	-1	18	2	-1	3	0	-1	0	0	-1	-1	-1	-1	-1
-1	-1	12	2	-1	15	-1	40	-1	3	-1	15	-1	2	13	-1	-1	-1	0	0	-1	-1	-1	-1
-1	-1	19	24	-1	3	0	-1	6	-1	17	-1	-1	-1	8	39	-1	-1	-1	0	0	-1	-1	-1
20	-1	6	-1	-1	10	29	-1	-1	28	-1	14	-1	38	-1	-1	0	-1	-1	-1	0	0	-1	-1
-1	-1	10	-1	28	20	-1	-1	8	-1	36	-1	9	-1	21	45	-1	-1	-1	-1	-1	0	0	-1
35	25	-1	37	-1	21	-1	-1	5	-1	-1	0	-1	4	20	-1	-1	-1	-1	-1	-1	-1	0	0
-1	6	6	-1	-1	-1	4	-1	14	30	-1	3	36	-1	14	-1	1	-1	-1	-1	-1	-1	-1	0

Figure 2.7: Base model of the rate-2/3, type A code(from [2]).

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	47	-1	13	-1	61	-1	84	-1	55	-1	78	-1	41	95	-1	-1	-1	-1	-1	-1	0

Figure 2.8: Base model of the rate-2/3, type B code(from [2]).

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 2.9: Base model of the rate-3/4, type A code(from [2]).

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	-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 2.10: Base model of the rate-3/4, type B code(from [2]).

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 2.11: Base model of the rate-5/6 code(from [2]).

Table 2.4: Bit Interleaved Block Sizes and Modulos

Modulation	Coded Bits per Carrier (N_{cpc})	Modulo Used (d)
QPSK	2	16
16QAM	4	16
64QAM	6	16

2.2.3 Interleaver

The encoded data bits are interleaved by a block interleaver with a block size corresponding to the number of coded bits per the encoded block size, N_{cbps} (see Table 2.3). The interleaver is defined by a two-step permutation. The first ensures that adjacent coded bits are mapped onto non-adjacent carriers. The second insures that adjacent coded bits are mapped alternately onto less or more significant bits of the constellation, thus avoiding long runs of lowly reliable bits.

Let $s = N_{cpc}/2$, k be the index of the coded bit before the first permutation, m the index after the first and before the second permutation and j the index after the second permutation, just prior to modulation mapping. The first permutation is defined by

$$m = (\frac{N_{cbps}}{d}) \cdot \ k_{mod(d)} + floor(\frac{k}{d}), \quad k = 0, 1, \cdots, N_{cbps} - 1, \tag{2.18}$$

and the second permutation by

$$j = s \cdot \ floor(\frac{m}{s}) + (m + N_{cbps} - floor(\frac{d \cdot m}{N_{cbps}}))_{mod(s)}, \quad m = 0, 1, \cdots, N_{cbps} - 1. \tag{2.19}$$

The de-interleaver, which performs the inverse operation, is also defined by two permutations. Let j be the index of the received bit before the first permutation, m be the index after the first and before the second permutation, and k be the index after the second

permutation, just prior to delivering the coded bits to the convolutional decoder. The first permutation is defined by

$$m = s \cdot floor(\frac{j}{s}) + (j + floor(\frac{d \cdot j}{N_{cbps}}))_{mod(s)}, \quad j = 0, 1, \cdots, N_{cbps} - 1, \tag{2.20}$$

and the second permutation by

$$k = d \cdot m - (N_{cbps} - 1) \cdot floor(\frac{d \cdot m}{N_{cbps}}), \quad m = 0, 1, \cdots, N_{cbps} - 1. \tag{2.21}$$

2.2.4 Modulation

After bit interleaving, the data bits are entered serially to the constellation mapper. QPSK and Gray-mapped 16-QAM are supported, whereas the support of Gray-mapped 64-QAM is optional. The constellations as shown in Fig. 2.12 shall be normalized by multiplying the constellation points with the indicated factor c to achieve equal average power. The constellation-mapped data shall be subsequently modulated onto the allocated data carriers.

2.3 Analysis of LDPC Codes in IEEE 802.16e OFDMA

2.3.1 Girth Analysis

In this section, we compute the girth of LDPC in IEEE 802.16e for all kinds of code rate. Hence we can broadbrush estimate the specific code performance. The result is listed in Table 2.5.

From Table 2.5, we can roughly estimate the performance of code rate $\frac{2}{3}A$ is a little better than $\frac{2}{3}B$ under the same condition of codeword length, modulation, channel, and decoding algorithm, because of the longer average girth. While code rate $\frac{3}{4}B$ would perform slightly better than rate $\frac{3}{4}A$ by the same reason described above.

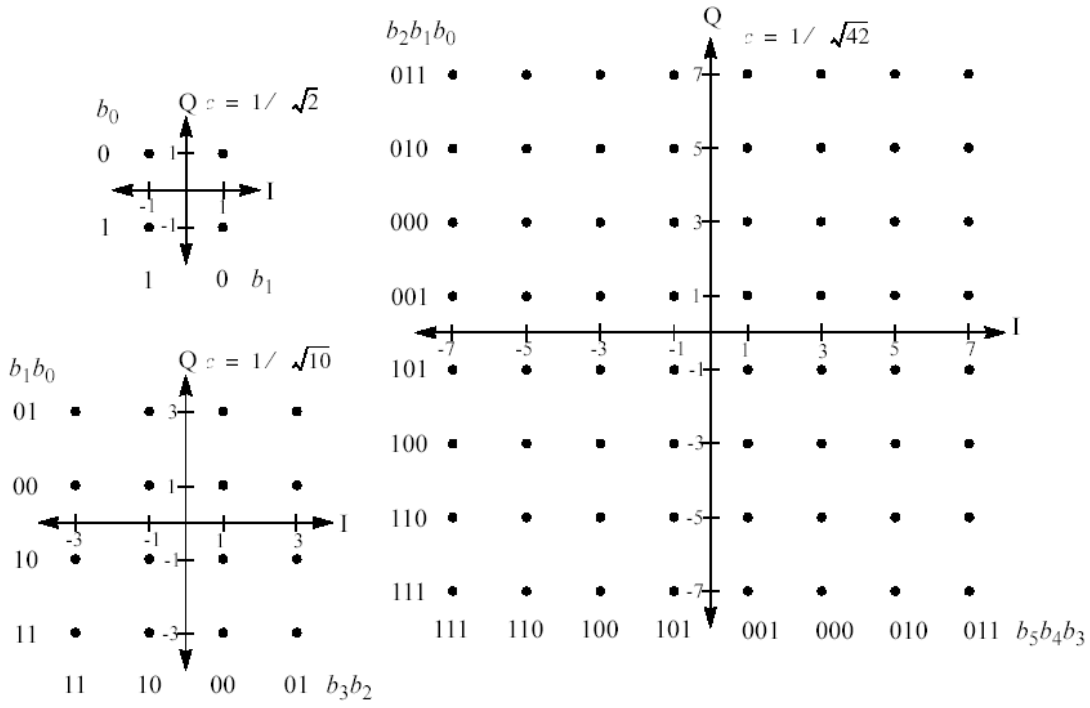


Figure 2.12: QPSK, 16-QAM, and 64-QAM constellations (from [2]).

Table 2.5: Girths of LDPC Codes in IEEE 802.16e OFDMA

Codeword Length (Bits)	$\frac{1}{2}$	$\frac{2}{3}A$	$\frac{2}{3}B$	$\frac{3}{4}A$	$\frac{3}{4}B$	$\frac{5}{6}$
576	6	6	6	4	6	6
672	4	6	4	4	4	4
768	6	6	4	4	4	4
864	6	6	4	4	4	4
960	6	6	4	4	4	4
1056	6	6	4	4	4	4
1152	6	6	4	4	4	4
1248	6	6	4	4	4	4
1344	6	6	4	4	4	4
1440	6	6	4	4	4	4
1536	6	6	4	4	4	4
1632	6	6	4	4	4	4
1728	6	6	4	4	4	4
1824	6	6	4	4	4	4
1920	6	6	4	4	4	4
2016	6	6	4	4	4	4
2112	6	6	4	4	4	4
2208	6	6	4	4	4	4
2304	6	6	4	4	4	4

2.3.2 Density Evolution

For many channels and iterative decoders of interest, LDPC codes exhibit a *threshold* phenomenon [14]: as the block length tends to infinity, an arbitrarily small bit error probability can be achieved if the noise level is smaller than a certain threshold. For a noise level above this threshold, on the other hand, the probability of bit error is larger than a positive constant.

Density evolution provides an efficient way to determine the thresholds of LDPC codes ensemble by tracking the probability density functions (pdf's) of the message in the Tanner graph of an LDPC code. Since there is no theoretical guideline. for the design of LDPC codes, it is meaningful to optimize the code by density evolution [15].

Without loss of generality, assume that the all-0 codeword is transmitted. Firstly, choose a value for the threshold parameter δ to start density evolution. If the pdf of all bit messages tend to infinity after enough iterations, such a value of δ is within the threshold. Then, increase the value δ until the density evolution cannot succeed, that is, the pdf cannot tend to infinity after enough iterations. The maximum value of δ found is the *threshold* of the irregular LDPC codes with degree distribution pair (λ, ρ).

In Table 2.6, we list the degree distribution pairs(λ, ρ) and the thresholds of the LDPC codes in IEEE 802.16e OFDMA. Here we assume BPSK modulation, belief propagation (BP) decoding which will be introduced in Chapter 5, and AWGN channel.

From these threshold values, not only $\frac{2}{3}A$ is larger than $\frac{2}{3}B$ but $\frac{3}{4}B$ is larger than $\frac{3}{4}A$, this result is the same as what we broadbrush estimate the specific code performance by using the girth analysis.

Table 2.6: Degree Distribution and Threshold for Each Code Rate under BPSK Modulation, AWGN Channel, and BP Decoding

Code Rate	Bit Node Degree Distribution	Check Node Degree Distribution	Threshold
$1/2$	$0.2895x + 0.3158x^2 + 0.3947x^3$	$0.6315x^5 + 0.3685x^6$	0.9273
$2/3A$	$0.175x + 0.45x^2 + 0.375x^5$	x^9	0.7282
$2/3B$	$0.1729x + 0.037x^2 + 0.7901x^3$	$0.8642x^9 + 0.1358x^{10}$	0.7163
$3/4A$	$0.1176x + 0.0353x^2 + 0.8471x^3$	$0.8235x^{13} + 0.1765x^{14}$	0.6358
$3/4B$	$0.1137x + 0.409x^2 + 0.4773x^5$	$0.3182x^{13} + 0.6818x^{14}$	0.6446
$5/6$	$0.075x + 0.375x^2 + 0.55x^3$	x^{19}	0.5607

Chapter 3

DSP Implementation Environment

We conduct a DSP (digital signal processor) implementation for the channel coding scheme of OFDM in our work. In this we employ the Quixote DSP-FPGA baseboard made by Innovative Integration (II), on which the DSP is Texas Instruments's (TI) TMS320C6416. Because of our purely software implementation on the DSP, discussion in this chapter will mainly focus on the DSP chip and the associated system development environment.

3.1 The TMS320C6416 DSP Chip

The following text is mainly taken from references [16] and [17].

3.1.1 TMS320C6416 Features

The TMS320C64x DSPs are the highest-performance fixed-point DSP generation on the TMS320C6000 DSP platform. The TMS320C64x device is based on the second-generation high-performance, very-long-instruction-word (VLIW) architecture developed by TI. The C6416 device has two high-performance embedded coprocessors, Viterbi Decoder Coprocessor (VCP) and Turbo Decoder Coprocessor (TCP) that can significantly speed up channel-decoding operations on-chip, but we do not make use of these coprocessors in the present

work.

The C64x core CPU consists of 64 general-purpose 32-bits registers and 8 function units. Features of C6000 devices include:

- The eight functional units include two multipliers and six arithmetic units:
  - Execute up to eight instructions per cycle.
  - Allow designers to develop highly effective RISC-like code for fast development time.
- Instruction packing:
  - Gives code size equivalence for eight instructions executed serially or in parallel.
  - Reduces code size, program fetches, and power consumption.
- Conditional execution of all instructions:
  - Reduces costly branching.
  - Increases parallelism for higher sustained performance.
- Efficient code execution on independent functional units:
  - Efficient C compiler on DSP benchmark suite.
  - Assembly optimizer for fast development and improved parallelization.
- 8/16/32-bit data support, providing efficient memory support for a variety of applications.
- 40-bit arithmetic options add extra precision for applications requiring it.
- Saturation and normalization provide support for key arithmetic operations.

- Field manipulation and instruction extract, set, clear, and bit counting support common operation found in control and data manipulation applications.

The C64x additional features include:

- Each multiplier can perform two 16×16 bits or four 8×8 bits multiplies every clock cycle.
- Quad 8-bit and dual 16-bit instruction set extensions with data flow support.
- Support for non-aligned 32-bit (word) and 64-bit (double word) memory accesses.
- Special communication-specific instructions have been added to address common operations in error-correcting codes.
- Bit count and rotate hardware extends support for bit-level algorithms.

3.1.2 Central Processing Unit Features [18]

The block diagram of C6416 DSP is shown in Fig. 3.1. The DSP contains: program fetch unit, instruction dispatch unit, instruction decode unit, two data paths which each has four functional units, 64 32-bit registers, control registers, control logic, and logic for test, emulation, and interrupt logic.

The TMS320C64x DSP pipeline provides flexibility to simplify programming and improve performance. The pipeline can dispatch eight parallel instructions every cycle. The following two factors provide this flexibility: Control of the pipeline is simplified by eliminating pipeline interlocks, and the other is increasing pipelining to eliminate traditional architectural bottlenecks in program fetch, data access, and multiply operations. This provides single cycle throughput.

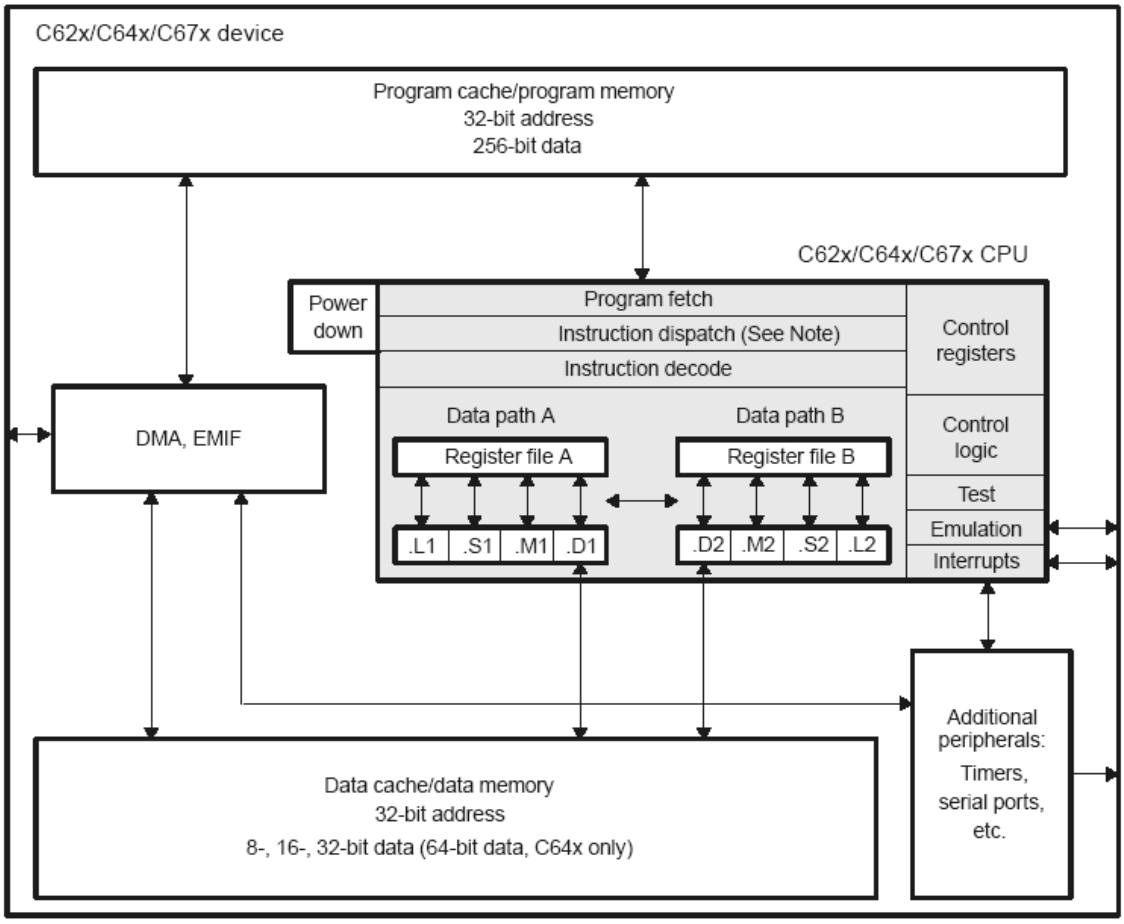


Figure 3.1: Block diagram of TMS320C6416 DSP (from [18]).

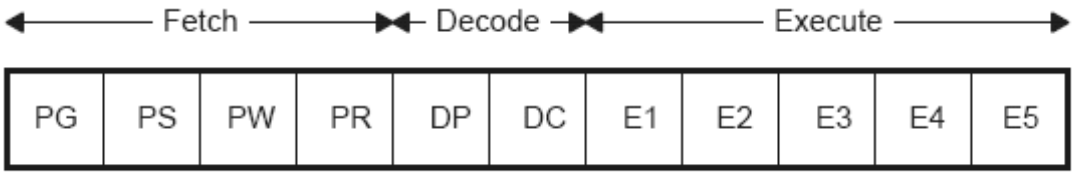


Figure 3.2: Pipeline phases of TMS320C6416 DSP (from [18]).

The pipeline phases are divided into three stages: fetch, decode, and execute. All instructions in the C62x/C64x instruction set flow through the fetch, decode, and execute stages of the pipeline. The fetch stage of the pipeline has four phases for all instructions, and the decode stage has two phases for all instructions. The execute stage of the pipeline requires a varying number of phases, depending on the type of instruction. The stages of the C62x/C64x pipeline are shown in Fig. 3.2.

Reference [18] contains detailed information regarding the fetch and decode phases. The pipeline operation of the C62x/C64x instructions can be categorized into seven instruction types. Six of these are shown in Table 3.1, which gives a mapping of operations occurring in each execution phase for the different instruction types. The delay slots associated with each instruction type are listed in the bottom row.

The execution of instructions can be defined in terms of delay slots. A delay slot is a CPU cycle that occurs after the first execution phase (E1) of an instruction. Results from instructions with delay slots are not available until the end of the last delay slot. For example, a multiply instruction has one delay slot, which means that one CPU cycle elapses before the results of the multiply are available for use by a subsequent instruction. However, results are available from other instructions finishing execution during the same CPU cycle in which the multiply is in a delay slot.

The eight functional units in the C6000 data paths can be divided into two groups of

Table 3.1: Execution Stage Length Description for Each Instruction Type (from [18]).

		Instruction Type					
		Single Cycle	**16 X 16 Single Multiply/ C64x .M Unit Non-Multiply**	**Store**	**C64x Multiply Extensions**	**Load**	**Branch**
Execution phases	E1	Compute result and write to register	Read operands and start computations	Compute address	Reads operands and start computations	Compute address	Target-code in PG‡
	E2		Compute result and write to register	Send address and data to memory		Send address to memory	
	E3			Access memory		Access memory	
	E4				Write results to register	Send data back to CPU	
	E5					Write data into register	
Delay slots		0	1	0†	3	4†	5‡

four; each functional unit in one data path is almost identical to the corresponding unit in the other data path. The functional units are described in Table 3.2.

Besides being able to perform 32-bit operations, the C64x also contains many 8-bit and 16-bit extensions to the instruction set. For example, the MPYU4 instruction performs four 8×8 unsigned multiplies with a single instruction on a .M unit. The ADD4 instruction performs four 8-bit additions with a single instruction on a .L unit.

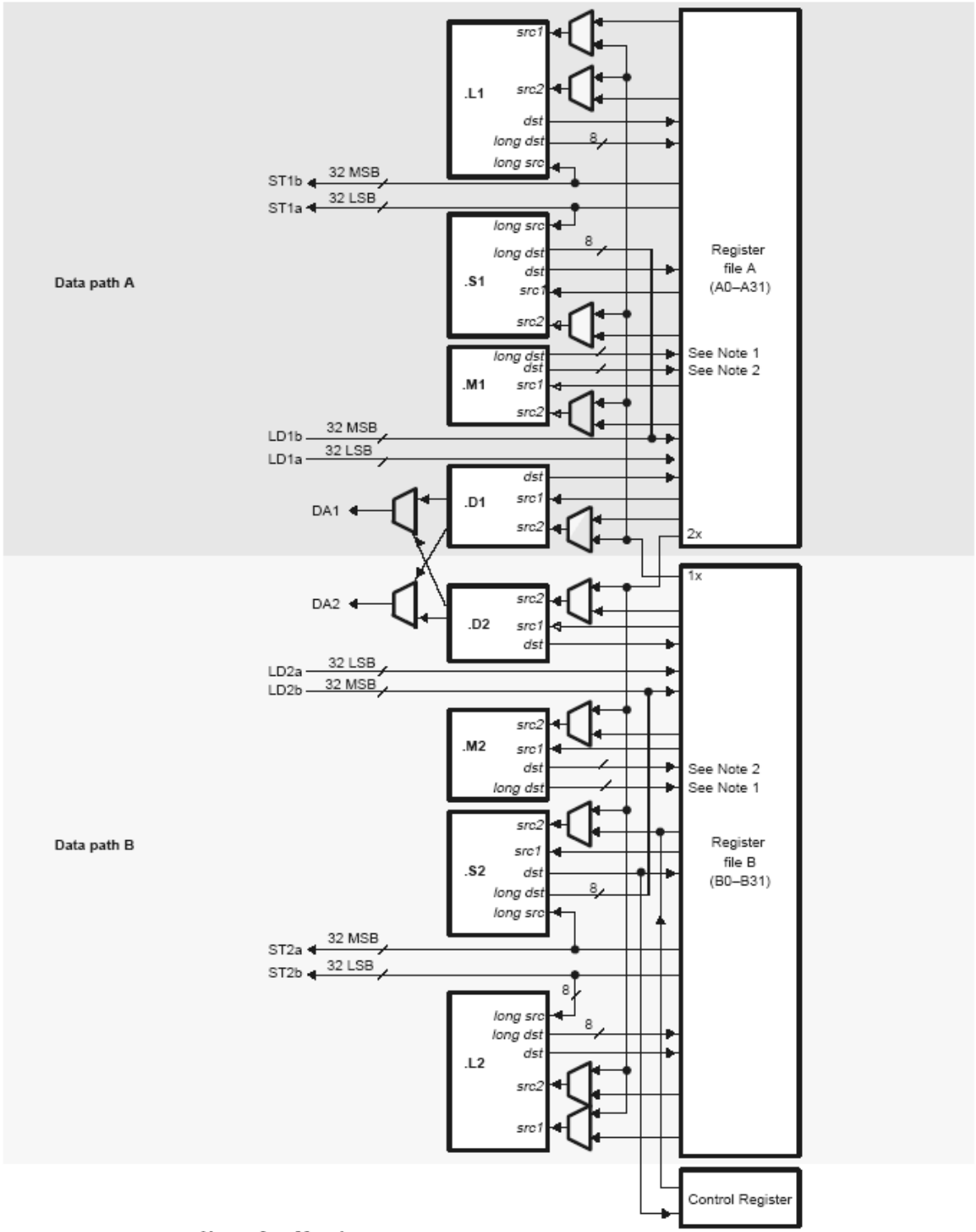
The data line in the CPU supports 32-bit operands, long (40-bit) and double word (64-bit) operands. Each functional unit has its own 32-bit write port into a general-purpose register file (see Fig. 3.3). All units ending in 1 (for example, .L1) write to register file A, and all units ending in 2 write to register file B. Each functional unit has two 32-bit read ports for source operands src1 and src2. Four units (.L1, .L2, .S1, and .S2) have an extra 8-bit-wide port for 40-bit long writes, as well as an 8-bit input for 40-bit long reads. Because each unit has its own 32-bit write port, when performing 32-bit operations all eight units can be used in parallel every cycle.

3.1.3 Cache Memory Architecture Overview [19]

The C64x memory architecture consists of a two-level internal cache-based memory architecture plus external memory. Level 1 cache is split into program (L1P) and data (L1D) caches. The C64x memory architecture is shown in Fig. 3.4. On C64x devices, each L1 cache is 16 kB. All caches and data paths are automatically managed by cache controller. Level 1 cache is accessed by the CPU without stalls. Level 2 cache is configurable and can be split into L2 SRAM (addressable on-chip memory) and L2 cache for caching external memory locations. On a C6416 DSP, the size of L2 cache is 1 MB, and the external memory on Quixote baseboard is 32 MB. More detailed introduction to the cache system can be found in [19].

Table 3.2: Functional Units and Operations Performed (from [18])

Function Unit	Operations
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits Byte shifts Data packing/unpacking 5-bit constant generation Dual 16-bit arithmetic operations Quad 8-bit arithmetic operations Dual 16-bit min/max operations Quad 8-bit min/max operations
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from control register file (.S2 only) Byte shifts Data packing/unpacking Dual 16-bit compare operations Quad 8-bit compare operations Dual 16-bit shift operations Dual 16-bit saturated arithmetic operations Quad 8-bit saturated arithmetic operations
.M unit (.M1, .M2)	16 x 16 multiply operations 16 x 32 multiply operations Quad 8 x 8 multiply operations Dual 16 x 16 multiply operations Dual 16 x 16 multiply with add/subtract operations Quad 8 x 8 multiply with add operation Bit expansion Bit interleaving/de-interleaving Variable shift operations and rotation Galois Field Multiply
.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with 5-bit constant offset Loads and stores with 15-bit constant offset (.D2 only) Load and store double words with 5-bit constant Load and store non-aligned words and double words 5-bit constant generation 32-bit logical operations



Notes for .M unit:
1. *long dst* is 32 MSB
2. *dst* is 32 LSB

Figure 3.3: TMS320C64x CPU data paths (from [18]).

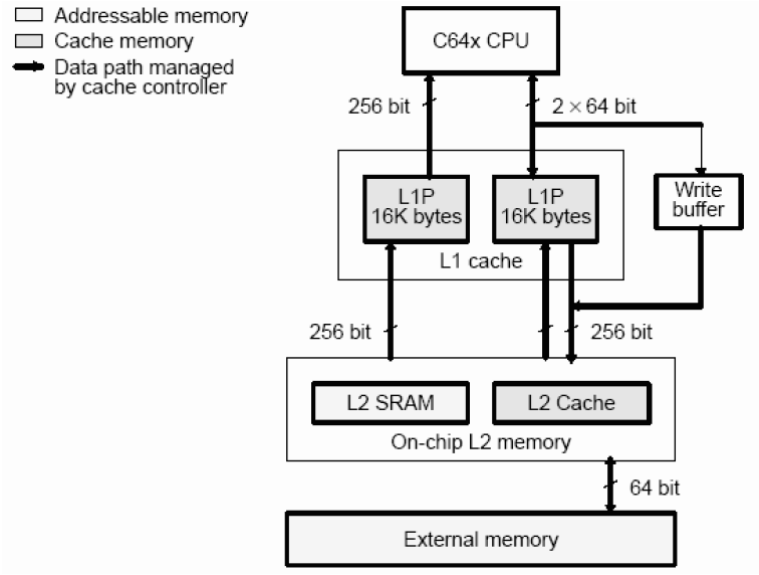


Figure 3.4: C64x cache memory architecture (from [19]).

3.2 The Quixote Baseboard [20]

The DSP-FPGA embedded card used in our implementation is Innovative Integration's (II) Quixote baseboard, which is illustrated in Fig. 3.5. Quixote is one of II's Velocia-family baseboards for various applications requiring high-speed computation. Figure. 3.6 shows a block diagram of the Quixote board. It combines a 600 MHz C6416 32-bit fixed-point DSP with a Virtex-II FPGA, and some system-level peripherals. The FPGAs on our boards are the six-million-gate version. The TI C6416 DSP operating at 600 MHz offers a processing power of 4800 MIPS. Some detailed features of the board are as follows:

- TMS320C6416 processor running at frequency up to 600 MHz.
- Onboard 32 MB SDRAM for the DSP chip.
- A 32/64 bits PCI bus host interface with direct host memory access capability for busmastering data between the card and the memory.

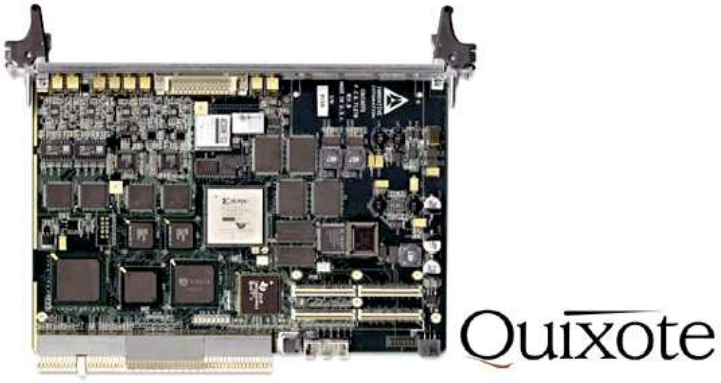


Figure 3.5: Picture of the Quixote board [20].

3.3 TI's Code Development Environment [21], [22]

TI provides a useful GUI development interface to DSP users for developing and debugging their projects: Code Composer Studio (CCS). The CCS development tools are a key element of the DSP software and development tools from Texas Instruments. The fully integrated development environment includes real-time analysis capabilities, easy to use debugger, C/C++ compiler, assembler, linker, editor, visual project manager, simulators, XDS560 and XDS510 emulation drivers and DSP/BIOS support.

Some of CCS's fully integrated host tools include:

- Simulators for full devices, CPU only and CPU plus memory for optimal performance.
- Integrated visual project manager with source control interface, multi-project support and the ability to handle thousands of project files.
- Source code debugger common interface for both simulator and emulator targets:
  - C/C++/assembly language support.
  - Simple breakpoints.

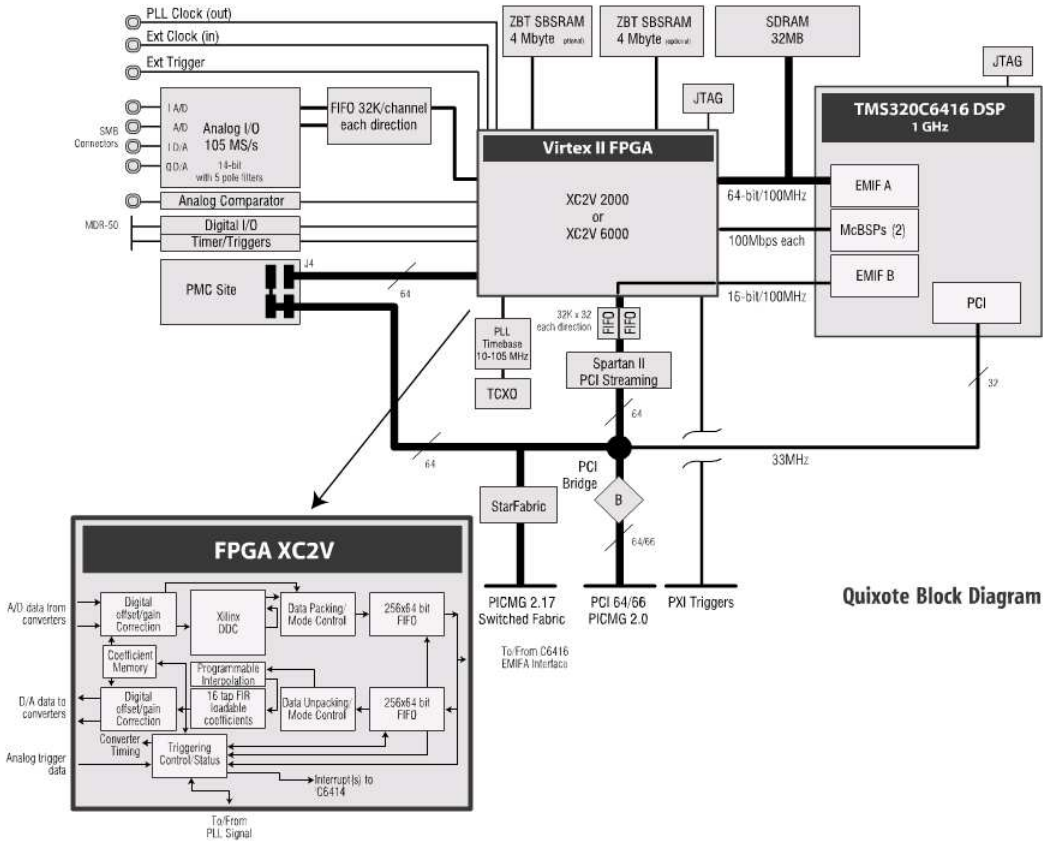


Figure 3.6: Block diagram of the Quixote board (from [16]).

  - Advanced watch window.
  - Symbol browser.
- DSP/BIOS host tooling support (configure, real-time analysis and debug).
- Data transfer for real time data exchange between host and target.
- Profiler to understand code performance.

CCS also delivers foundation software consisting of:

- DSP/BIOS kernel for the TMS320C6000 DSPs:
  - Pre-emptive multi-threading.
  - Interthread communication.
  - Interupt Handling.
- TMS320 DSP Algorithm Standard to enable software reuse.
- Chip Support Libraries (CSL) to simplify device configuration. CSL provides C-program functions to configure and control on-chip peripherals.
- DSP libraries for optimum DSP functionality. The libraries include many C-callable, assembly-optimized, general-purpose signal-processing and image/video processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical.

The DSP Library (DSPLIB) for TMS320C64x includes routines that are organized into seven groups:

- Adaptive filtering.

- Correlation.
- FFT.
- Filtering and convolution.
- Math.
- Matrix functions.
- Miscellaneous.

3.4 Code Development Flow [23]

The recommended code development flow involves utilizing the C6000 code generation tools to aid in optimization rather than forcing the programmer to code by hand in assembly. These advantages allow the compiler to do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation. These features simplify the maintenance of the code, as everything resides in a C framework that is simple to maintain, support, and upgrade.

The recommended code development flow for the C6000 involves the phases described in Fig. 3.7. The tutorial section of the Programmers Guide [23] focuses on phases 1–2 and the Guide also instructs the programmer when to go to the tuning stage of phase 3. What is learned is the importance of giving the compiler enough information to fully maximize its potential. An added advantage is that this compiler provides direct feedback on the entire program's high MIPS areas (loops). Based on this feedback, there are some very simple steps the programmer can take to pass complete and better information to the compiler allowing the programmer a quicker start in maximizing compiler performance. The following items list the goal for each phase in the 3-phase software development flow shown in Fig. 3.7.

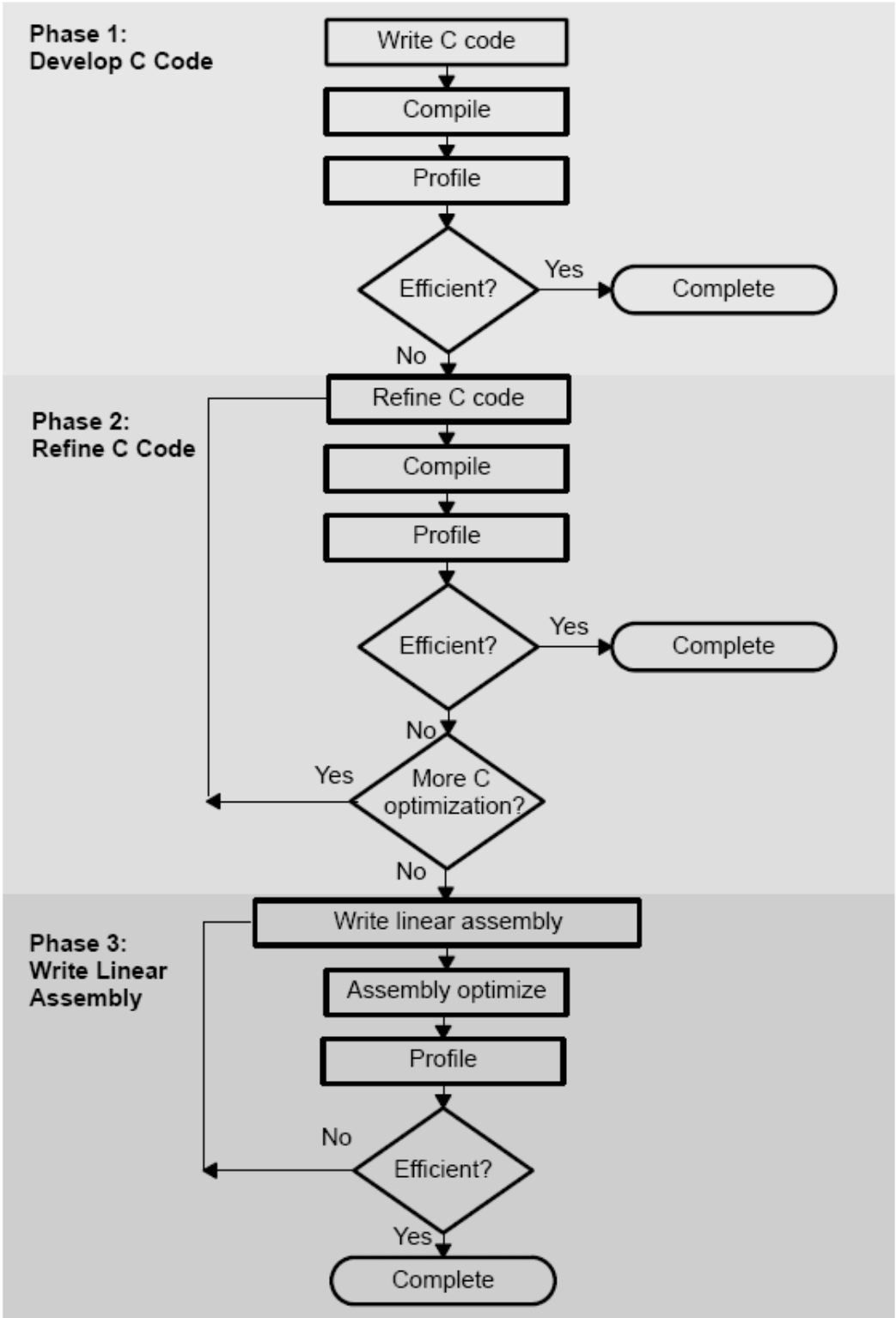


Figure 3.7: Code development flow for TI C6000 DSP (from [23]).

- Developing C code (phase 1) without any knowledge of the C6000. Use the C6000 profiling tools to identify any inefficient areas that we might have in the C code. To improve the performance of the code, proceed to phase 2.
- Use techniques described in [23] to improve the C code. Use the C6000 profiling tools to check its performance. If the code is still not as efficient as we would like it to be, proceed to phase 3.
- Extract the time-critical areas from the C code and rewrite the code in linear assembly. We can use the assembly optimizer to optimize this code.

TI provides high performance C program optimization tools, and they do not suggest the programmer to code by hand in assembly. In this thesis, the development flow is stopped at phase 2. We do not optimize the code by writing linear assembly. Coding the program in high level language keeps the flexibility of porting to other platforms.

3.4.1 Compiler Optimization Options [23]

The compiler supports several options to optimize the code. The compiler options can be used to optimize code size or execution performance. Our primary concern in this work is the execution performance. The easiest way to invoke optimization is to use the cl6x shell program, specifying the -o*n* option on the cl6x command line, where *n* denotes the level of optimization (0, 1, 2, 3) which controls the type and degree of optimization:

- -o0:
  - Performs control-flow-graph simplification.
  - Allocates variables to registers.
  - Performs loop rotation.

  - Eliminates unused code.
  - Simplifies expressions and statements.
  - Expands calls to functions declared inline.

- -o1. Performs all -o0 optimization, and:
  - Performs local copy/constant propagation.
  - Removes unused assignments.
  - Eliminates local common expressions.

- -o2. Performs all -o1 optimizations, and:
  - Performs software pipelining.
  - Performs loop optimizations.
  - Eliminates global common subexpressions.
  - Eliminates global unused assignments.
  - Converts array references in loops to incremented pointer form.
  - Performs loop unrolling.

- -o3. Performs all -o2 optimizations, and:
  - Removes all functions that are never called.
  - Simplifies functions with return values that are never used.
  - Inline calls to small functions.
  - Reorders function declarations so that the attributes of called functions are known when the caller is optimized.

– Propagates arguments into function bodies when all calls pass the same value in the same argument position.

– Identifies file-level variable characteristics.

Chapter 4

Implementation and Optimization of IEEE 802.16e OFDM Channel Codec on DSP

In this chapter, we discuss the decoding algorithms of the IEEE 8021.16e OFDM channel codec on DSP. Our DSP is a TI TMS320C6416 chip, housed on II's Quixote baseboard. We base our implementation on modification of the code of Lee [24] for IEEE 802.16a OFDMA to the specifications of IEEE 802.16e OFDM. We present the performance results obtained from the profiler generated by the built-in profiler in TI's Code Composer Studio (CCS) tool set.

4.1 Decoding of RS Code [5]

The Berlekamp-Massey (BM) algorithm is a common decoding algorithm for RS codes [25]. It includes four steps:

1. Compute the syndrome value.
2. Compute the error location polynomial.
3. Compute the error location.

4. Compute the error value.

Under the unable-to-correct condition (e.g., errors number greater than T'), the received word will not be dealt with.

The shortening does not affect the RS decoder because the RS code in IEEE802.16e is a systematic code and the initial zero bytes will not affect each step of the decoder. As for the puncturing, the punctured bytes can be viewed as erasures. Thus the decoder we adopt should be able to correct erasures [25].

4.2 Viterbi Decoding of Punctured Convolutional Code

Viterbi algorithm is the most well-known technique for the convolutional decoding process. The operation of Viterbi algorithm can be explained by the trellis diagram, which is provided by the CC encoder structure. The concept of the trellis diagram is based on the state transition diagram. Hence, we can expand the state transition diagram to a trellis diagram. The trellis diagram is consistent with all the features of finite state machine and can be regarded as the time axis expansion of the finite state machine. A simple trellis diagram is shown in Fig. 4.1 as an example. In this trellis diagram, the upper outgoing branch for each state corresponds to an input of 0, whereas the lower outgoing branch corresponds to an input of 1. Each state has two incoming and two outgoing branches. Each information sequence, uniquely encoded into an encoded sequence, corresponds to a unique path in the trellis. Therefore, for a given path through the trellis, we can obtain the corresponding information sequence by reading off the input labels on all the branches that make up the path. The procedure is called "traceback".

Viterbi algorithm operates by computing the branch metric for each path at each stage of the trellis. The metric is calculated and stored as a partial metric for each branch as the

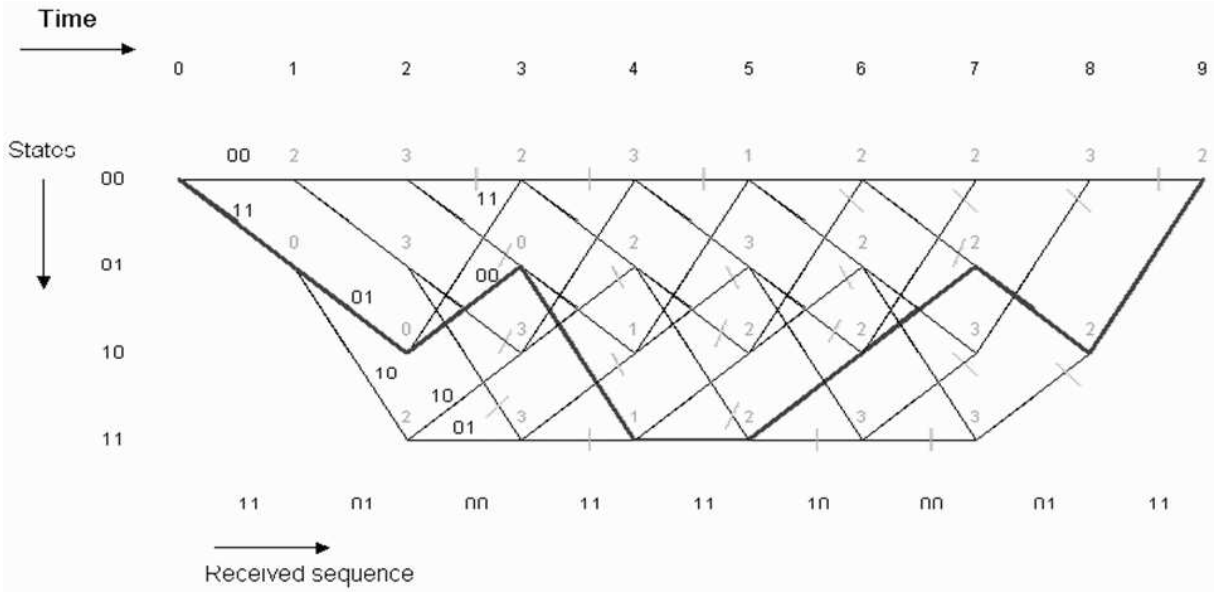


Figure 4.1: Trellis diagram example of Viterbi decoder (from [24]).

trellis is traversed. Since there are two paths merging at each node, the path with a smaller metric is selected while the other is discarded. This is based on the assumption that the optimum path must contain the sub-optimum survivor path. The survivor path for a given state at time instance n is the sequence of symbols closest to the received sequence up to time n. For the case of punctured convolutional code, the metrics associated with the punctured bits are simply disregarded in the metric calculation stage. The overall operation discussed above is the computational core of Viterbi algorithm and is the so-called add-compare-select (ACS) operation.

4.3 Decoding of Bit-Interleaved Coded Modulation

Similar techniques as that discussed in [5] and [27] can be used to demodulate and decode the received signal. The following gives a very brief introduction.

For Viterbi decoding, there are two decision types: hard-decision and soft-decision. If

hard-decision is adopted, the metric used in decoding is the Hamming distance, which counts the bit errors, between each trellis path and the hard-limited output of the demodulator to find the path with least errors. The coding gain is worse by 2 to 3 dB compared to soft-decision decoding. Hence, soft-decision is considered in this work.

For optimal soft-decision Viterbi decoding in AWGN channel, the metric should be the Euclidean distance between each trellis path and the soft-output of the demodulator. The problem now is that there is a bit interleaver between the convolutional encoder and the modulator in the transmitter. Therefore, the optimal decoder should be based on the super-trellis combining the convolutional code, the interleaver, and the QAM modulator, but this is too complex to be practical. Moreover, the puncturing mechanism adds further complexity to the super-trellis structure. Thus, we consider a suboptimal decoder based on bit-by-bit metric computation.

Consider 16QAM first. We denote the in-phase bits by $b_{I,1}$ and $b_{I,2}$, and the quadrature bits by $b_{Q,1}$ and $b_{Q,2}$, which are the four bits corresponding to the transmitted 16QAM symbol s. The soft-decision metric for $b_{I,k}$ is evaluated simply from $y_I[i]$ as

$$D_{I,1} = \left\{ \begin{array}{ll} -y_I(i), & |y_I(i)| \leq 2 \\ -2(y_I(i)-1), & y_I(i) > 2 \\ -2(y_I(i)+1), & y_I(i) < 2 \end{array} \right\} \cong -y_I(i), \tag{4.1}$$

$$D_{I,2} = |y_I(i)| - 2. \tag{4.2}$$

where $y_I(i)$ is the real part of the received signal after channel compensation. The evaluation of $D_{Q,k}$ for the two quadrature bits are the same as the evaluation of $D_{I,1}$ and $D_{I,2}$ with $y_I(i)$ replaced by $y_Q(i)$, where $y_Q(i)$ is the imaginary part of the received signal after channel compensation.

We also compute the log-likelihood ratio (LLR) of each received LDPC codeword bit by the above method [28].

Similar observations hold for QPSK and 64-QAM constellations. For QPSK,

$$D_I = -y_I[i], \tag{4.3}$$

$$D_Q = -y_Q[i]. \tag{4.4}$$

For 64-QAM,

$$D_{I,1} = \left\{ \begin{array}{ll} -y_I[i], & |y_I[i]| \le 2 \\ -2(y_I[i]-1), & 2 < y_I[i] \le 4 \\ -3(y_I[i]-2), & 4 < y_I[i] \le 6 \\ -4(y_I[i]-3), & y_I[i] > 6 \\ -2(y_I[i]+1), & -4 \le y_I[i] < -2 \\ -3(y_I[i]+2), & -6 \le y_I[i] < -4 \\ -4(y_I[i]+3), & y_I[i] < -6 \end{array} \right\} \cong -y_I[i], \tag{4.5}$$

$$D_{I,2} = \left\{ \begin{array}{ll} 2(|y_I[i]|-3), & |y_I[i]| \le 2 \\ -4+|y_I[i]|, & 2 < |y_I[i]| \le 6 \\ 2(|y_I[i]|-5), & |y_I[i]| > 6 \end{array} \right\} \cong -4+|y_I[i]|, \tag{4.6}$$

$$D_{I,3} = \left\{ \begin{array}{ll} -|y_I[i]|+2, & |y_I[i]| \le 4 \\ |y_I[i]|-6, & |y_I[i]| > 4 \end{array} \right\} = ||y_I[i]|-4|-2. \tag{4.7}$$

4.4 Profile of the DSP Code

We mention again that our implementation is based on modification of the code of Lee [24] for IEEE 802.16a OFDMA to the specifications of IEEE 802.16e OFDM. If more detailed steps of optimization and implementation are needed, [24] is the reference.

In this section, we show the optimized profile of our FEC encoder, which concatenates the RS encoder and the convolutional encoder. Table 4.1 shows the code size and the execution speed of the final concatenated encoding program for processing 144 data bytes (which includes data input and output) on DSP, for four of the mandatory coding and modulation modes of IEEE 802.16e OFDM. Here "data input and output included" means the execution time spent on input and output operations using fread() and fwrite() are included. Table 4.2 shows the corresponding information for the concatenated program of decoding 144 data bytes.

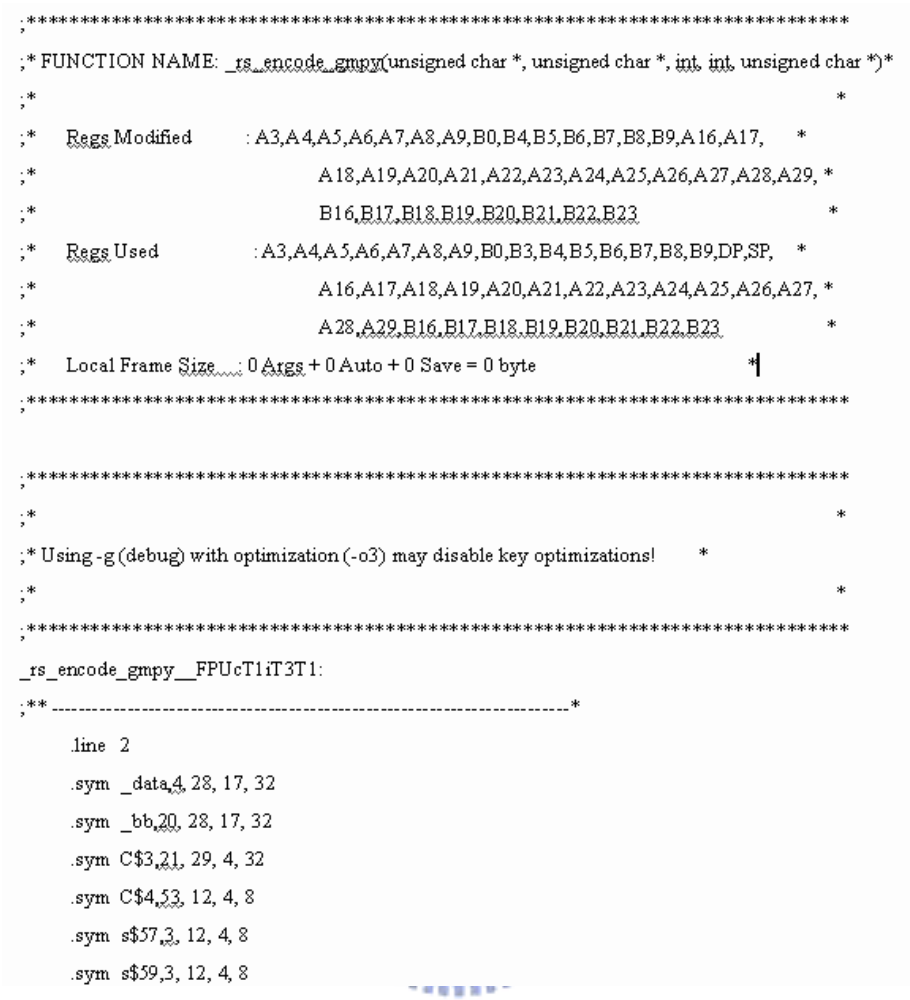
Table 4.1: Profile of Channel Encoder under Different Coding and Modulation Modes

Modulation	RS Code	CC Code Rate	Code Size (% RS, % CC)		Cycles (% RS, % CC)		Processing Rate (kbps)
QPSK	(32,24,4)	2/3	2208	(30,70)	107592	(50,50)	6424
QPSK	(40,36,2)	5/6	2544	(33,67)	70851	(25,75)	9755
16QAM	(64,48,8)	2/3	2524	(33,67)	80821	(18,82)	8552
16QAM	(80,72,4)	5/6	2908	(27,73)	94394	(9,91)	7322
Average			2546	(31,69)	88414	(26,74)	8013

Table 4.2: Profile of Channel Decoder under Different Coding and Modulation Modes

Modulation	RS Code	CC Code Rate	Code Size (% RS, % CC)		Cycles (% RS, % CC)		Processing Rate (kbps)
QPSK	(32,24,4)	2/3	8608	(68,32)	1063148	(13,87)	650
QPSK	(40,36,2)	5/6	8864	(67,33)	889147	(14,86)	777
16QAM	(64,48,8)	2/3	8148	(65,35)	903422	(4,96)	765
16QAM	(80,72,4)	5/6	8876	(60,40)	782811	(8,92)	883
Average			8624	(65,35)	909532	(8,92)	769

As they stand now, the programs will require multiple DSPs to run in parallel to handle the data rate under a 10 MHz transmission bandwidth. Acknowledgeably, further optimization of the programs may be possible. In addition, the C64x is equipped with a Viterbi decoder co-processor [29]. Using this co-processor may be helpful in raising the decoding speed. But its use requires study and testing of the "enhanced direct memory access (EDMA)" mechanism of the C64x chips, which is bypassed in the present study.



```
;******************************************************************************
;* FUNCTION NAME: _rs_encode_gmpy(unsigned char *, unsigned char *, int, int, unsigned char *)*
;*                                                                            *
;*    Regs Modified        : A3,A4,A5,A6,A7,A8,A9,B0,B4,B5,B6,B7,B8,B9,A16,A17,     *
;*                           A18,A19,A20,A21,A22,A23,A24,A25,A26,A27,A28,A29, *
;*                           B16,B17,B18,B19,B20,B21,B22,B23                  *
;*    Regs Used            : A3,A4,A5,A6,A7,A8,A9,B0,B3,B4,B5,B6,B7,B8,B9,DP,SP,    *
;*                           A16,A17,A18,A19,A20,A21,A22,A23,A24,A25,A26,A27, *
;*                           A28,A29,B16,B17,B18,B19,B20,B21,B22,B23           *
;*    Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                   *|
;******************************************************************************


;******************************************************************************
;*                                                                            *
;* Using -g (debug) with optimization (-o3) may disable key optimizations!    *
;*                                                                            *
;******************************************************************************
_rs_encode_gmpy__FPUcT1iT3T1:
;** --------------------------------------------------------------------------*
       .line  2
       .sym  _data,4, 28, 17, 32
       .sym  _bb,20, 28, 17, 32
       .sym  C$3,21, 29, 4, 32
       .sym  C$4,53, 12, 4, 8
       .sym  s$57,3, 12, 4, 8
       .sym  s$59,3, 12, 4, 8
```

Figure 4.2: The assembly codes of RS encoding (1/7).

4.5 Appendix

This section shows some figures that the assembly codes in RS encoding and the Chien search in RS decoding. In Figs. 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, and 4.8, we show the assembly codes of RS encoding.

In Figs. 4.9, 4.10, 4.11, and 4.12, we show the assembly codes of Chien search in RS encoding.

```
.sym  s$61,3, 12, 4, 8
.sym  s$63,3, 12, 4, 8
.sym  s$65,3, 12, 4, 8
.sym  s$67,3, 12, 4, 8
.sym  s$71,3, 12, 4, 8
.sym  s$73,3, 12, 4, 8
.sym  s$75,3, 12, 4, 8
.sym  s$77,3, 12, 4, 8
.sym  s$79,3, 12, 4, 8
.sym  s$81,3, 12, 4, 8
.sym  s$83,3, 12, 4, 8
.sym  s$85,3, 12, 4, 8
.sym  _feedback,3, 4, 4, 32
.sym  _data,3, 28, 4, 32
.sym  _bb,25, 28, 4, 32
.sym  L$1,53, 4, 4, 32
.sym  U$142,49, 12, 4, 8
.sym  U$133,41, 12, 4, 8
.sym  U$130,20, 12, 4, 8
.sym  U$123,9, 12, 4, 8
.sym  U$120,24, 12, 4, 8
.sym  U$113,3, 12, 4, 8
.sym  U$110,23, 12, 4, 8
.sym  U$97,37, 12, 4, 8
.sym  U$94,22, 12, 4, 8
```

Figure 4.3: The assembly codes of RS encoding (2/7).

```
        .sym  U$87,8, 12, 4, 8
        .sym  U$84,43, 12, 4, 8
        .sym  U$77,5, 12, 4, 8
        .sym  U$74,44, 12, 4, 8
        .sym  U$67,39, 12, 4, 8
        .sym  U$64,21, 12, 4, 8
        .sym  U$57,4, 28, 4, 32
;** 268      ----------------------      bb[14] = C$4 = (unsigned char)0;
;** 268      ----------------------      bb[15] = C$4;
;** 268      ----------------------      bb[12] = C$4;
;** 268      ----------------------      bb[13] = C$4;
;** 268      ----------------------      bb[10] = C$4;
;** 268      ----------------------      bb[11] = C$4;
;** 268      ----------------------      bb[8] = C$4;
;** 268      ----------------------      bb[9] = C$4;
;** 268      ----------------------      bb[6] = C$4;
;** 268      ----------------------      bb[7] = C$4;
;** 268      ----------------------      bb[4] = C$4;
;** 268      ----------------------      bb[5] = C$4;
;** 268      ----------------------      bb[2] = C$4;
;** 268      ----------------------      bb[3] = C$4;
;** 268      ----------------------      *bb = C$4;
;** 268      ----------------------      bb[1] = C$4;
;**    ----------------------      U$57 = data;
;**    ----------------------      C$3 = &((unsigned short *)Gg_poly)[0];
;**    ----------------------      U$64 = ((unsigned char *)C$3)[14];
```

Figure 4.4: The assembly codes of RS encoding (3/7).

```
;**   ----------------------      U$67 = ((unsigned char *)C$3)[15];
;**   ----------------------      U$74 = ((unsigned char *)C$3)[12];
;**   ----------------------      U$77 = ((unsigned char *)C$3)[13];
;**   ----------------------      U$84 = ((unsigned char *)C$3)[10];
;**   ----------------------      U$87 = ((unsigned char *)C$3)[11];
;**   ----------------------      U$94 = ((unsigned char *)C$3)[8];
;**   ----------------------      U$97 = ((unsigned char *)C$3)[9];
;**   ----------------------      U$110 = ((unsigned char *)C$3)[6];
;**   ----------------------      U$113 = ((unsigned char *)C$3)[7];
;**   ----------------------      U$120 = ((unsigned char *)C$3)[4];
;**   ----------------------      U$123 = ((unsigned char *)C$3)[5];
;**   ----------------------      U$130 = ((unsigned char *)C$3)[2];
;**   ----------------------      U$133 = ((unsigned char *)C$3)[3];
;**   ----------------------      U$142 = ((unsigned char *)C$3)[1];
;** 270 ----------------------      L$1 = 239;
;**   ----------------------      #pragma MUST_ITERATE(239, 239, 239)
;**   ----------------------      #pragma LOOP_FLAGS(4096u)
;**   ----------------------g2:
;** 271 ----------------------      s$57 = bb[2];
;** 271 ----------------------      s$59 = bb[3];
;** 271 ----------------------      s$61 = bb[4];
;** 271 ----------------------      s$63 = bb[5];
;** 271 ----------------------      s$65 = bb[6];
;** 271 ----------------------      s$67 = bb[7];
;** 271 ----------------------      feedback = *U$57++^*bb;
;** 87  ----------------------      *bb = _gmpy4(U$67, feedback)^bb[1];   // [6]
```

Figure 4.5: The assembly codes of RS encoding (4/7).

```
;** 87      ----------          bb[1] = _gmpy4(U$64, feedback)^s$57;   // [6]
;** 87      ----------          bb[2] = _gmpy4(U$77, feedback)^s$59;   // [6]
;** 87      ----------          bb[3] = _gmpy4(U$74, feedback)^s$61;   // [6]
;** 87      ----------          bb[4] = _gmpy4(U$87, feedback)^s$63;   // [6]
;** 87      ----------          bb[5] = _gmpy4(U$84, feedback)^s$65;   // [6]
;** 87      ----------          bb[6] = _gmpy4(U$97, feedback)^s$67;   // [6]
;** 87      ----------          s$71 = bb[9];   // [6]
;** 87      ----------          s$73 = bb[10];   // [6]
;** 87      ----------          s$75 = bb[11];   // [6]
;** 87      ----------          s$77 = bb[12];   // [6]
;** 87      ----------          s$79 = bb[13];   // [6]
;** 87      ----------          s$81 = bb[14];   // [6]
;** 87      ----------          s$83 = bb[15];   // [6]
;** 87      ----------          bb[7] = _gmpy4(U$94, feedback)^bb[8];   // [6]
;** 87      ----------          bb[8] = _gmpy4(U$113, feedback)^s$71;   // [6]
;** 87      ----------          bb[9] = _gmpy4(U$110, feedback)^s$73;   // [6]
;** 87      ----------          bb[10] = _gmpy4(U$123, feedback)^s$75;   // [6]
;** 87      ----------          bb[11] = _gmpy4(U$120, feedback)^s$77;   // [6]
;** 87      ----------          bb[12] = _gmpy4(U$133, feedback)^s$79;   // [6]
;** 87      ----------          bb[13] = _gmpy4(U$130, feedback)^s$81;   // [6]
;** 87      ----------          s$85 = Gg_poly[0];   // [6]
;** 87      ----------          bb[14] = _gmpy4(U$142, feedback)^s$83;   // [6]
;** 87      ----------          bb[15] = _gmpy4(s$85, feedback);   // [6]
;** 275     ----------          if ( --L$1 ) goto g2;
;** 277     ----------          return;
```

Figure 4.6: The assembly codes of RS encoding (5/7).

```
           ZERO    .D2     B5                  ; |268|
||         MV      .S2     B4,B9               ; |261|

           STB     .D2T2   B5,*+B9(1)          ; |268|
           STB     .D2T2   B5,*B9              ; |268|
           STB     .D2T2   B5,*+B9(14)         ; |268|
           STB     .D2T2   B5,*+B9(13)         ; |268|
           STB     .D2T2   B5,*+B9(15)         ; |268|
           STB     .D2T2   B5,*+B9(12)         ; |268|
           STB     .D2T2   B5,*+B9(10)         ; |268|
           STB     .D2T2   B5,*+B9(9)          ; |268|
           STB     .D2T2   B5,*+B9(11)         ; |268|
           STB     .D2T2   B5,*+B9(8)          ; |268|
           STB     .D2T2   B5,*+B9(6)          ; |268|
           STB     .D2T2   B5,*+B9(5)          ; |268|
           STB     .D2T2   B5,*+B9(7)          ; |268|
           STB     .D2T2   B5,*+B9(4)          ; |268|
           STB     .D2T2   B5,*+B9(2)          ; |268|
           STB     .D2T2   B5,*+B9(3)          ; |268|

           MVK     .S2     (_Gg_poly-$bss),B5
||         LDBU    .D2T2   *+B9(9),B22         ; |87| (P) <0,3>
```

```
           ADD     .S2     DP,B5,B18
||         LDBU    .D2T1   *+DP(_Gg_poly),A25 ; |87| (P) <0,0>
```

Figure 4.7: The assembly codes of RS encoding (6/7).

```
        LDBU    .D2T1   *+B18(7),A3
        LDBU    .D2T1   *+B18(10),A22
        LDBU    .D2T1   *+B18(9),A16
        MV      .D1X    B4,A27              ; |261|
||      LDBU    .D2T2   *+B18(2),B4
        LDBU    .D2T1   *+B18(1),A28
        LDBU    .D2T1   *+B18(3),A20
        LDBU    .D2T1   *+B18(11),A8
        LDBU    .D2T2   *+B18(4),B8
        LDBU    .D2T2   *+B18(6),B7
        LDBU    .D2T1   *+B18(12),A23
        LDBU    .D2T2   *+B18(8),B6
        MV      .D1     A4,A6
||      LDBU    .D2T1   *+B18(15),A18
        LDBU    .D2T2   *B9,B16             ; |271| (P) <0,5>   ^
||      LDBU    .D1T1   *A6++,A4            ; |271| (P) <0,5>
        MVC     .S2     CSR,B23
||      LDBU    .D2T1   *+B18(5),A9
        AND     .S2     -2,B23,B17
||      LDBU    .D2T2   *+B18(14),B5
        MVC     .S2     B17,CSR             ; interrupts off
||      LDBU    .D2T1   *+B18(13),A5
```

Figure 4.8: The assembly codes of RS encoding (7/7).

```
;*****************************************************************************
;* FUNCTION NAME: _chien                                                     *
;*                                                                           *
;*    Regs Modified        : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,B0,B1,B2,B3,B4,B5,B6,   *
;*                           B7,B8,A16,B31                                   *
;*    Regs Used            : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,B0,B1,B2,B3,B4,B5,B6,   *
;*                           B7,B8,DP,SP,A16,B31                             *
;*    Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                  *
;*****************************************************************************

;*****************************************************************************
;*                                                                           *
;* Using -g (debug) with optimization (-o2) may disable key optimizations!   *
;*                                                                           *
;*****************************************************************************
_chien:
;** ---------------------------------------------------------------------*
        .line   2
        .sym    A$4,4, 4, 4, 32
        .sym    C$2,21, 20, 4, 32
        .sym    C$3,20, 20, 4, 32
        .sym    U$9,8, 20, 4, 32
        .sym    K$13,21, 4, 4, 32
        .sym    U$12,3, 4, 4, 32
```

Figure 4.9: The assembly codes of Chien search in RS decoding (1/4).

```
        .sym  U$29,20, 4, 4, 32
        .sym  U$0,7, 4, 4, 32
        .sym  U$37,5, 28, 4, 32
        .sym  U$90,3, 20, 4, 32
        .sym  K$88,5, 4, 4, 32
        .sym  L$1,0, 4, 4, 32
        .sym  L$2,22, 4, 4, 32
        .sym  L$3,16, 4, 4, 32
        .sym  L$4,16, 4, 4, 32
        .sym  _i,8, 4, 4, 32
        .sym  V$2,0, 4, 4, 32
        .sym  V$1,6, 4, 4, 32
        .sym  V$0,9, 4, 4, 32
        .sym  s$0,3, 4, 4, 32
        .sym  K$16,37, 28, 4, 32
        .sym  U$22,9, 20, 4, 32
        .sym  U$22,20, 20, 4, 32
        .sym  U$28,23, 4, 4, 32
        .sym  U$28,16, 4, 4, 32
        .sym  _j,20, 4, 4, 32
;** 116     ----------------------      j = deg_lambda;
;** 116     ----------------------      if ( (U$0 = j) < 0 ) goto g4;
;**    ----------------------      U$9 = &lambda[U$0];
;**    ----------------------      U$12 = _lo(_mpyli(202, U$0));
```

Figure 4.10: The assembly codes of Chien search in RS decoding (2/4).

```
;**   -----------------------      K$13 = 255;
;**   -----------------------      U$22 = &reg[U$0];
;** 117     -----------------------      L$1 = U$0+1;
;**   -----------------------      K$16 = &Alpha_to[0];
;**   -----------------------      #pragma MUST_ITERATE(1, 1099511627775, 1)
;**   -----------------------      #pragma LOOP_FLAGS(4096u)
            LDW       .D2T2    *+DP(_deg_lambda),B4 ; |116|
            NOP                1
            MVK       .S1      202,A3
            MVK       .S2      0xff,B5
            MV        .L2      B3,B31              ; |113|
            MV        .D1X     B4,A7               ; |116|

            CMPLT     .L2      B4,0,B0             ; |116|
||          MPYLI     .M1      A3,A7,A5:A4

   [ B0]    BNOP      .S1      L19,1               ; |116|

            MVK       .S1      (_reg-$bss),A3
||          ADD       .D1X     1,B4,A0             ; |117|

            MVK       .S1      (_lambda-$bss),A3
||          ADD       .D1X     DP,A3,A5
```

Figure 4.11: The assembly codes of Chien search in RS decoding (3/4).

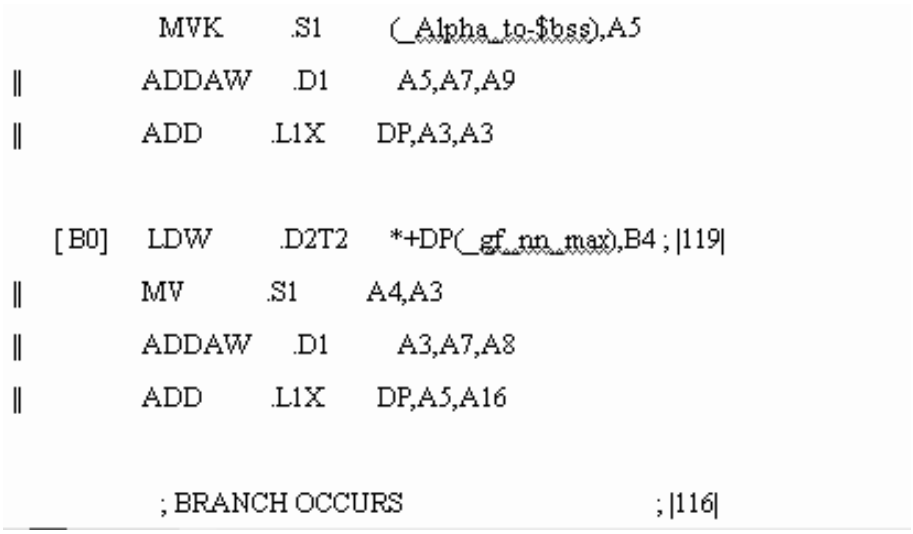


Figure 4.12: The assembly codes of Chien search in RS decoding (4/4).

Chapter 5

Decoding Algorithms of LDPC Codes in IEEE 802.16e OFDMA

In this chapter, we describe some decoding algorithms for LDPC codes and some simulation results in AWGN channel. The simulation results provide us the information to select appropriate decoding scheme, code rate, codeword length, and modulation type according to the system performance requirement, computational complexity, and latency. The material in Section 5.1 and 5.2 is mainly from [30].

5.1 The Belief Propagation Algorithm [30]

Using Tanner graph representation of LDPC codes is attractive, because it not only helps understand their parity-check structure, but, more importantly, also facilitates a powerful decoding approach. The key decoding steps are the local application of Bayes rule at each node and the exchange of the results (messages) with neighboring nodes. At any given iteration, two types of messages are passed: probabilities or beliefs from bit nodes to check nodes, and probabilities or beliefs from check nodes to bit nodes.

Let $M(n)$ denote the set of check nodes connected to bit node n, i.e., the positions of ones in the nth column of H, and let $N(m)$ denote the set of bit nodes that participate in the mth

parity-check equation, i.e., the positions of ones in the mth row of H. Let $N(m)\backslash n$ represent the exclusion of n from the set $N(m)$, and $M(n)\backslash m$ represent the exclusion of m from the set $M(n)$. In addition, $q_{n\to m}(0)$ and $q_{n\to m}(1)$ denote the message from bit node n to check node m indicating the probability of bit n being zero or one, respectively, based on all the checks involving n except m. Similarly, $r_{m\to n}(0)$ and $r_{m\to n}(1)$ denote the message from check node m to bit node n indicating the probability of bit n being zero or one, respectively, based on all the bit checked by m except n. Let $\mathbf{x} = [x_1, x_2, \cdots, x_N]$ and $\mathbf{y} = [y_1, y_2, \cdots, y_N]$ denote the transmitted codeword and the received codeword respectively. Finally, $L_n^{(0)}$ denotes $\log(P(x_n = 0|y_n)/P(x_n = 1|y_n))$ at iteration 0, and $L_{mn}^{(i)}$ denotes $\log(r_{m\to n}(0)/r_{m\to n}(1))$ at iteration i. $Z_{mn}^{(i)}$ denotes $\log(q_{n\to m}(0)/q_{n\to m}(1))$ at iteration i.

The belief propagation (BP) algorithm is summarized as follows. This algorithm is also known as the sum-product (SP) algorithm.

Step 1 (check-node update): For each m and for each $n \in N(m)$, compute

$$L_{mn}^{(i)} = 2\tanh^{-1}\left\{\prod_{n'\in N(m)\backslash n} \tanh\frac{Z_{mn'}^{(i-1)}}{2}\right\}. \tag{5.1}$$

Step 2 (bit-node update): For each n, and for each $m \in M(n)$ compute

$$Z_{mn}^{(i)} = L_n^{(0)} + \sum_{m'\in M(n)\backslash m} L_{m'n}^{(i)}. \tag{5.2}$$

Step 3 (decision):

$$Z_n^{(i)} = L_n^{(0)} + \sum_{m\in M(n)} L_{mn}^{(i)}. \tag{5.3}$$

The decoder output vector follows the rule: $\hat{x}_n = 0$ if $Z_n^{(i)} \geq 0$, and $\hat{x}_n = 1$ if $Z_n^{(i)} < 0$.

The decoded bit vector is checked with the parity check matrix H. The iterative decoding decoding procedure stops when either $H.X$=0 or as the maximum decoding iteration number has been reached, where $\mathbf{X} = [X_1, X_2, \cdots, X_N]$ is the decoded codeword.

5.2 Some Reduced-Complexity Decoding Algorithms [30]

In this section, we focus on simplifying the check node updates to obtain reduced-complexity BP algorithms and also achieve good enough performance.

5.2.1 BP-Based Algorithm

Implementing the calculation in Eq. (5.1) in a hardware circuit is very difficult and complex. Hence, we can simplify this equation in the check nodes process as

$$\begin{aligned} L_{mn}^{(i)} &= 2\tanh^{-1}\left\{\prod_{n'\in N(m)\backslash n}\tanh\frac{Z_{mn'}^{(i-1)}}{2}\right\} \\ &= \prod_{n'\in N(m)\backslash n} sgn(Z_{mn'}^{(i-1)}) f\left(\sum_{n'\in N(m)\backslash n} f\left(|Z_{mn'}^{(i-1)}|\right)\right) \\ &\approx \prod_{n'\in N(m)\backslash n} sgn(Z_{mn'}^{(i-1)}) f\left(f\left(\min_{n'\in N(m)\backslash n}|Z_{mn'}^{(i-1)}|\right)\right) \\ &= \prod_{n'\in N(m)\backslash n} sgn(Z_{mn'}^{(i-1)}) \min_{n'\in N(m)\backslash n}|Z_{mn'}^{(i-1)}|, \end{aligned} \tag{5.4}$$

where $f(x) = \log\frac{e^x+1}{e^x-1} = -\log(\tanh\frac{x}{2})$ is an exponential decay function. Therefore the 2nd row in Eq. (5.4) can be approximated as the 3rd row in Eq. (5.4). Because the f function has the property, $f(x) = f^{-1}(x)$, we can simplify the 3rd row Eq. (5.4) to the 4th row in Eq. (5.4).

This is a famous approximation called min-sum algorithm or BP-based algorithm which only uses the signum and the minimum functions for the check nodes process. The process procedure in bit nodes is identical to that of BP decoding. But coming with the approximation in check nodes is a performance degradation. We will discuss the degradation effect later in the simulation results.

5.2.2 Balanced Belief Propagation Algorithm [31]

It can be observed that the conventional BP algorithm has unbalanced computation complexity between check nodes operation (5.1) and bit nodes operation (5.2).

A modified version based on algorithmic transformation has been proposed in order to balance the computation load between the two decoding phases. The new algorithm can be expressed as

$$L_{mn}^{(i)} = \prod_{n' \in N(m) \backslash n} sgn(Z_{mn'}^{(i-1)}) \sum_{n' \in N(m) \backslash n} f\left(|Z_{mn'}^{(i-1)}|\right), \tag{5.5}$$

$$Z_{mn}^{(i)} = L_n^{(0)} + \sum_{m' \in M(n) \backslash m} sgn(L_{m'n}^{(i)}) f\left(L_{m'n}^{(i)}\right). \tag{5.6}$$

We note that $L_{mn}^{(i)}$ computed here is different from what is obtained with the BP algorithm. The main benefit with the modified algorithm is the balance of computation complexity between two decoding phases.

5.2.3 Normalized BP-Based Algorithm

Let L_1 and L_2 represent the values $L_{mn}^{(i)}$ computed by the BP algorithm and the BP-based algorithm with (5.1) and (5.4), respectively. It can be shown that L_1 and L_2 have the same sign, i.e., sgn$(L1)$ = sgn$(L2)$ and L_2 has larger magnitude than L_1, i.e., $|L_2| > |L_1|$ [32].

According by [32], we can further modify (5.4) to let the BP-based algorithm obtain a BER vs. $\frac{Eb}{No}$ performance curve as close as the conventional BP algorithm.

Because sgn$(L1)$ = sgn$(L2)$, the BP-based decoding can be improved by employing a check-node update $L_{mn}^{(i)}$ that uses a normalization constant α greater than one, that is,

$$\widehat{L_{mn}^{(i)}} \longleftarrow \frac{L_{mn}^{(i)}}{\alpha}. \tag{5.7}$$

Here $\widehat{L_{mn}^{(i)}}$ is the value computed from the check node operation for normalized BP-based algorithm, and the bit node operation for normalized BP-based algorithm is the same as BP algorithm.

Although α should vary with different signal-to-noise ratios (SNRs) and different iterations to achieve the optimum performance, it is kept a constant for the sake of simplicity.

5.2.4 Offset BP-Based Algorithm

For offset BP-based decoding, we modify $L_{mn}^{(i)}$ in BP-based decoding by subtracting a positive constant β as

$$\widehat{L_{mn}^{(i)}} \longleftarrow sgn(L_{mn}^{(i)}) \max(|L_{mn}^{(i)}| - \beta, 0) \tag{5.8}$$

where $\widehat{L_{mn}^{(i)}}$ is the value computed from the check node operation for offset BP-based algorithm, and the bit node operation for offset BP-based algorithm is the same as BP algorithm.

Although β should vary with different signal-to-noise ratios (SNRs) and different iterations to achieve the optimum performance, it is kept a constant for the sake of simplicity.

As we describe above, the BP decoding needs $tanh^{-1}$ and $tanh$ operations, the min-sum algorithm needs minimal operation, the normalized BP-based algorithm needs minimal and division operations, and the offset BP-based algorithm needs minimal, maximum and substraction operations. These operations for all different algorithm are listed in Table 5.1. Obviously, the BP decoding needs the most complex operation, and the min-sum decoding needs the lowest complex operation. The two improved decoding methods are between the BP decoding and min-sum decoding.

Table 5.1: Operation Comparison for all Decoding Algorithms

Decoding Algorithms	Main Operations
BP Decoding	$tanh$ and $tanh^{-1}$
Min-Sum Decoding	Minimal
Normalized BP-Based Decoding	Minimal and Division
Offset BP-Based Decoding	Minimal, Maximum and Substraction

5.3 Early Termination [33]

The LDPC decoding is based on iterative convergence of a bit probability towards zero or one. A particular property is the convergence pattern.

After a set of initial iterations, the convergence patterns start to separate either towards 0 or towards 1. We can exploit this particular property and produce the decoding result sooner than it would have. In other words, we can reduce the iteration numbers as we expect.

We do that by counting the number of incoming LLR values which fall under (over) a certain threshold LLR. In our simulation, when two consecutive incoming LLR values are under -2 or over 2, then we can set the convergence outcome immediately without the need to further iterate.

Reducing the number of iterations (and hence computations) results in both savings in power consumption and increased throughput, with the only drawback being a slight increase in the BER.

Actually, we got some advice of modifying the original (or above) early termination technique by private communication of Professor S. G. Chen. That is, after a specified iteration number if the difference of LLR values between this and next iteration is small enough, we also can stop the iterating. This is because of the LLR value is no more changed

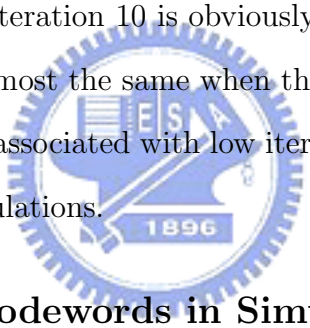
a lot if we still iterate further.

5.4 Simulation Results and Analysis

5.4.1 Determine the Number of Iterations

One of the most important factors of concern when decoding the received codewords is the iteration number. As the number becomes larger, the correct codewords are more likely to be exactly decoded. But using more iterations, the cost is that the latency is increased. Therefore we need to choose a proper iteration number in decoding. In Fig. 5.1, we show the simulation results with different iteration numbers, for the LDPC codes at rate 1/2 and length 576 with QPSK modulation and and BP decoding.

In Fig. 5.1, the BER curve of iteration 10 is obviously degenerated by the reason of less iterations. The BER curves are almost the same when the iteration numbers are 20, 30, 50, and 70. To avoid the degradation associated with low iteration numbers, we adopt 50 as the iteration number in the other simulations.



5.4.2 Use of All-Zero Codewords in Simulation

We have described the LDPC code encoder specified in IEEE 802.16e in Section 2.2.2. In Fig. 5.2, we show the simulation results of LDPC code with random data which would be encoded by the LDPC encoder and all zero codeword respectively when code rate 1/2, length 576, QPSK, and BP decoding are adopted. This result depicts that the BER curves almost have no difference when the random data and all zero codeword are transmitted. From the theoretical view, this result can also be expected. This is due to that the LDPC codes are in the class of linear codes. For the sake of simplicity (or faster simulation speed) and without loss of generality, we will take the all zero codeword as the transmitted codeword in the

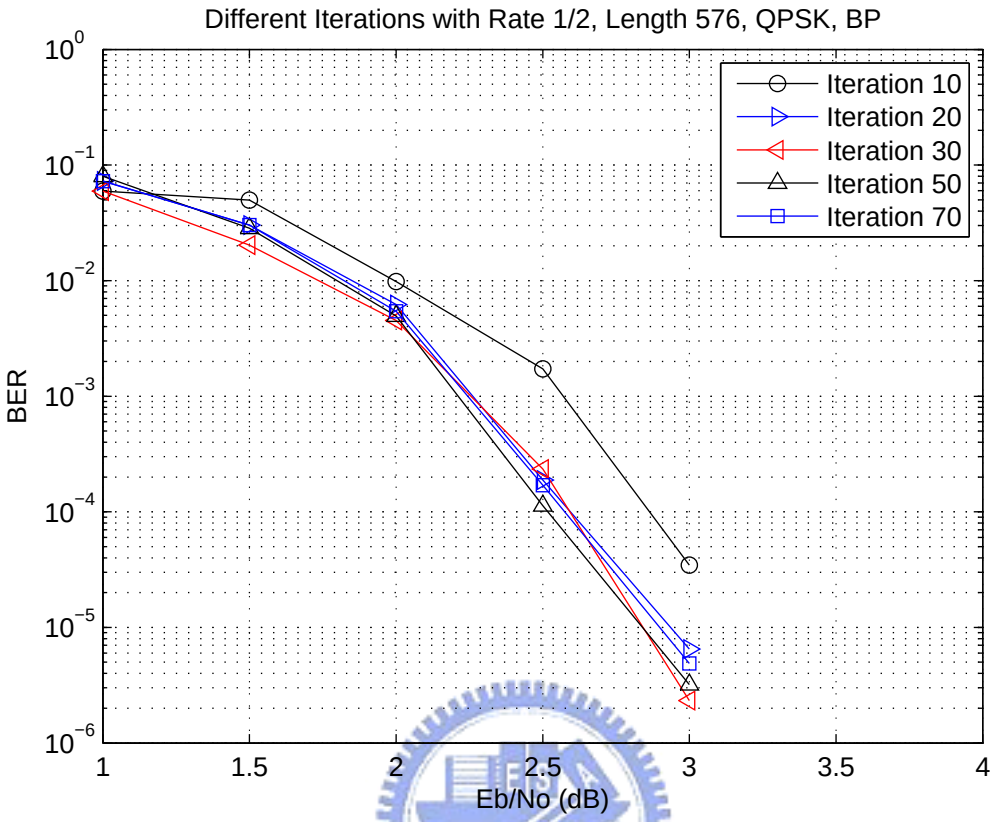


Figure 5.1: Decoding performance at different iteration numbers.

other simulations.

5.4.3 Performance of the IEEE 802.16e LDPC Codes under the BP Algorithm

In this section, we simulate different modulation types, codeword lengths, and code rates specified in IEEE 802.16e standard respectively.

Figure 5.3 depicts the performance of the code at rate 1/2 and length 576, under different modulation schemes with BP decoding with iteration 50. As we expect in advance, the performance of QPSK is better than that of 16QAM and the performance of 16QAM is

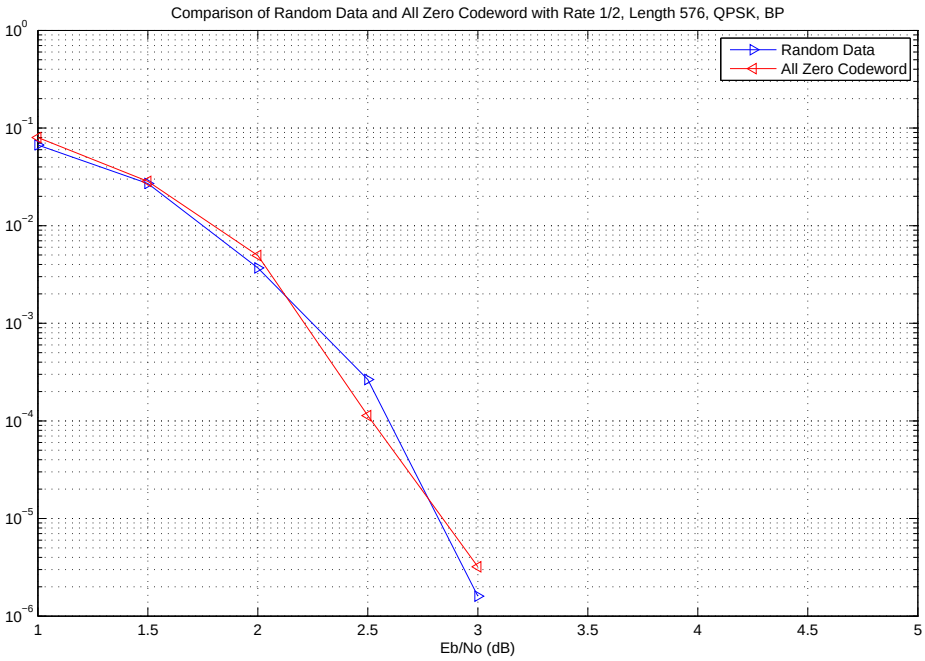


Figure 5.2: Performance of random data versus all-zero codeword.

also better than that of 64QAM. From Fig. 5.3, the coding gain values of QPSK, 16QAM, and 64QAM modulation are 6.6589 dB, 6.6511 dB, and 8.4999 dB respectively when the bit error rate is 10^{-5}. The coding gain values of QPSK and 16QAM modulation are almost the same, but the coding gain of 64QAM modulation is larger than the coding gain of other two modulation types about 1.8 dB.

Figure 5.4 depicts four different codeword lengths, that are 576, 1152, 1728, and 2304 when code rate 1/2, QPSK, and BP decoding with iteration 50 are adopted. We have some observations from Fig. 5.4. First, as the codeword is longer, the improved performance is obtained. Second, for the codeword length 2304, the bit error rate reaches about 10^{-9} and only E_b/N_0 2.5 dB is needed. The coding gain between length 2304 curve and uncoded curve is about 8 dB when the bit error rate is 10^{-6}. This coding gain value somehow depicts the

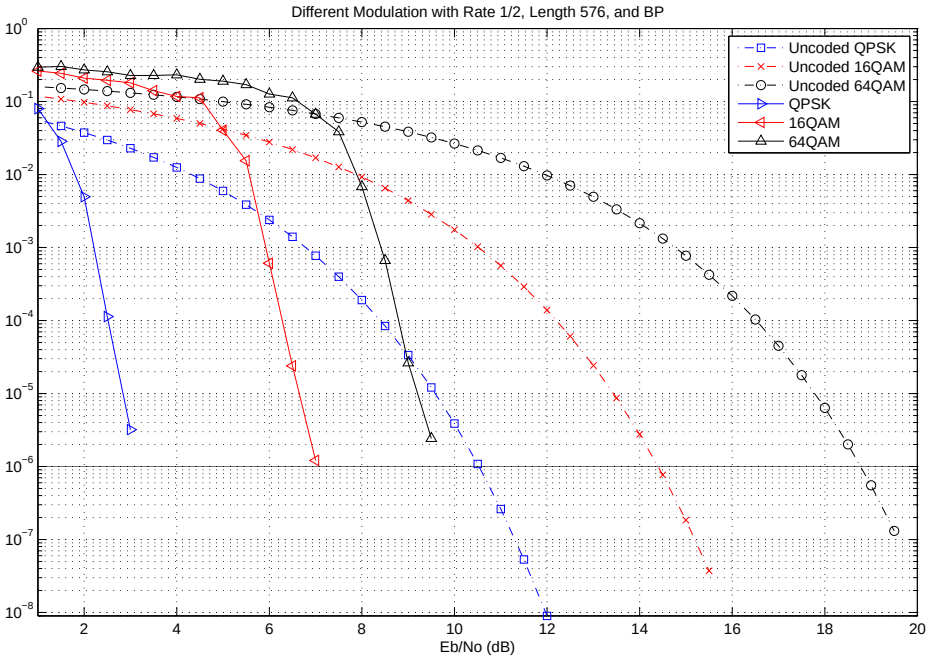


Figure 5.3: Performance of the rate-1/2 code, length 576 code.

error correcting ability of LDPC codes is really amazing.

Figure 5.5 depicts six different code rate types, that are $\frac{1}{2}$, $\frac{2}{3}A$, $\frac{2}{3}B$, $\frac{3}{4}A$, $\frac{3}{4}B$, and $\frac{5}{6}$ when length 576, QPSK, and BP decoding with iteration 50 are adopted. There are some observations can be obtained from Fig. 5.5. As the code rate is higher, the performance is worse. Besides, we notice that the two BER curves of $\frac{2}{3}A$ and $\frac{2}{3}B$ are very close, but still have some difference. We can explain why this little difference exists from the view of threshold previously obtained by density evolution method. We have obtained some threshold results in Table 2.6. From Table 2.6, the threshold of $\frac{2}{3}A$ is larger than that of $\frac{2}{3}B$. Moreover, the difference of threshold for code rate $\frac{2}{3}A$ and $\frac{2}{3}B$ is only about 0.012 dB. Thus we reasonably anticipate the BER curves are very close, and the curve for $\frac{2}{3}A$ is a little better than that of $\frac{2}{3}B$. In our simulation, these two curves really follow the threshold analysis. By the similar

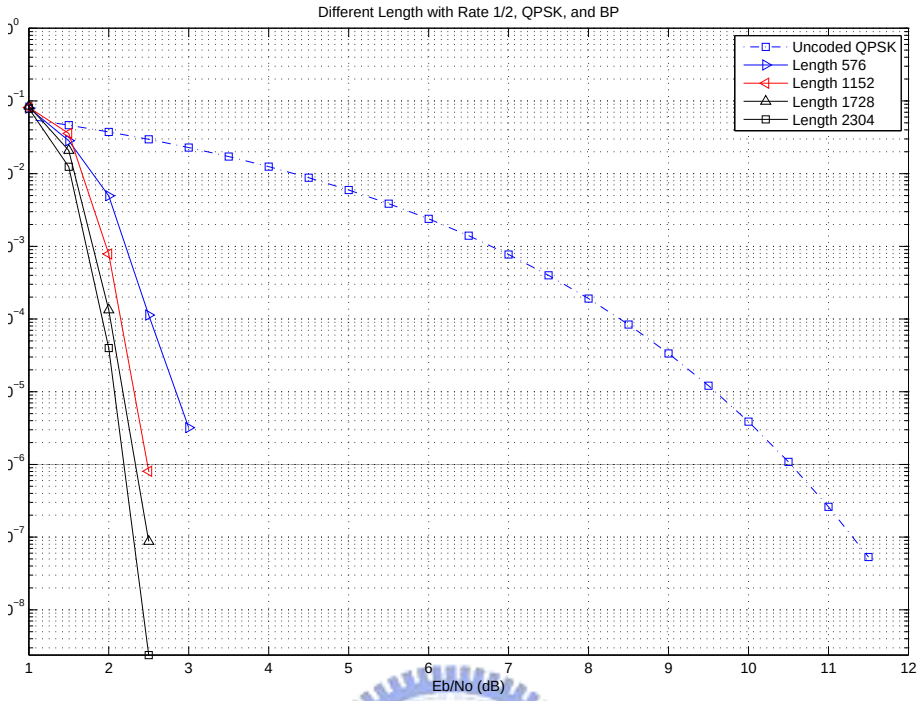


Figure 5.4: Performance of the rate-1/2 code at different codeword lengths, under QPSK modulation and BP decoding.

method, we also easily explain the relationship between the two BER curves of $\frac{3}{4}A$ and $\frac{3}{4}B$ from Table 2.6.

Table 5.2 shows the relation between the $\frac{Eb}{No}$ value when BER is 10^{-5}, girth when length is 576, and the threshold for all code rate. As the threshold is larger, we need less channel $\frac{Eb}{No}$ to reach BER 10^{-5}.

5.4.4 Performance of Balanced BP Decoding Algorithm

We have described the concept of balanced BP decoding algorithm in Section 5.2.2. We depict the performance of conventional BP decoding and balanced BP decoding in Fig. 5.6

Table 5.2: Relation between $\frac{Eb}{No}$, Girth, and Threshold

Code Rate	$\frac{1}{2}$	$\frac{2}{3}A$	$\frac{2}{3}B$	$\frac{3}{4}A$	$\frac{3}{4}B$	$\frac{5}{6}$
$\frac{Eb}{No}$ under BER $10^{-5}(dB)$	2.9691	4.9506	5.1603	5.9819	5.9730	6.9898
Girth under Length 576	6	6	6	4	6	6
Threshold	0.9273	0.7282	0.7163	0.6358	0.6446	0.5607

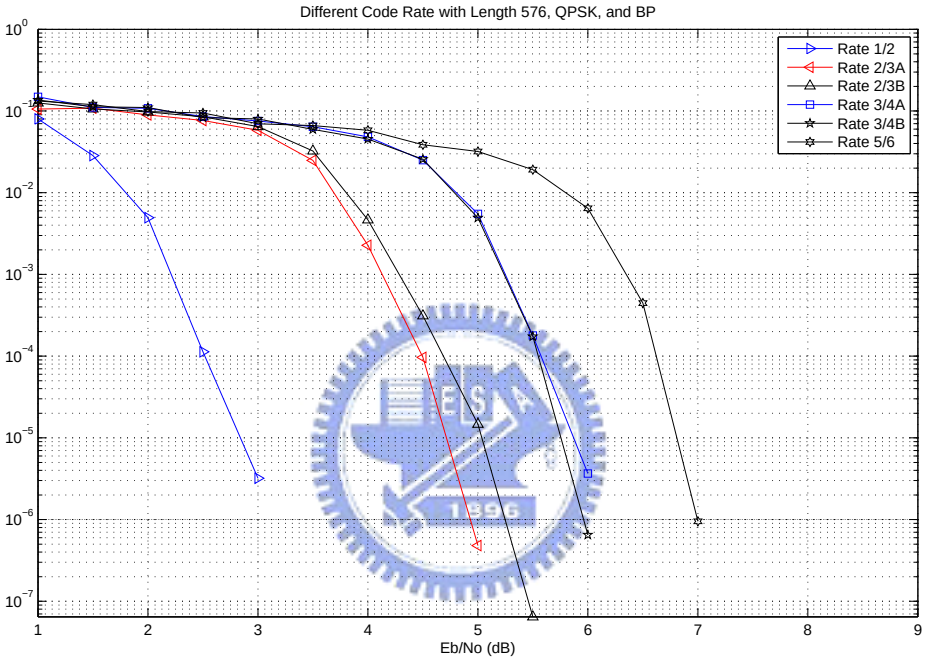


Figure 5.5: Performance of different code rates at codeword length 576, under QPSK modulation and BP decoding.

when length 576, rate $\frac{1}{2}$, and QPSK are applied. We observe that these two curves with different decoding algorithms are almost the same. Therefore, in our future fixed-point DSP implementation, we can consider balanced BP decoding to reduce the clock cycles.

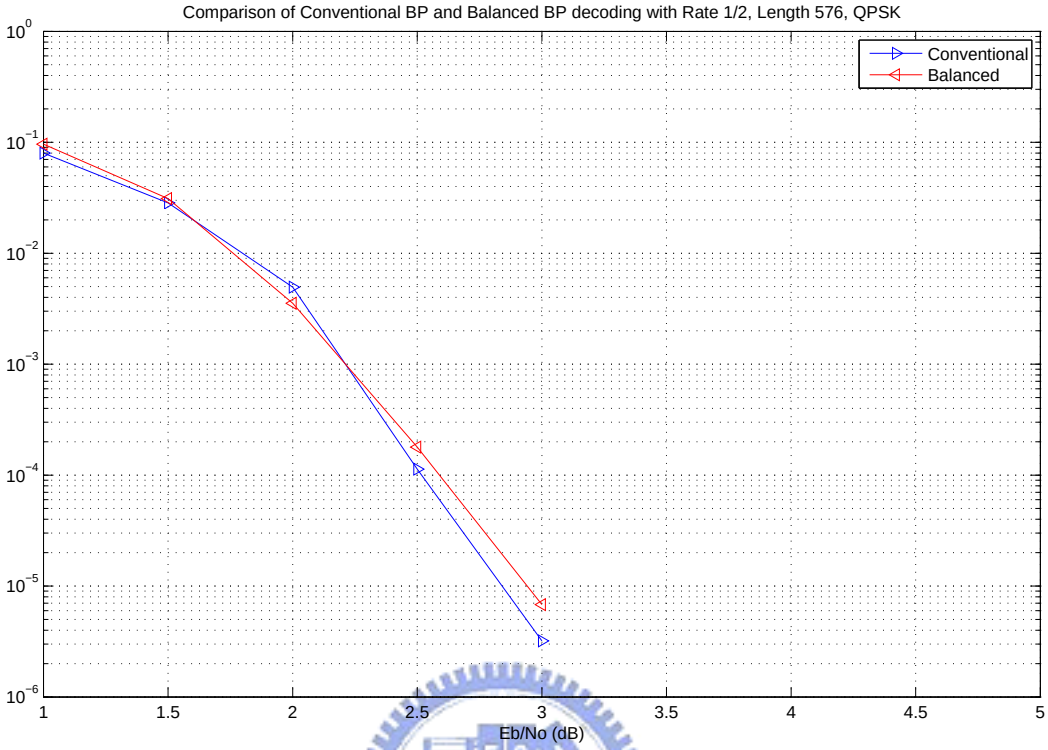


Figure 5.6: Conventional BP and balanced BP decoding with length 576, rate 1/2 code, and QPSK modulation.

5.4.5 Choose Appropriate Early Termination Parameters

In Section 5.5, we described the concept of early termination. Here we consider the appropriate parameters for early termination. First we define some parameters. Type 1 early termination means when two consecutive incoming LLR values are over 2 or under -2, then we set the convergence outcome immediately without further iteration. Type 2 early termination means when all incoming LLR values are over 2 or under -2 one time or when two consecutive incoming LLR values are over 1 or under -1, we set the convergence outcome immediately without further iteration. From the complexity view, type 2 early termination is simpler than type 1, but the performance may be an issue. Fig. 5.7 shows some performance

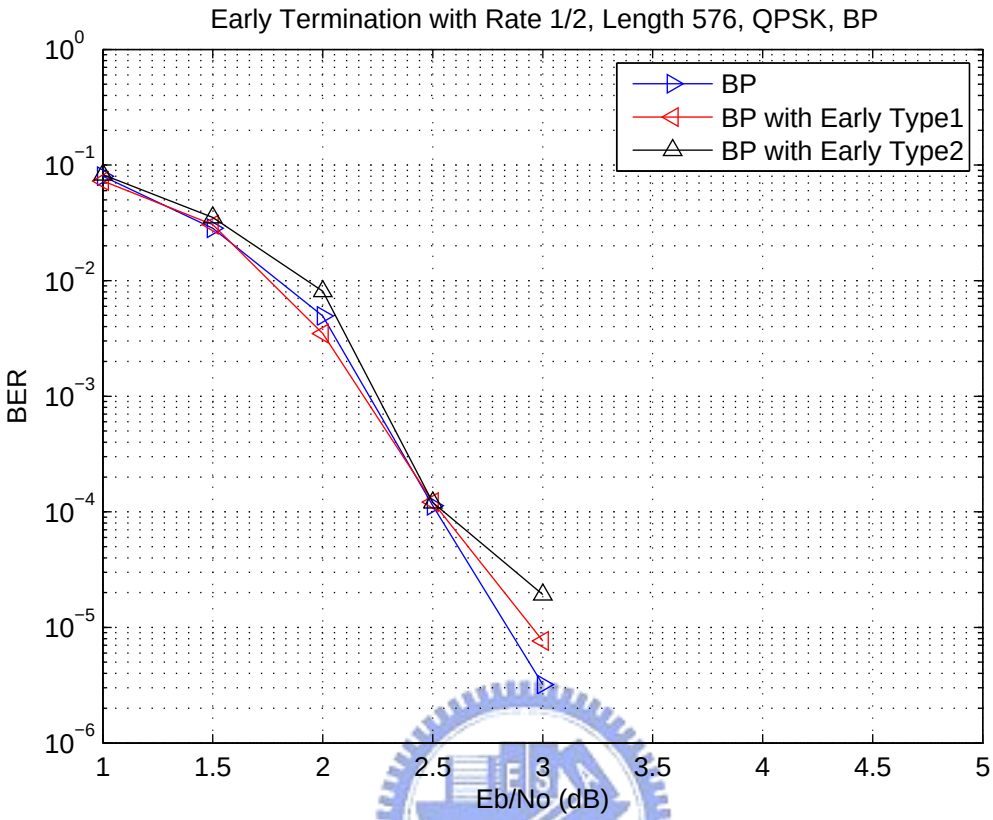


Figure 5.7: Effects of differenct ways of early termination.

results for conventional BP decoding, BP decoding with type 1 early termination, and BP decoding with type 2 early termination. The code rate $\frac{1}{2}$, length 576, and QPSK are applied. From Fig. 5.7, BP decoding with type 1 early termination has a better performance than the one with type 2 early termination at high SNR values. Conventional BP decoding still has the best performance of all three methods. Considering the trade-off between complexity and performance, we adopt the type 1 early termination as the early termination scheme in our next simulation.

We are also interested in the pdf of reduced iteration number for type 1 early termination besides the performance. Fig. 5.8 shows the pdf with lengths 576 and 2304. When E_b/N_0 is

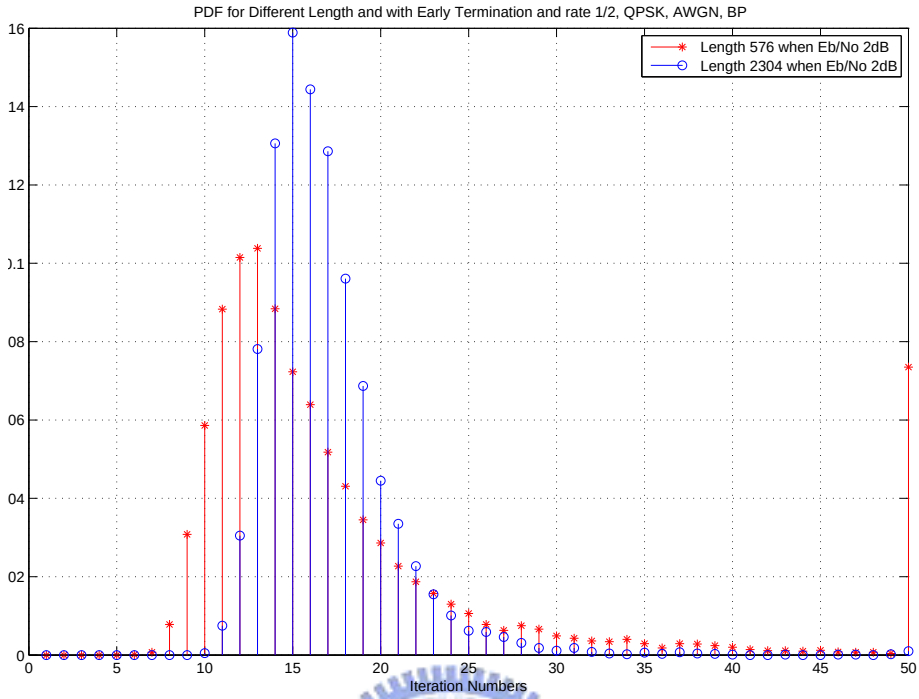


Figure 5.8: Distribution of iteration numbers for codes of different lengths.

2 dB, code rate $\frac{1}{2}$ and QPSK are applied. It shows the probability of iteration number 50 is about 0.075 for the pdf of length 576, but the one is almost zero for the pdf of length 2304. This is perhaps because the performance of length 2304 is better than the one of length 576, it does not need so many iterations to converge. Another effect is that the pdf of length 576 is more concentrated toward left than the one of length 2304. This is perhaps because when doing the early termination technique, we need to conform all 576 bits to the termination criteria for length 576, but for length 2304, we must conform all 2304 bits to the termination criteria. Hence, it is reasonable to do more iterations for length 2304.

Figure 5.9 shows the pdf of code rate $\frac{1}{2}$, length 576, and QPSK under different E_b/N_0 values. It shows almost all iteration number is 50 for length 576 when E_b/N_0 is 1 dB. But

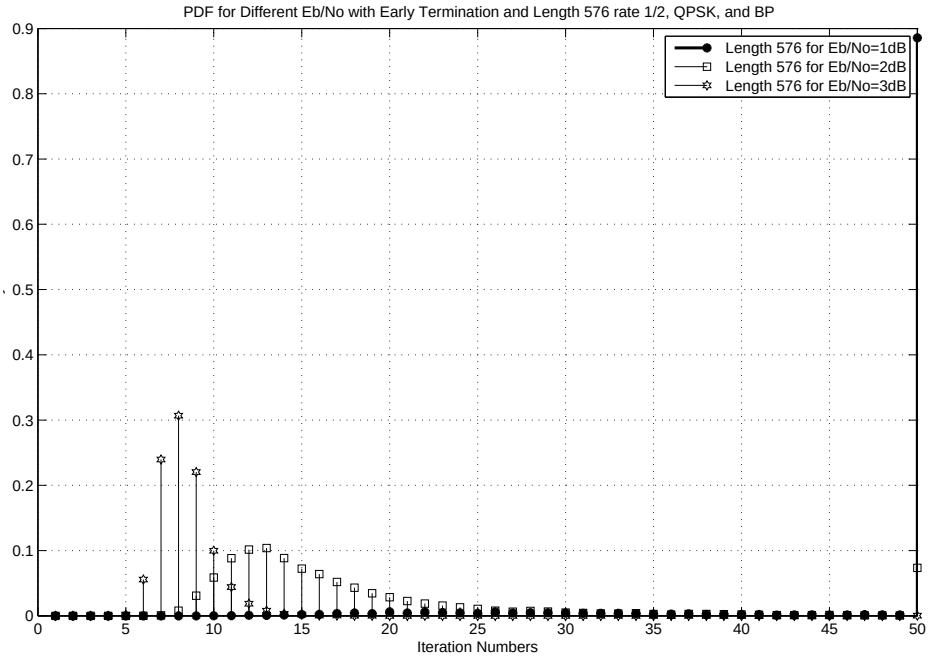


Figure 5.9: Distribution of iteration numbers at different SNR values.

after increasing the E_b/N_0 value, the pdf is toward to the left side, this means we need less iterations for larger E_b/N_0 values under the same length condition.

5.4.6 Compare Early Termination and Parity Check Termination

Generally, in the decoding iteration step, the decoded codeword checks with the parity check matrix to insure the decoded codeword is correct when the maximum iteration is not reached. If the syndrome is a zero vector, we stop the iteration to reduce the iteration number. If not, the decoding iteration continues until the maximum iteration is reached. Therefore, we can view the parity check step as a kind of early termination, so we name it "parity check termination".

In Figure 5.10, we compare the performance difference of early termination and parity

check termination when code rate $\frac{1}{2}$, length 576, QPSK, and BP decoding are applied. Obviously, from Fig. 5.10 there is almost no difference. But we just want to choose one kind of "early termination" technique to avoid the waste. Then we compare the iteration numbers in Fig. 5.11. As the E_b/N_0 is larger, "early termination" needs a slightly higher iteration number than the "parity check termination," but the early termination technique only needs "compare" operations and some space to store the temporary comparison results unlike parity check termination technique needs XOR operations and "compare" operations. Therefore, under the consideration of iteration number and computational complexity, we choose the early termination, not the parity check termination to early stop the decoding step.

5.4.7 Performance of Some Reduced-Complexity Decoding Algorithms [30]

Figures 5.15, 5.13, and 5.14 show the BER performance of different decoding algorithms for length 576, six code rates and three modulation types. The maximum iteration is 50 and early termination technique is used. Besides, the α parameter in normalized BP-based decoding is 1.25, and the β parameter in offset BP-based decoding is 0.25.

In some sub-figures, the BP-based decoding algorithm suffers a 0.3 or 0.4 dB degradation in performance, compared with BP decoding. When QPSK is applied, the two reduced-complexity algorithms have even a slightly better performance than the BP algorithm. These results are not surprising, because at medium or short code lengths, the BP algorithm is not optimum. This is because the number of short cycles in their Tanner graphs influences the BP decoding performance depended on the amount of correlation between messages, and the two reduced-complexity BP-based algorithms seem to outperform the BP algorithm by reducing the negative effect of correlations. The normalized BP-based algorithm slightly outperforms

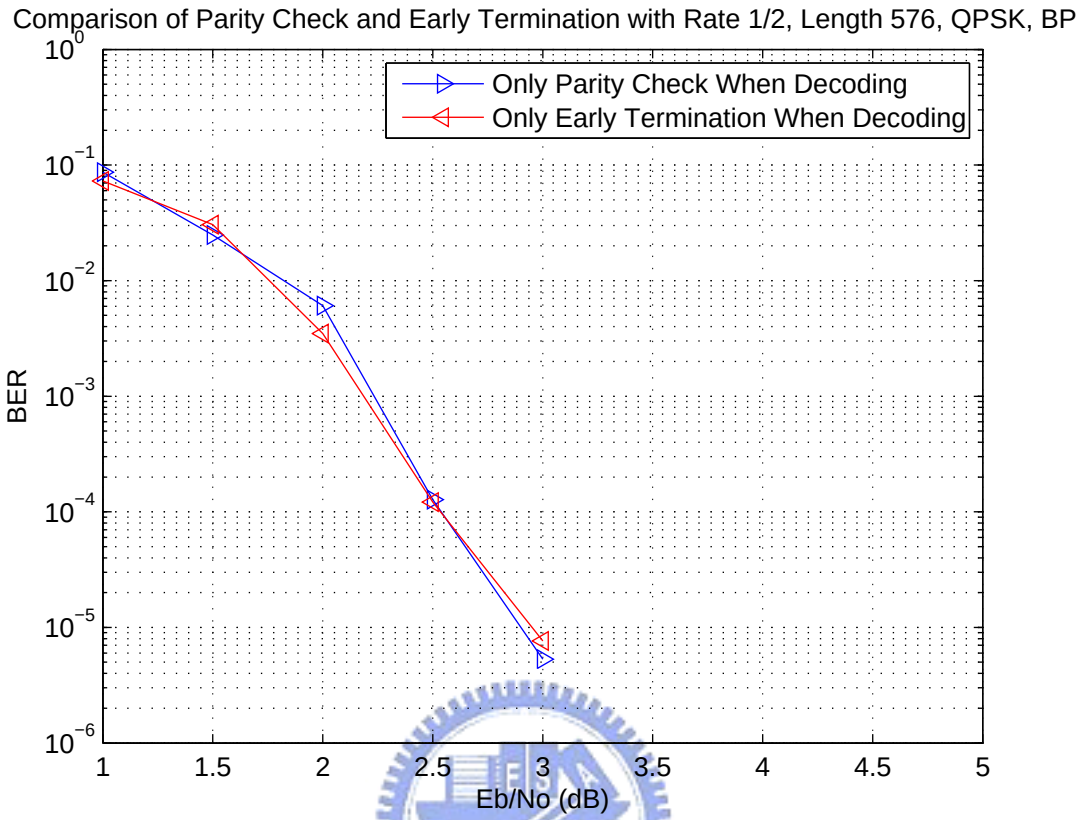


Figure 5.10: Comparison of the performance of parity check termination and early termination.

the offset BP-based algorithm, but may also be slightly more complex to implement. When 16QAM and 64QAM are applied, the BP approach has slightly better performance than the two improved approaches, but the performance of these three decoding approaches are very close.

Figure 5.15 shows the BER performance of different decoding algorithms for code types of $\frac{1}{2}$, $\frac{2}{3}A$, and $\frac{3}{4}B$. Each sub-figure has two different lengths, 576 and 2304, and one modulation type, 16QAM.

For rate $\frac{1}{2}$ and $\frac{2}{3}A$, their girth is 6 for both length 576 and 2304. For rate $\frac{3}{4}B$, its girth

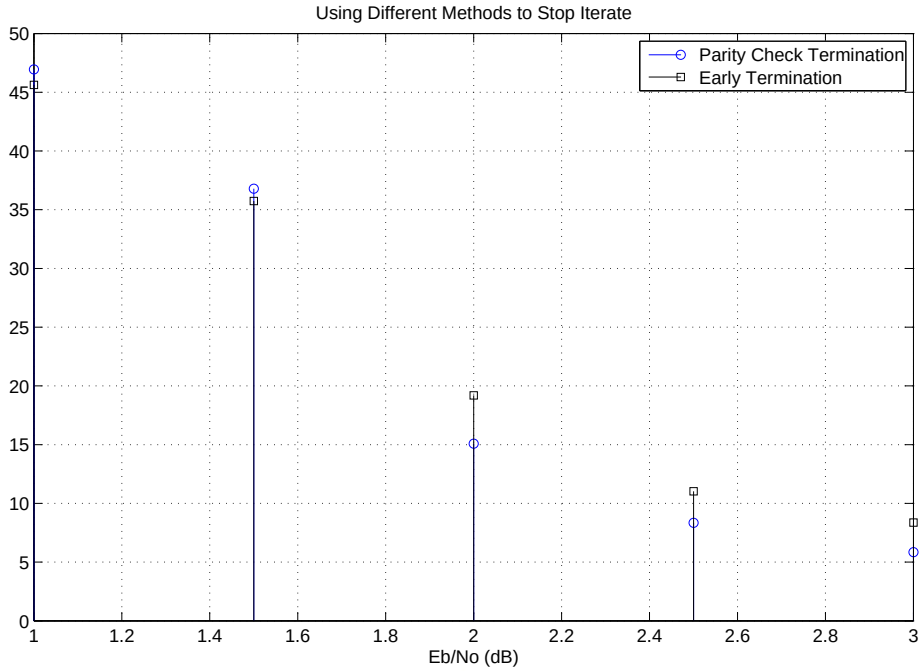


Figure 5.11: Comparison of the iteration numbers of parity check termination and early termination.

is 6 and 4 for length 576 and 2304 respectively. From Fig. 5.15, the performance of these two improved decoding does not always have better performance than BP decoding by the reason of girth 4 or 6 for length 576 and 2304. But their performance is still very close.

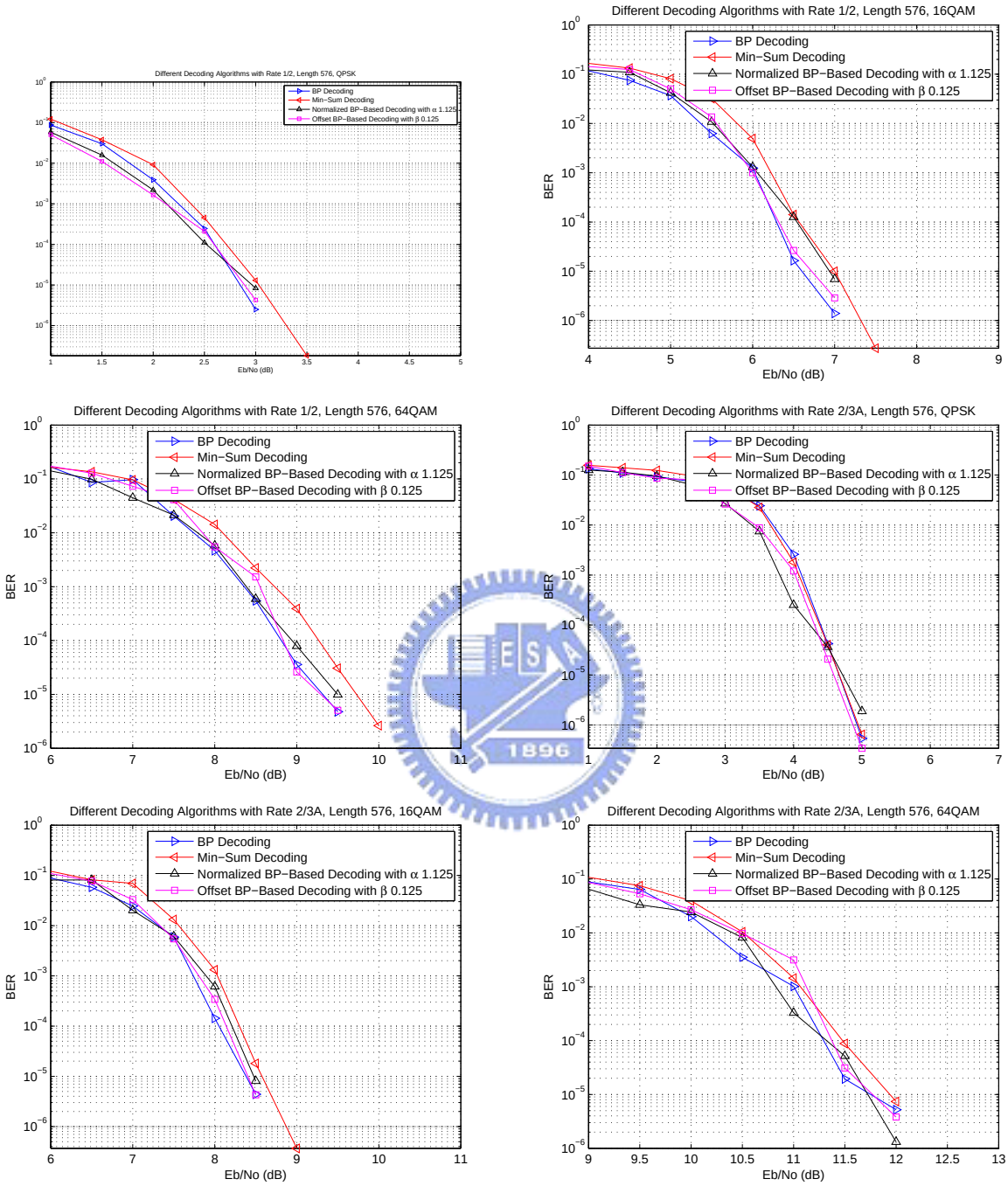


Figure 5.12: Performance of different decoding algorithms with rate $\frac{1}{2}$ and $\frac{2}{3}A$, length 576.

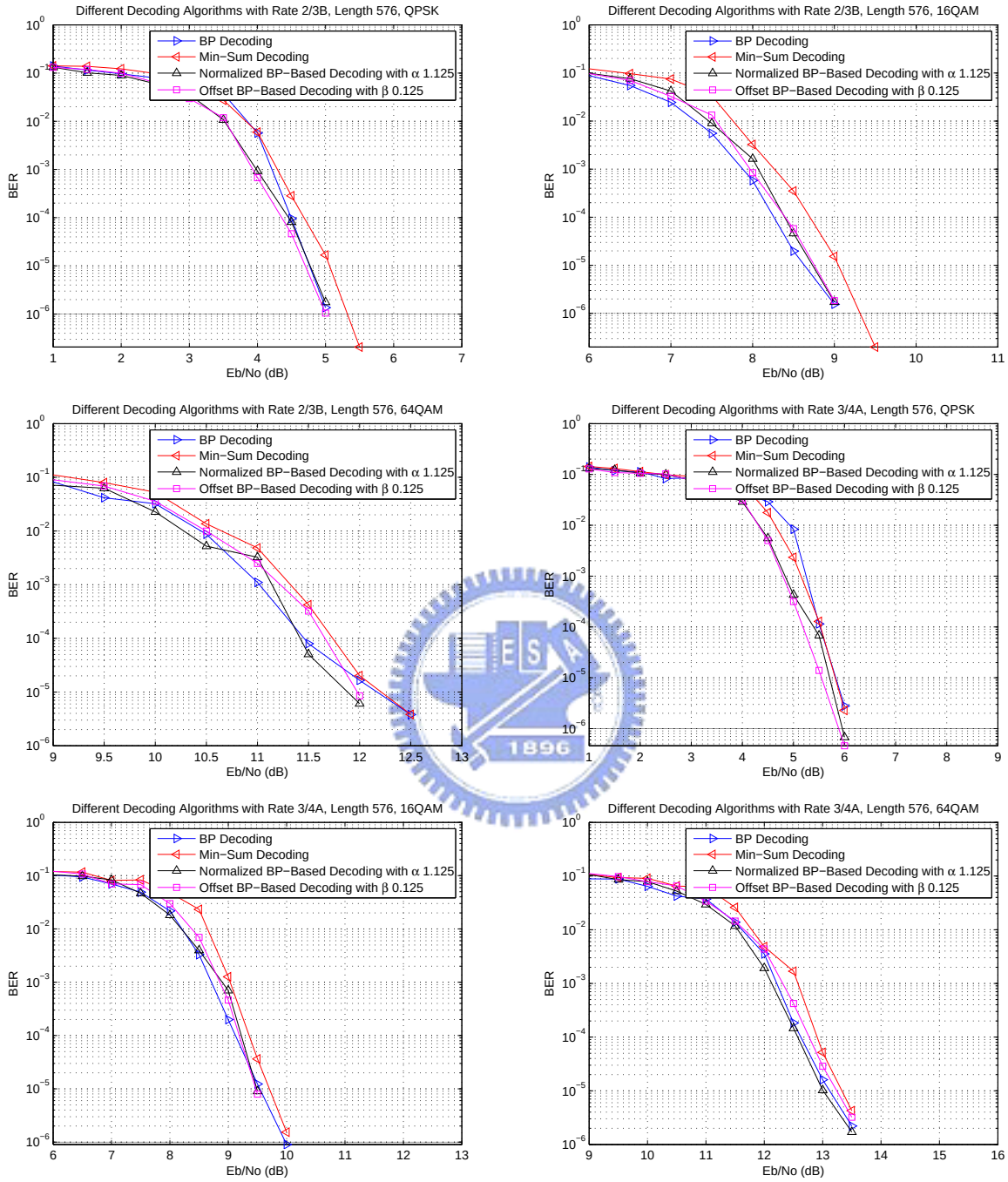


Figure 5.13: Performance of different decoding algorithms with rate $\frac{2}{3}B$ and $\frac{3}{4}A$, length 576.

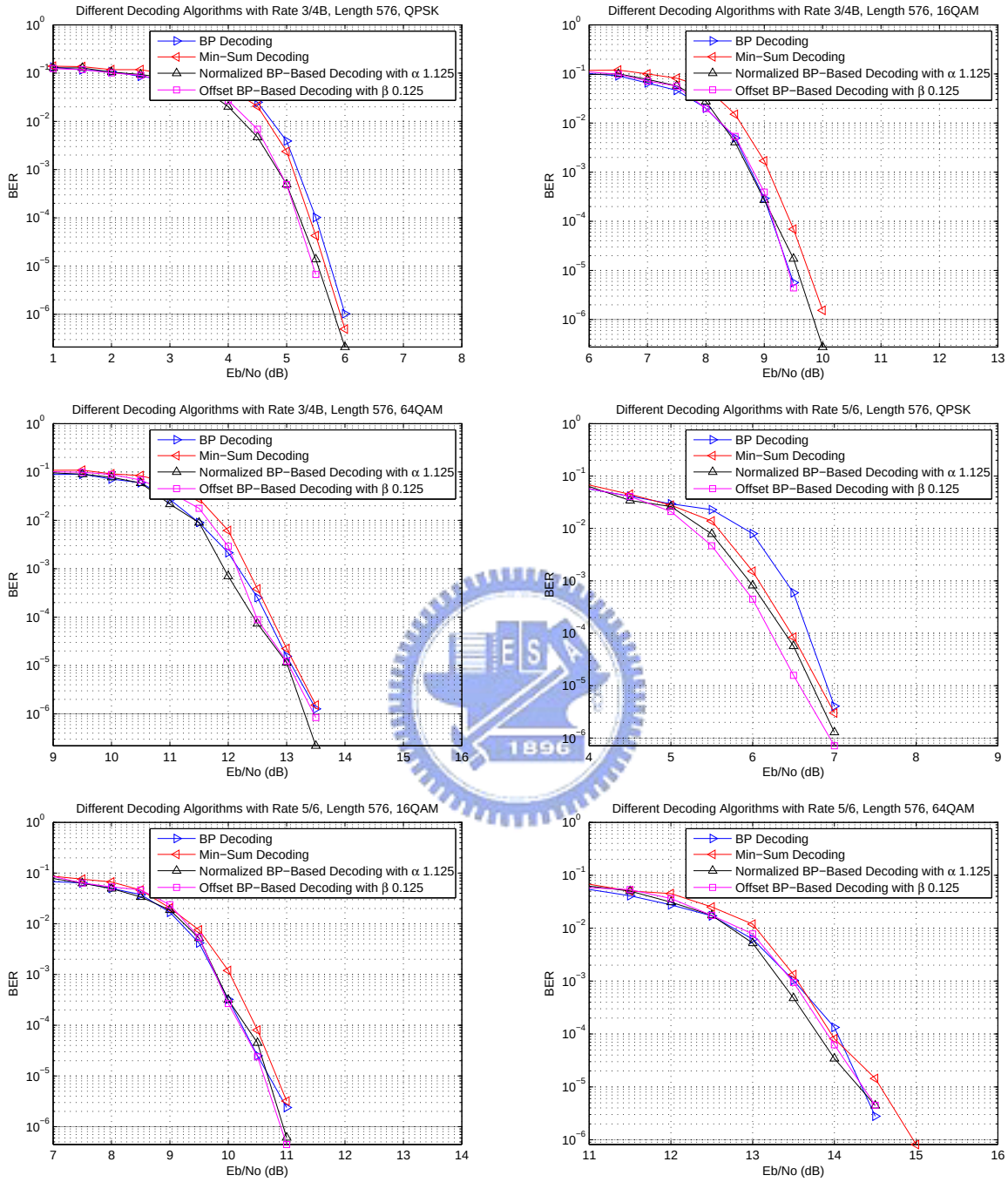


Figure 5.14: Performance of different decoding algorithms with rate $\frac{3}{4}B$ and $\frac{5}{6}$, length 576.

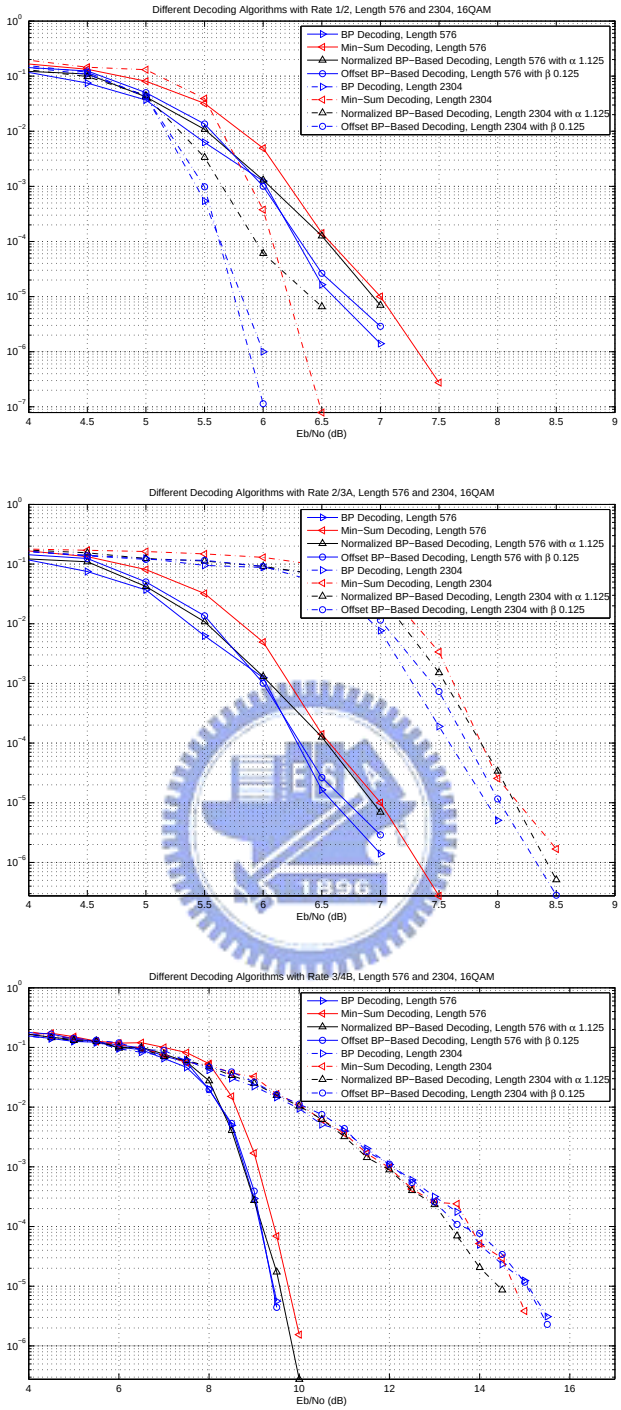


Figure 5.15: Performance of different decoding algorithms with rate $\frac{1}{2}$, $\frac{2}{3}A$, and $\frac{3}{4}B$.

Chapter 6

Conclusion and Future Work

This work studied two parts of IEEE802.16e: one was the implementation and optimization of 802.16e FEC scheme on DSP platform for WirelessMAN-OFDM and the other investigated the reduced-complexity decoding of the LDPC codes for WirelessMAN-OFDMA.

In the first part's work, the programs will require multiple DSPs to run in parallel to handle the data rate under a 10 MHz transmission bandwidth. Acknowledgeably, further optimization of the programs may be possible. In addition, the C64x is equipped with a Viterbi decoder co-processor [29]. Using this co-processor may be helpful in raising the decoding speed. But its use requires study and testing of the "enhanced direct memory access (EDMA)" mechanism of the C64x chips, we skipped this study in my thesis.

In the second part's work, first we analyzed the girth and threshold values in AWGN channel. Then, we evaluated the performance of LDPC codes and compared the results with the numerical results. Then we proposed a modified version BP algorithm based on algorithmic transformation to balance the computation load. Another topic is about the complexity reducing. We focused on two directions to reduce the complexity. One was to reduce the iteration numbers by using early termination technique. Another was to evaluate the performance by three kinds of approximate algorithms. The approximation

approaches used can lead to performance degradation but, interestingly, there appeared to be a dependence on the properties of the LDPC codes selected, such as the choice of modulation types. Our LDPC codes in IEEE 802.16e have a girth 6 at most, this is not a large enough number to make the conventional BP decoding approach optimal ([34] gives examples to derive a rate $\frac{1}{2}$ code with girth 14, which is large enough). Therefore, these simplified reduced-complexity decoding schemes sometimes can outperform the BP decoding algorithm and offer significant advantages for hardware implementation.

In the future work, we need to revise the coding algorithms to be fixed-point to reduce the complexity for actual DSP implementation. But the two improved decoding algorithms may not have as good performance as our simulation results. Besides, we need more realistic simulations in multipath channel to show how the LDPC codes are performed. In our before analysis, the performance of code rate type $\frac{2}{3}A$ was better than $\frac{2}{3}B$, and code rate type $\frac{3}{4}B$ was better than $\frac{3}{4}A$. But why did these two code rate types both exist? We guess that if in multipath channel simulation, not in AWGN channel, the performance of code rate type $\frac{2}{3}B$ is better than that of $\frac{2}{3}A$, and the performance of code rate type $\frac{3}{4}A$ is better than that of $\frac{3}{4}B$. But the exact answer should be done by more research.

About subsequent algorithm modifications, we had find some references. If we need further reducing complexity by other decoding algorithms, [35] is one of the references. If we need to remove the effects of cycles in the factor graph to make the BP decoding algorithm optimal or improve the decoding performance, [36] is one of the references.

Bibliography

[1] IEEE Std 802.16-2004, *IEEE Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Fixed Broadband Wireless Access Systems*. New York: IEEE, June 2004.

[2] IEEE Std 802.16e, *IEEE Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Fixed Broadband Wireless Access Systems*. New York: IEEE, Feb. 2006.

[3] I. S. Reed and X. Chen, *Error-Control Coding for Data Network*. Boston: Kluwer Academic Publishers, 1999.

[4] E. Zehavi, "8-PSK trellis codes for a Rayleigh channel," *IEEE Trans. Commun.*, vol. 40, pp. 873–884, May 1992.

[5] Yu-Ping Ho, "Study on OFDM signal description and channel coding in the IEEE 802.16a TDD OFDMA wireless communication standard," M.S. thesis, Department of Electronics Engineering National Chiao Tung University, June 2003.

[6] J. G. Proakis, *Digital Communication, 4th ed.* New York: McGraw-Hill, 2001.

[7] Marjan Karkooti, "Semi-parallel architectures for real-time LDPC coding," M.S. thesis, Rice Unversity, Houston, Texas, May 2004.

[8] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Information Theory*, vol. 8, pp. 21–28, Jan. 1962.

[9] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performace of low density parity check codes," *Electronics Letters*, vol. 32, no. 18, pp. 1645–1646, Aug. 1996.

[10] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Information Theory*, vol. 45, pp. 399–431, Mar. 1999.

[11] S. Chung, Jr. G. D. Forney, T. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Communications Letters.*, vol. 5, no. 2, pp. 58–60, Feb. 2001.

[12] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Information Theory*, vol. 27, no. 5, pp. 533–547, Sep. 1981.

[13] B. J. Frey F. R. Kschischang and H. A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Information Theory*, vol. 47, no. 2, pp. 498–519, Feb. 2001.

[14] S.Y. Chung, T.J. Richardson, and R. L. Urbanke, "Analysis of sum-product decoding of low-density parity-check codes using a Gaussian approximation," *IEEE Trans. Information Theory,* vol. 47, no. 2, pp. 657–670, Feb. 2001.

[15] W. Lin, X. Juan, and G. Chen, "Density evolution method and threshold decision for irregular LDPC codes," *Int. Conf. Commun. Circuits Systems*, vol. 1, June 2004, pp. 25–28.

[16] Innovative Integration, *Quixote Data Sheet.* http://www.innovative-dsp.com/support/datasheets/quixote.pdf.

[17] T.-S. Chiang, "Study and DSP implementation of IEEE 802.16a TDD OFDM downlink synchronization," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., July 2004.

[18] Texas Instruments, *TMS320C6000 CPU and Instruction Set*. Literature number SPRU189F, Oct. 2000.

[19] Texas Instruments, *TMS320C6000 DSP Cache User's Guide*. Literature number SPRU656A, May 2003.

[20] Innovative Integration, *Quixote User's Manual*. June 2004.

[21] Texas Instruments, *Code Composer Studio User's Guide*. Literature number SPRU328B, Feb. 2000.

[22] Texas Instruments, *TMS320C6000 Code Composer Studio Getting Started Guide*. Literature number SPRU509D, Aug. 2003.

[23] Texas Instruments, *TMS320C6000 Programmer's Guide*. Literature number SPRU198G, Oct. 2002.

[24] Y.-T. Lee, "DSP implementation and optimization of the forward error correction scheme in IEEE 802.16a standard," M.S. thesis, National Chiao Tung University, Dep. of Electronics Eng., Hsinchu, Taiwan, R.O.C., June 2004.

[25] S. Lin and D. J. Costello, Jr., *Error Control Coding — Fundamentals and Applications*. New Jersey: Prentice-Hall, 1983.

[26] J. H. Jeng and T. K. Truong,"On decoding of both errors and erasures of a reed-solomon code using an inverse-free Berlekamp-Massey algorithm," *IEEE Trans. Commun.*, vol. 47, pp. 1488–1494, Oct. 1999.

[27] Y.-P. Ho and D. W. Lin, "Study on channel coding in the IEEE 802.16a OFDMA wireless communication standard," *Int. J. Elec. Eng.*, vol. 11, no. 4, pp. 347–354, Nov. 2004.

[28] M. Lei and H. Harada, "Low-density parity-check coded ultra high-data-rate OFDM system in frequency-selective fading," *IEEE Vehicular Technology Conference*, vol. 3, June 2005, pp. 1590–1594.

[29] Texas Instruments, *TMS320C64x DSP Viterbi-Decoder Coprocessor (VCP) Reference Guide.* Literature no. SPRU533D, Sep. 2004.

[30] J. Chen, A. Dholakia, E. Eleftheriou, and M. P. C. Fossorier, and X.Y. Hu, "Reduced-complexity decoding of LDPC codes," *IEEE Trans. Commun.* , vol. 53, pp. 1288–1299, July 2005.

[31] Z. Wang, Y. Chen, and K. K. Parhi, "Area efficient decoding of quasi-cyclic low density parity check codes," *IEEE Int. Conf. Acoustics Speech Signal Processing*, vol. 5, May 2004, pp. 49–52.

[32] J. Chen, and M. Fossorier, "Near optimum universal belief propagation based decoding of low-density parity check codes," *IEEE Trans. Commun.* , vol. 50, no. 3, pp. 406–414, March 2002.

[33] T. Theocharides, G. Link, N. Vijaykrishnan, and M. J. Irwin, "Implementing LDPC decoding on network-on-chip," *18th Int. Conf. VLSI Design*, Jan. 2005, pp. 134–137.

[34] M. E. O'Sullivan, "Algebraic construction of sparse matrices with large girth," *IEEE Trans. Information Theory*, vol. 52, no. 2, pp. 718–727, Feb. 2006.

[35] J. Zhang, M. Fossorier, and D. Gu, "Two-dimensional correction for min-sum decoding of irregular LDPC codes," *IEEE Communications Letters.*, vol. 10, no. 3, pp. 180–182, Mar. 2006.

[36] D. Yongqiang, Z. Guangxi, L. Wenming, and M. Yijun, "An improved decoding algorithm of low-density parity-check codes," *IEEE Int. Conf. Wireless Commun. Networking Mobile Computing*, vol. 1, Sep. 2005, pp. 449–452.

作者簡歷

姓名：陳勇竹 (Yung-Chu Chen)

生日：1982 年 8 月 3 日

出生地：台南縣

學歷：交通大學管理科學系學士(2000.9~2004.6)

交通大學電信工程系學士(2000.9~2004.6)

交通大學電子研究所碩士(2004.9~2006.6)

研究領域：通訊系統、通道編碼及數位訊號處理

論文題目：IEEE 802.16e OFDM 與 OFDMA 通道編碼技術與數位訊號處理器實現之研究 (Research in Channel Coding Techniques and DSP Implementation for IEEE 802.16e OFDM and OFDMA)