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 OFDMA 通道編

碼技術與數位訊號處理器實現之研究

Research in and DSP Implementation of Channel Coding

Techniques for IEEE 802.16e OFDMA

研 究 生：吳柏昇

指導教授：林大衛 博士

中 華 民 國 九 十 六 年 六 月

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

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

研究生:吳柏昇 Student: Po-Sheng Wu

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 the Requirements

for the Degree of

Master of Science

in

Electronics Engineering

June 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年六月

IEEE 802.16e OFDMA 通道編

碼技術與數位訊號處理器實現之研究

研究生:吳柏昇　　　　　　　　　　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

IEEE 802.16e 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。

本篇論文前半部份重點在於，研究 IEEE 802.16e OFDMA 所訂定的迴旋編碼系統並且實現在數位訊號處理器(DSP)上，針對 DSP 平台的特性以及迴旋編碼編碼的演算法進行程式的改進。在論文中，我們將標準中制訂的四個必備的前向誤差改正編碼系統，利用 C 語言驗證我們整個系統演算法上的正確性，在加成性白色高斯通道下模擬了各種調變，模擬的結果增益比理論值大約有 1dB 的誤差，接著進一步以德州儀器公司所發展的 TMS320C6416 DSP 為核心的平台上實現。經過在 DSP 平台上最佳化我們的程式後，迴旋編碼的編碼器部份，於 DSP 模擬器上，可以到每秒 13793K 位元的處理速度，而解碼器的部份可以達到每秒 805K 位元的處理速度。

本論文後半部份重點，研究 IEEE 802.16e OFDMA 所訂定的低密度奇偶校驗碼系統並且實現在數位訊號處理器。研究低密度奇偶校驗碼傳統的編碼與解碼演算法，並且介紹一些降低解碼複雜度的演算法。用 C 語言驗證系統演算法上的正確性，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。模擬的結果顯示降低複雜度的演算法和傳統的解碼表現相當接近。接著從這些演算法中，根據運算複雜度，延遲時間，找出合適的演算法，實現在德州儀器公司所發展的 DSP 平台上。經過在 DSP 平台上最佳化我們的程式後，編碼器部份經過改進,可以到每秒 835K 位元的處理速度,而解碼器的部份僅可以達到每秒 4.7K 位元的處理速度。

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

Student: Po-Sheng Wu 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 the research of the convolutional code defined in IEEE 802.16e OFDMA 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 C program to insure the correctness of our algorithm. We simulate the different modulation in AWGN channel and the coding gain is almost achieve theoretic values. Then we implement the project on the Texas Instruments digital signal processor (DSP). After optimizing the programs on the DSP platform, the improved FEC encoder can achieve a data processing rate of 13793 kbps and the improved FEC decoder can achieve a processing rate of 805 kbps on the TI TMS320C6416 DSP simulator.

The focus of second part is the low-density parity-check (LDPC) code defined in IEEE 802.16e OFDMA. We explain the conventional encoding and decoding algorithm, and some reduced-complexity decoding algorithms. We simulate the LDPC code for different 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. According to computational complexity and latency, we choose the adaptable algorithm and implement on DSP. After optimizing the programs on the DSP platform, the improved encoder can achieve a data processing rate of 835 kbps and the improved decoder can achieve a processing rate of 4.7 kbps on the TI C6416 DSP simulator.

誌謝

本篇論文的完成，誠摯地感謝我的指導老師 林大衛 博士，從踏入交通大學電子所開始，多虧老師的循循善誘，不但給予我在課業、研究上的幫助，使我學到了分析問題及解決問題的能力。同時老師樂觀的生活態度也影響了我，讓我更有勇氣面對各種困難。在此，僅向老師及老師的家人致上最高的感謝之意。

另外要感謝的，是實驗室的洪崑健學長和吳俊榮學長。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 93 級國偉、治傑、勇竹三位學長的指導，以及 94 級介遠、志岡、政達、耀鈞、順成、凱庭、錫祺、浩廷、育成、耀仚等實驗室成員，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鶩的從事研究工作。

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 2007.6 風城交大

柏昇

Contents

1 Introduction **1**

1.1 Scope of the Work 1

1.2 Organization of This Thesis 2

2 FEC in IEEE 802.16e OFDMA and Associated Decoding methods **3**

2.1 Convolutional Code Specifications [1] 3

2.1.1 Randomizer [1] 5

2.1.2 Convolutional Encoder [1] 6

2.1.3 Interleaver [1] 8

2.1.4 Modulation [1] 10

2.2 Decoding Under Convolutional Encoding 10

2.2.1 Demodulation Under Bit-Interleaved Coded Modulation 11

2.2.2 De-Interleaver 14

2.2.3 Tail-Biting Convolutional Decoding 15

2.3 LDPC Code Specifications 16

2.3.1 Overview of LDPC Code 18

2.3.2 LDPC Code in IEEE 802.16e OFDMA [1] 20

2.4 Decoding of LDPC code 21

2.4.1 The Belief Propagation Decoding Algorithm [17] 21

2.4.2 Some Reduced-Complexity LDPC Decoding Algorithms 24

3 DSP Implementation Environment 28

3.1 The DSP Baseboard (SMT395) 28

3.2 The DSP Chip 29

3.2.1 Central Processing Unit [23] 32

3.2.2 Memory [24] 37

3.3 TI's Code Development Environment [25], [26] 39

3.4 Code Development Flow [27] 41

3.5 Acceleration Rules 43

3.5.1 Compiler Optimization Options [27] 43

3.5.2 Fixed–Point Coding 45

3.5.3 Loop Unrolling 45

3.5.4 Packet Data Processing 46

3.5.5 Register and Memory Arrangement 47

3.5.6 Software Pipelining 47

3.5.7 Macros and Intrinsic Functions 48

3.5.8 Other Acceleration Rules 48

4 Simulation and DSP Implementation of Convolutional Encoder and Decoder **49**

4.1 Coding Gain Analysis . . . 49

4.2 Performance in AWGN with Floating-Point Processing . . . 52

4.3 Performance in AWGN with Fixed-Point Processing . . . 55

4.4 Implementation on DSP . . . 61

4.4.1 Profile of the DSP code . . . 62

5 Simulation and DSP Implementation of LDPC Encoder and Decoder **71**

5.1 Performance in AWGN Channel with Floating-Point Processing . . . 71

5.1.1 Number of Iterations . . . 71

5.1.2 Performance at Different Codeword Lengths . . . 72

5.1.3 Performance with Different Modulations . . . 72

5.1.4 Performance at Different Coding Rates . . . 74

5.1.5 Performance of Reduced-Complexity Algorithm . . . 76

5.2 Performance in AWGN Channel with Fixed-Point Processing . . . 77

5.2.1 Profile of the DSP code . . . 81

6 Conclusion and Future Work **92**

Bibliography **94**

List of Figures

2.1 Convolutional coding structure in transmitter (top path) and decoding in receiver (bottom path). 4

2.2 PRBS for data randomization (from [1]). 5

2.3 Convolutional encoder of rate 1/2 (from [1]). 6

2.4 The second permutation of interleaver. 9

2.5 QPSK, 16-QAM, and 64-QAM constellations (from [1]). 10

2.6 Metric partitions of the 16-QAM constellation (from [9]). 14

2.7 Trellis for tail-biting convolutional decoding (from [2]). 16

2.8 LDPC coding structure in transmitter (top path) and decoding in receiver (bottom path). 17

2.9 Tanner graph of a parity check matrix 19

2.10 Base model of the rate-1/2 code (from [1]). 21

2.11 Base model of the rate-2/3, type A code (from [1]). 21

2.12 Base model of the rate-2/3, type B code (from [1]). 22

2.13 Base model of the rate-3/4, type A code (from [1]). 22

2.14 Base model of the rate-3/4, type B code (from [1]). 22

2.15 Base model of the rate-5/6 code (from [1]). 22

2.16 Fast decaying function $f(x) = \log\frac{e^x+1}{e^x-1}$. 25

3.1 SMT395 Module. 29

3.2 Block diagram of TMS320C6416 DSP (from [23]). 31

3.3 The TMS320C64x DSP chip architecture and comparison with earlier TMS320C62x/C67x chip (from [23]). 33

3.4 Pipeline phases of TMS320C6416 DSP (from [23]). 34

3.5 Execution stage length description for each instruction type (from [23]). . . . 35

3.6 TMS320C64x CPU data paths (from [23]). 38

3.7 C64x cache memory architecture (from [24]). 39

3.8 Code development flow for TI C6000 DSP (from [27]). 42

3.9 Loop unrolling. 46

3.10 The block diagram of SIMD. 47

3.11 Software-pipelined loop. 48

4.1 Soft-decision decoding performance of rate–1/2 coding in AWGN with different value of α and β employing floating-point computation. 53

4.2 oft-decision decoding performance of rate–2/3 and rate–3/4 coding in AWGN with different value of α and β employing floating-point computation. 54

4.3 Soft-decision decoding performance in AWGN employing floating-point computation with $\alpha = \beta = 48$. 55

4.4 Soft-decision decoding performance in AWGN with different input precisions. 57

4.5 Soft-decision decoding performance employing fixed-point computation in AWGN with different value α and β. 58

4.6 Soft-decision decoding performance in AWGN with $\alpha = 48$ and $\beta = 48$ employing fixed-point computation. 59

4.7 Comparison between soft-decision decoding performance in AWGN using floating-point computation and that using fixed-point computation. 60

4.8 The C code of Viterbi decoder. 63

4.9 The assembly code of Viterbi decoder (1/5). 64

4.10 The assembly code of Viterbi decoder (2/5). 65

4.11 The assembly code of Viterbi decoder (3/5). 66

4.12 The assembly code of Viterbi decoder (4/5). 67

4.13 The assembly code of Viterbi decoder (5/5). 68

4.14 Software pipeline information for Viterbi decoder. 69

5.1 LDPC decoding performance in different iteration numbers with floating-point computation. 72

5.2 LDPC decoding performance in different codeword length with floating-point computation. 73

5.3 LDPC decoding performance with different modulation employing floating-point computation. 73

5.4 LDPC Decoding Performance in Different Coding Rate (floating-point). . . . 75

5.5 LDPC decoding performance using different decoding algorithm employing floating-point computation. 76

5.6 LDPC decoding performance at different bit numbers with different modulations employing fixed-point computation. 78

5.7 LDPC decoding performance at different bit numbers at two different coding rate employing fixed-point computation. 79

5.8 LDPC decoding performance at different bit numbers at two different codeword lengths employing fixed-point computation. 80

5.9 The C codes of circular shift. 83

5.10 The assembly codes of circular shift (1/2). 84

5.11 The assembly codes of circular shift (2/2). 85

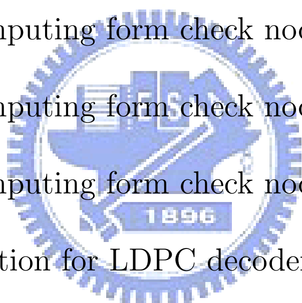
5.12 The C code of computing form check nodes to bit nodes. 86

5.13 The assembly code of computing form check nodes to bit nodes (1/3). 88

5.14 The assembly code of computing form check nodes to bit nodes (2/3). 89

5.15 The assembly code of computing form check nodes to bit nodes (3/3). 90

5.16 Software pipeline information for LDPC decoder. 91



List of Tables

2.1 Mandatory Channel Coding Schemes for Each Modulation Method 4

2.2 The Convolutional Code with Puncturing Configuration 7

2.3 Bit Interleaved Block Sizes and Modulos 9

2.4 Bit Metric for Method-ML and Method-LLR 13

2.5 Comparison of Main Operations of Different Decoding Algorithms 27

3.1 Functional Units and Operations Performed (from [23]) 36

3.2 Sizes of Different Data Types 45

3.3 Comparison Between Unrolled and not Unrolled 46

4.1 Coding Gain Upper-Bound in AWGN at BER $= 10^{-6}$ 51

4.2 Approximate Coding Gain Based on Analysis of Minimum Codeword Distance 52

4.3 Comparison of Convolutional Coding Gain froms in AWGN at BER $= 10^{-6}$ 56

4.4 Soft-Decision Decoding Performance with $\alpha = 48$ and $\beta = 48$, in AWGN at BER $= 10^{-6}$ Employing Fixed-Point Computation 61

4.5 Final Profile of Convolution Code (Cycles) 69

4.6 Final Profile of Convolution Code (Processing Rate) 70

4.7 Final Profile of Convolution Code (Code Size) 70

5.1 Comparison of Coding Gain Between LDPC Codes and Convolutional Codes at Code Rate 1/2 in AWGN at BER = 10^{-6} 74

5.2 Threshold for Each Code Rate under BPSK Modulation in AWGN Channel [20]. 75

5.3 LDPC Coding Gain between Floating-point and Fixed-point in AWGN at BER = 10^{-5}. 77

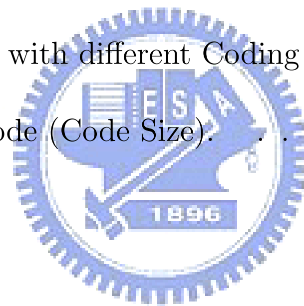
5.4 Original Profile of LDPC Encoder (Cycles) 82

5.5 Profile of LDPC Encoder with Matrix Table (Cycles) 82

5.6 Profile of LDPC Encoder with Different Coding Rates 82

5.7 Profile of LDPC Decoder with different Coding Rate 83

5.8 Final Profile of LDPC Code (Code Size). 87



Chapter 1

Introduction

1.1 Scope of the Work

Digital wireless transmission is a trend in the next generation of consumer electronics. Due to this demand high data transmission rate and mobility are needed. The OFDM modulation technique for wireless communication has been a main stream in recent years. IEEE has completed several standards, including the 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 multiple access systems providing multiple access.

In wireless communication, the transmitted signals are easily interfered and distorted by variance things sources such as the crowd traffic, bad weather, the obstacle of buildings, etc. Digital wireless transmission with multimedia contents such as audio and video is a trend. These services often exhibit high data rates and require high quality reproduction. To improve the robustness of the wireless communication against the noisy channel condition, the FEC (forward-error-correcting coding) mechanism is a must in almost every commercial communication standard, including the IEEE 802.16e.

The mandatory channel coding scheme in IEEE 802.16e for OFDMA employs punctured

convolutional coding. In addition, bit interleaver and M-ary QAM modulation are used after coding. We also discuss the LDPC code in IEEE 802.16e for OFDMA.

In this thesis, we focus on the study of the simulation and the DSP implementation of the FEC schemes of the IEEE 802.16e standard. We first review the FEC methods used in IEEE 802.16e and study the encoding and decoding techniques. Then we perform computer simulation to investigate the coding performance. Finally, we implement the FEC algorithms on DSP with fixed-point computation. We also seek to optimize the DSP program for efficient execution.

1.2 Organization of This Thesis

This thesis is organized as follows.

- Chapter 2 introduces the convolutional code and the LDPC code of IEEE 802.16e.
- Chapter 3 describes the DSP implementation environment.
- Chapter 4 discusses simulation and the DSP implementation of the convolution code.
- Chapter 5 discusses simulation and the DSP implementation of the LDPC code.
- Chapter 6 contains the conclusion and points out some future work.

Chapter 2

FEC in IEEE 802.16e OFDMA and Associated Decoding methods

The channel coding schemes usually used in IEEE 802.16e is tail-biting convolutional code. Block turbo code, convolutional turbo code, zero tailed convolutional code and LDPC code are the options.

2.1 Convolutional Code Specifications [1]

The contents of this section have been taken a large extent from [2].

The mandatory channel coding scheme used in IEEE 802.16e OFDMA is as shown in Fig. 2.1. Input data streams are divided by the randomizer to clean up the bit correlation, and then each data block is encoded by the convolutional encoder. The block-by-block coding makes the convolutional code effectively a block code.

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-interleaved coded modulation (BICM)" in the literature [3].

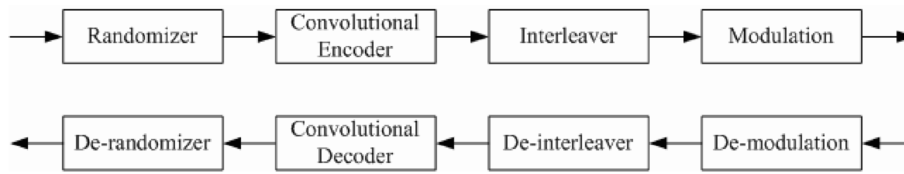


Figure 2.1: Convolutional coding structure in transmitter (top path) and decoding in receiver (bottom path).

Table 2.1: Mandatory Channel Coding Schemes for Each Modulation Method

Modulation	Uncoded Block Size (bytes)	Overall Code Rate	Coded Block Size (bytes)	Number of Used Sub-channels
QPSK	6	1/2	12	1
QPSK	12	1/2	24	2
QPSK	18	1/2	36	3
QPSK	24	1/2	48	4
QPSK	30	1/2	60	5
QPSK	36	1/2	72	6
QPSK	9	3/4	12	1
QPSK	18	3/4	24	2
QPSK	27	3/4	36	3
QPSK	36	3/4	48	4
16QAM	12	1/2	24	1
16QAM	24	1/2	48	2
16QAM	36	1/2	72	3
16QAM	18	3/4	24	1
16QAM	36	3/4	48	2
64QAM	18	1/2	36	1
64QAM	36	1/2	72	2
64QAM	24	2/3	36	1
64QAM	27	3/4	36	1

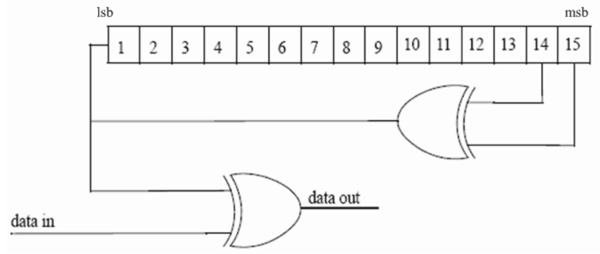


Figure 2.2: PRBS for data randomization (from [1]).

To make the system more flexibly adaptable to the channel condition, nineteen coding-modulation schemes are defined in IEEE 802.16e, as shown in Table 2.1. The different coding rates are made by puncturing of the native convolutional code. The puncturing mechanism in convolutional coding can provide variable code rates through one convolutional encoder.

2.1.1 Randomizer [1]

The randomizer is a pseudo random binary sequence (PRBS) generator, as depicted in Fig. 2.2. If the amount of data to transmit does not fit exactly the amount of data allocated, padding of 0xFF ("1" only) shall be added to the end of the transmission block, up to the amount of data allocated. The shift-register of the randomizer shall be initialized for every 1250 bytes passed through (if the allocation is larger then 1250 bytes).

The randomizer sequence is applied only to information bits. Preambles are not randomized.

Both in the uplink and downlink, the randomizer shall be re-initialized at the start of each frame with the sequence

(lsb) 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 (msb).

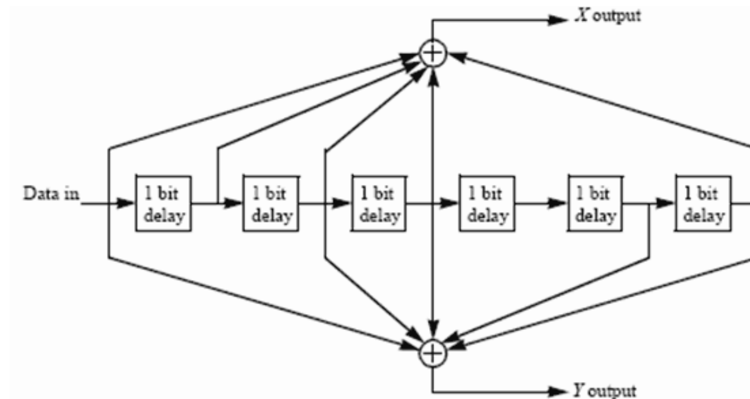


Figure 2.3: Convolutional encoder of rate 1/2 (from [1]).

2.1.2 Convolutional Encoder [1]

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

The coded bits may be punctured to allow different rates, which is known as rate-compatible punctured convolutional coding (RCPC). Furthermore, tail-biting is performed, by initializing the encoder's memory with the last data bits of the block. The encoding algorithm and the decoding algorithm (based on Viterbi decoder) for the RCPC with tail-biting convolutional are discussed late.

Punctured Convolutional Code

Puncturing patterns and serialization order of the convolutional code in IEEE 802.16e are as defined in Table 2.2. In this table, "1" means a transmitted bit and "0" a removed bit, whereas X and Y are in reference to Fig. 2.3. Note that the D_{free} after puncturing is lower than that of the native convolutional code at rate 1/2, which is equal to 10 [7, Chapter 8].

Table 2.2: The Convolutional Code with Puncturing Configuration

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

Tail-Biting

The convolutional code in IEEE 802.16e is terminated in a block, and thus becomes a block code. In general, there are three methods to achieve code termination[4]. For ease of understanding, we describe these methods in terms of a binary (n, k, m) convolutional code (of rate k/n and register length m) for an information sequence length of L bits.

- Direct truncation. The codeword is produced by inputting into the encoder (initialized with all zeros) L information bits, so the codeword length is nL/k. However, this code has the disadvantage that there is little error protection ability afforded to the last information bits.
- Zero tail. The codeword is produced by inputting into the encoder (initialized with all zeros) L information bits followed by m zeros (tail bits), so the codeword length is $n(L + m)/k$. However, this code has the disadvantage of rate loss of $m/(L + m)$ since the effective rate is $(k/n)(L/(L + m)) = (k/n)(1 - m/(L + m))$.
- Tail biting. We first initialize the encoder with the last m information bits, and then inputting into the encoder L information bits to produce codeword whose length is nL/k. This code has the disadvantage of complex Viterbi decoder since the starting and ending states of the trellis are unknown.

IEEE 802.16e uses the tail-biting approach, which has better performance compared with direct-truncation convolutional code and does not lose rate compared with zero-tail convolutional code. However, we pay the price of a complex decoder. The optimal decoder of tail-biting convolutional code, as suggested in [4], is to run M parallel Viterbi decoders, where $M = 2^m$ is the number of states in the trellis. Each Viterbi decoder postulates a different starting and ending state. The Viterbi decoder that produces the globally best metric gives the maximum likelihood estimate of the transmitted bits. The obvious disadvantage of this method is the M times complexity compared to decoding for the code with zero tail bits. Therefore, we consider a suboptimal decoder which can reduce the complexity to less than 2 times the normal Viterbi algorithm. This decoder combines the algorithms proposed in [5] and [6]. We introduce it later.

Another interesting property is the error rates at different positions in the codeword, which are analyzed in [5] and [6]. In zero-tail convolutional code, there is lower error rate in the first and the last information bits because the decoder knows the starting and ending states in the trellis. In tail-biting convolutional code, if the suboptimal decoder is adopted, there is almost equal error rate through the codeword when the parameters used in the decoder are proper.

2.1.3 Interleaver [1]

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 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.

Table 2.3: Bit Interleaved Block Sizes and Modulos

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

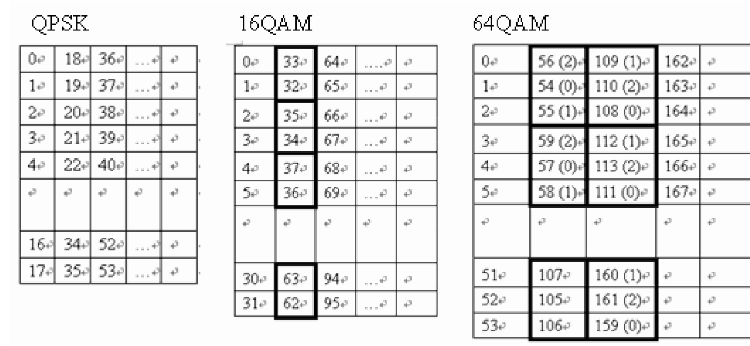


Figure 2.4: The second permutation of interleaver.

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

$$m = (\frac{N_{cbps}}{d}) \cdot \ k_{mod(d)} + floor(\frac{k}{d}), \quad k = 0, 1, \cdots, N_{cbps} - 1, \tag{2.1}$$

and the second permutation is defined by

$$j = s \cdot \ floor(\frac{m}{s}) + (m + N_{cbps} - floor(\frac{d \cdot m}{N_{cbps}}))_{mod(s),} \quad m = 0, 1, \cdots, N_{cbps} - 1. \tag{2.2}$$

The first permutation is a block interleaving. And in Fig. 2.4, we show the second permutation after the block interleaving.

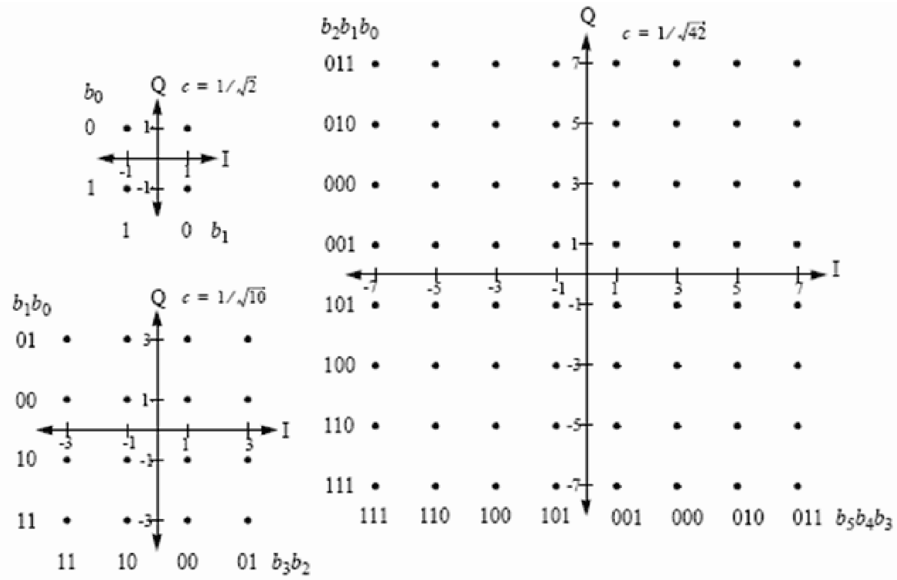


Figure 2.5: QPSK, 16-QAM, and 64-QAM constellations (from [1]).

2.1.4 Modulation [1]

After bit interleaving, the data bits are entered serially to the constellation mapper. Gray-mapped QPSK and 16-QAM are supported, whereas the support of 64-QAM is optional. The constellations as shown in Fig. 2.5 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 Decoding Under Convolutional Encoding

For Viterbi decoder, there are two decision types: hard-decision and soft-decision. If hard-decision is adopted, the metric used in Viterbi 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. However, the coding gain will lose 2 to 3 dB compared to soft-decision decoding [7, Chapter 8]. Hence soft-decision is adopted in our study.

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. Indeed, the puncturing mechanism adds further complexity to the super-trellis structure. Thus, we consider a suboptimal decoder based on bit-by-bit metric computation, which is proposed in [3], [8], and [9].

2.2.1 Demodulation Under Bit-Interleaved Coded Modulation

Let $a[i] = a_I[i] + ja_Q[i]$ denote the QAM symbol transmitted in the ith sub-carrier of OFDMA symbol and $\{b_{I,1}, \cdots, b_{I,k}, \cdots, b_{I,t}, b_{Q,1}, \cdots, b_{Q,k}, \cdots, b_{Q,t}\}$ be the corresponding bit sequence. Assuming that the ISI (inter–OFDMA symbol interference) and ICI (inter–channel interference) are completely eliminated, then the received signal of the sub-carrier can be written as

$$r[i] = G_{ch}[i] \cdot a[i] + w[i], \tag{2.3}$$

where $G_{ch}[i]$ is the channel frequency response complex coefficient for the ith sub-carrier and $w[i]$ is the complex additive white Gaussian noise (AWGN) with variance $\sigma^2 = N_0$. If the channel estimate is error free, the output of the one-tap equalizer is given by

$$y[i] = a[i] + w[i]/G_{ch}[i] = a[i] + w'[i], \tag{2.4}$$

where $w'[i]$ is still complex AWGN noise with variance $\sigma'^2(i) = \sigma^2/|G_{ch}[i]|^2$.

According to the MAPSE (maximum a posterior sequence estimation) criterion, the following maximization should be performed to estimate the encoded bit sequence $\mathbf{b}$:

$$\hat{\mathbf{b}} = \arg\max_{\mathbf{b}} P[\mathbf{b}|\mathbf{r}], \tag{2.5}$$

where $\mathbf{r}$ is the received sequence of QAM signals. Assume that the transmitted symbols are equally distributed. Then the MAPSE criterion can be replaced by the ML (maximum likelihood) criterion as:

$$\hat{\mathbf{b}} = \arg\max_{\mathbf{b}} P[\mathbf{r}|\mathbf{b}]. \tag{2.6}$$

We further assume that $G_{ch}[i]$ is known to the receiver and that the transmitted bits are i.i.d.

For each in-phase or quadrature bit (i.e., $b_{I,k}$ or $b_{Q,k}$), two metrics can be derived corresponding to the two possible values 0 and 1,respectively. For bit $b_{I,k}$, first the QAM constellation is split into two partitions of complex symbols, namely $S_{I,k}^{(0)}$ comprising the symbols with a "0" in position (I,k) and $S_{I,k}^{(1)}$, which is complementary. Then the two metrics are obtained by

$$m'_c(b_{I,k}) = \sum_{\alpha \in S_{I,k}^{(c)}} \log p(r[i]|a[i] = \alpha) \approx \max_{\alpha \in S_{I,k}^{(c)}} \log p(r[i]|a[i] = \alpha), \quad c = 0, 1. \tag{2.7}$$

Since the conditional pdf of $r[i]$ is complex Gaussian as

$$p(r[i]|a[i] = \alpha) = \frac{1}{\sqrt{2\pi\sigma}} \exp\{-\frac{1}{2}\frac{|r[i] - G_{ch}[i]\alpha|^2}{\sigma^2}\} \tag{2.8}$$

and $r[i] = G_{ch}[i] \cdot y[i]$, the metrics defined in (2.7) are equivalent to

$$m_c(b_{I,k}) = |G_{ch}[i]|^2 \cdot \min_{\alpha \in S_{I,k}^{(c)}} |y[i] - \alpha|^2. \tag{2.9}$$

Finally, these metrics are de-interleaved, i.e., each couple (m_0, m_1) is assigned to the bit position in the decoded sequence according to the de-interleaver map, and fed to the Viterbi decoder which selects the binary sequence with the smallest cumulative sum of metrics. We name this method *Method-ML* in the following discussion.

From the concept of log-likelihood ratio (LLR), a method named *Method-LLR* is proposed

Table 2.4: Bit Metric for Method-ML and Method-LLR

	Method-ML	Method-LLR
Bit metric (decided "0")	m_0	$[\frac{1}{4}(m_0 - m_1) + 1)]^2$
Bit metric (decided "1")	m_1	$[\frac{1}{4}(m_0 - m_1) - 1)]^2$

in [9] to reduce the complexity of *Method-ML*. It defines $LLR(b_{I,k})$ as

$$\begin{aligned} LLR(b_{I,k}) &\triangleq \frac{|G_{ch}[i]|^2}{4}\{\min_{\alpha \in S_{I,k}^{(0)}} |y[i]-\alpha|^2 - \min_{\alpha \in S_{I,k}^{(1)}} |y[i]-\alpha|^2\} \\ &\triangleq (m_0(b_{I,k}) - m_1(b_{I,k}))/4 \\ &\triangleq |G_{ch}[i]|^2 \cdot D_{I,k}. \end{aligned} \tag{2.10}$$

The quadrature part is similarly defined. The metrics sent to the Viterbi decoder of the two methods are defined in Table 2.4. Note that the difference between the bit metrics for the decided "0" and "1" is the same for the two methods, namely $\pm(m_0 - m_1)$. Thus the decoded bit sequence will be the same for the two methods.

In *Method-LLR*, only $(m_0 - m_1)/4$ is sent to the de-interleaver while in *Method-ML*, both m_0 and m_1 are sent. Besides, we can reduce $(m_0 - m_1)/4 = |G_{ch}[i]|^2 \cdot D_{I,k}$ to a simple form constituting of $y_I[i]$ itself because Gray-coding is used in the constellation map of M-ary QAM modulation in IEEE 802.16e.

Figure. 2.6 shows the partitions $(S_{I,k}^{(0)}, S_{I,k}^{(1)})$ for the generic bit $b_{I,k}$ in the case of the 16-QAM constellation. As a consequence,

$$D_{I,k} = \frac{1}{4}\{\min_{\alpha \in S_{I,k}^{(0)}} |y[i]-\alpha|^2 - \min_{\alpha \in S_{I,k}^{(1)}} |y[i]-\alpha|^2\}$$

can be simplified as follows.

$$D_{I,1} = \left\{ \begin{array}{ll} -y_I[i], & |y_I(i)| \le 2 \\ -2(y_I[i]-1), & y_I(i) > 2 \\ -2(y_I[i]+1), & y_I(i) < 2 \end{array} \right\} \cong -y_I[i], \tag{2.11}$$

$$D_{I,2} = |y_I[i]| - 2. \tag{2.12}$$

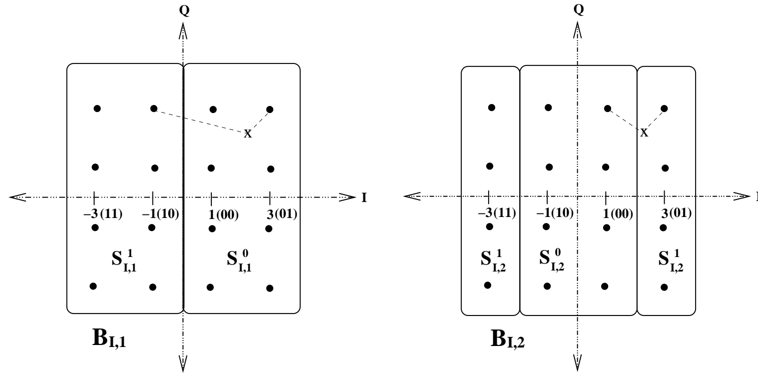


Figure 2.6: Metric partitions of the 16-QAM constellation (from [9]).

The same observation holds for QPSK and 64-QAM constellations.

For QPSK, $D_I = -y_I[i]$. For 64-QAM,

$$D_{I,1} = \left\{ \begin{array}{ll} -y_I[i], & |y_I[i]| \le 2 \\ -2(y_I[i]-1), & 2 < y_I[i] \le 4 \\ -3(y_I[i]-2), & 4 < y_I[i] \le 6 \\ -4(y_I[i]-3), & y_I[i] > 6 \\ -2(y_I[i]+1), & -4 \le y_I[i] < -2 \\ -3(y_I[i]+2), & -6 \le y_I[i] < -4 \\ -4(y_I[i]+3), & y_I[i] < -6 \end{array} \right\} \cong -y_I[i], \tag{2.13}$$

$$D_{I,2} = \left\{ \begin{array}{ll} 2(|y_I[i]|-3), & |y_I[i]| \le 2 \\ -4+|y_I[i]|, & 2 < |y_I[i]| \le 6 \\ 2(|y_I[i]|-5), & |y_I[i]| > 6 \end{array} \right\} \cong -4+|y_I[i]|, \tag{2.14}$$

$$D_{I,3} = \left\{ \begin{array}{ll} -|y_I[i]|+2, & |y_I[i]| \le 4 \\ |y_I[i]|-6, & |y_I[i]| > 4 \end{array} \right\} = ||y_I[i]|-4|-2. \tag{2.15}$$

2.2.2 De-Interleaver

The de-interleaver, as the interleaver, 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 the rule

$$m = s \cdot floor(\frac{j}{s}) + (j + floor(\frac{d \cdot j}{N_{cbps}}))_{mod(s)}, \quad j = 0, 1, \cdots, N_{cbps} - 1, \tag{2.16}$$

and the second permutation is defined by the rule

$$k = d \cdot m - (N_{cbps} - 1) \cdot floor(\frac{d \cdot m}{N_{cbps}}), \quad m = 0, 1, \cdots, N_{cbps} - 1. \tag{2.17}$$

Note that the quantity sent to the decoder are the bit metrics from the demodulator.

2.2.3 Tail-Biting Convolutional Decoding

We first extend the received sequence by repeating the first $(\alpha+\beta)(n/k)$ received bits, where α and β are two important parameters that we have to set. In the Viterbi decoder, the trellis is initialized by making all states equally likely, and the Viterbi algorithm is executed for the extended received sequence. A traceback is performed from the best state at the end of the extended received sequence, and a portion of the data in the decoded block, from position α on for the length of information bits, is chosen as the estimate of the data block.

This scheme relies on the fact that if the received sequence is circularly repeated, the trellis of the extended received sequence can be considered circular since tail-biting code starts and ends in the same state. The trellis of the tail-biting convolutional decoder is depicted in Fig. 2.7. Because the starting state is unknown, the first α surviving paths of the decoder may not be the correct paths. Only after enough depth can the surviving paths approach the correct ones. Thus the later part of the decoded block will be more likely to be the correct information data.

Another issue that should be considered is the traceback mechanism. The surviving path will be almost unique after some depth into the trellis. Therefore, the trellis can be truncated and the traceback mechanism performed after some delay, say τ. A smaller τ entails shorter

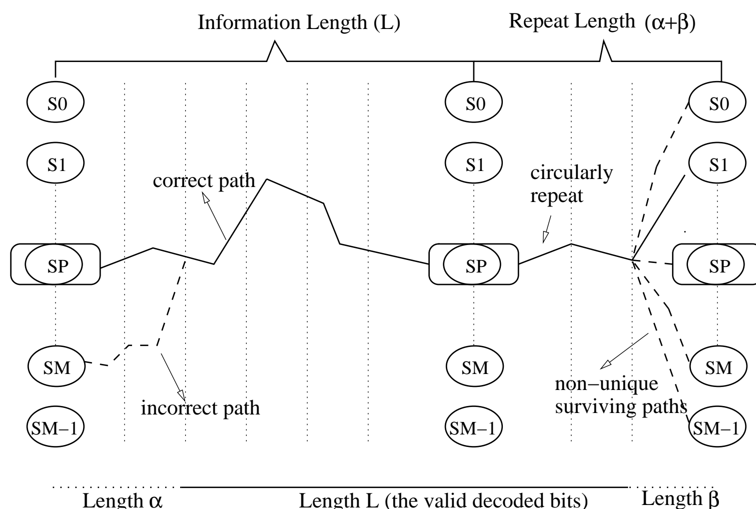


Figure 2.7: Trellis for tail-biting convolutional decoding (from [2]).

decoding delay and smaller amount of memory requirement. To avoid multiple tracebacks our Viterbi decoder does traceback only at the end of the extended received sequence, and the performance will be a little better than the one with truncation since the decision depth is much longer than τ for the earlier bit. For the value of τ , a conventional value is 5 times the register length [10].

Since the ending state of the trellis for the extended received data is unknown and the decision depths for the latest decoded data are not long enough to make the surviving paths unique, the latest decoded data will not be reliable and can not be as used the decoded data. The unreliable data length is set to β, which should be related (actually equal) to τ. We have used simulation results to decide the values of α and β.

2.3 LDPC Code Specifications

The low–density parity check (LDPC) coding scheme used in IEEE 802.16e OFDMA is shown in Fig. 2.8. The randomized input data are first encoded by the LDPC encoder.

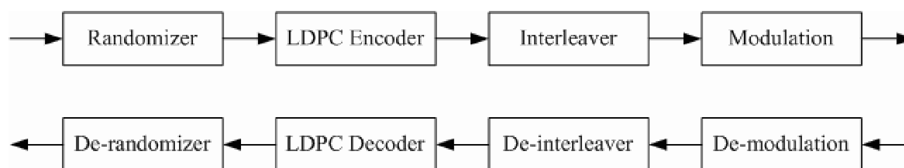


Figure 2.8: LDPC coding structure in transmitter (top path) and decoding in receiver (bottom path).

The encoder and then interleaved by the bit interleaver. Likewise, there are three different modulation types.

LDPC codes are a special case of error correcting codes that have recently been receiving received much attention because of their very high throughput and very good decoding performance. Inherent parallelism in 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 strong error correction ability are important.

Gallager [11] proposed LDPC codes in the early 1960s, but his work received little attention until after the invention of turbo codes in 1993, which used the same concept of iterative decoding. In 1996, MacKay and Neal [12], [13] re-discovered LDPC codes. Chung *et al.* [14] showed that a rate-1/2 LDPC code with block length of 10^7 in binary input AWGN can achieve a threshold of just 0.0045 dB away from the Shannon limit.

LDPC codes have several advantages over turbo codes. First, the sum-product decoding algorithm for these codes has inherent parallelism that can be exploited 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.

Our interest is in both low algorithm complexity and high decoding speed, as these are both desirable under the IEEE 802.16e applications.

Complexity in iterative decoding can be divided into two types: first, complexity of the computations in each iteration and second, the number iterations. Naturally, there is a trade-off between the decoding performance and the complexity and decoding speed.

In this section, we will only discuss the LDPC encoder and decoder block. Other blocks in Fig. 2.8 are the same as in previous section.

2.3.1 Overview of LDPC Code

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, i.e., $cH^T = 0$.

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 a *regular* code and W_c and W_r are tenoned the column degree and the row degree, respectively. The 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. The IEEE 802.16e standard uses irregular codes. Moreover, the codes in 802.16e are systematic, which means that $n - k$ redundant bits are added to k bits of message to form an n bits codeword.

LDPC codes can be represented effectively by a bipartite graph called a Tanner graph [15], [16]. A bi-partite graph is a graph (nodes or vertices are connected by undirected edges)

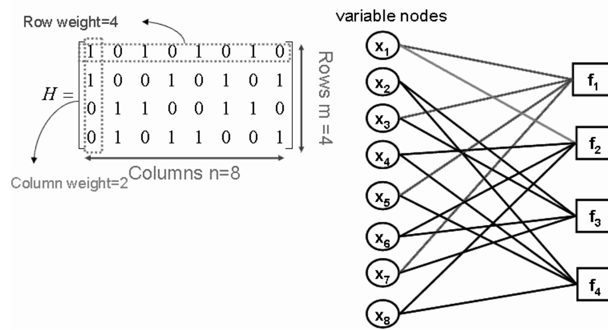


Figure 2.9: Tanner graph of a parity check matrix

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 (or variable 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 2.9 shows a Tanner graph 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*. Degree of a node is the number of branches that is connected to that node.

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

$$\lambda(x) = \sum_{i=2}^{d_{v_{max}}} \lambda_i x^{i-1} \tag{2.18}$$

as the variable node degree distribution, and

$$\rho(x) = \sum_{i=2}^{d_{c_{max}}} \rho_i x^{i-1} \tag{2.19}$$

as the check node degree distribution.

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. 2.9 has a cycle of length four which has been shown in dashed lines. 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.

2.3.2 LDPC Code in IEEE 802.16e OFDMA [1]

The LDPC codes in IEEE 802.16e are systematic linear block codes. They 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 as depicted in Fig. 2.10, two different ones for two rate 2/3 codes, type A in Fig. 2.11 and type B in Fig. 2.12, two different ones for two rate 3/4 codes, type A in Fig. 2.13 and type B in Fig. 2.14, and one for the rate 5/6 code as depicted in Fig. 2.15. 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 is $n_{min} = 24 \times 24 = 576$ bits and the maximum is $n_{max} = 24 \times 96 = 2304$ 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 ith row, jth column in the base matrices, by scaling $p(i, j)$ proportionally as

$$p(f,i,j) = \begin{cases} p(i,j), & p(i,j) \leq 0, \\ \lfloor \frac{p(i,j)z_f}{z_o} \rfloor, & p(i,j) > 0. \end{cases} \tag{2.20}$$

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

$$p(f,i,j) = \begin{cases} p(i,j), & p(i,j) \leq 0, \\ mod(p(i,j), z_f), & p(i,j) > 0. \end{cases} \tag{2.21}$$

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

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 2.11: Base model of the rate-2/3, type A code (from [1]).

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.

2.4 Decoding of LDPC code

2.4.1 The Belief Propagation Decoding Algorithm [17]

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

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 2.12: Base model of the rate-2/3, type B code (from [1]).

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

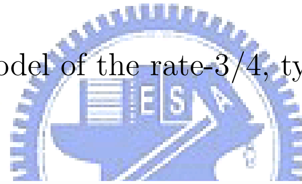


Figure 2.13: Base model of the rate-3/4, type A code (from [1]).

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 2.14: Base model of the rate-3/4, type B code (from [1]).

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 2.15: Base model of the rate-5/6 code (from [1]).

node and the exchange of the results (messages) with neighboring nodes. At each 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, let $q_{n\to m}(0)$ 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, let $r_{m\to n}(0)$ 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 bits 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, let $L_n^{(0)}$ denote $\log(P(x_n = 0|y_n)/P(x_n = 1|y_n))$ at iteration 0, $L_{mn}^{(i)}$ denote $\log\left(r_{m\to n}(0)/r_{m\to n}(1)\right)$ at iteration i and $Z_{mn}^{(i)}$ denotes $\log\left(q_{n\to m}(0)/q_{n\to m}(1)\right)$ at iteration i.

The belief propagation (BP) algorithm is summarized below. 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

$$L_{mn}^{(i)} = 2\tanh^{-1}\left\{\prod_{n'\in N(m)\backslash n} \tanh\frac{Z_{mn'}^{(i-1)}}{2}\right\}. \tag{2.22}$$

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

$$Z_{mn}^{(i)} = L_n^{(0)} + \sum_{m'\in M(n)\backslash m} L_{m'n}^{(i)}. \tag{2.23}$$

Step 3 (decision):

$$Z_n^{(i)} = L_n^{(0)} + \sum_{m\in M(n)} L_{mn}^{(i)}. \tag{2.24}$$

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 $H \cdot \mathbf{X}$=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.

2.4.2 Some Reduced-Complexity LDPC Decoding Algorithms

We focus on methods that simplify the check node updates to obtain reduced-complexity BP algorithms but also achieve good enough performance.

Min-Sum or BP-Based Algorithm [17]

Implementing the calculation in (2.22) in a hardware circuit is difficult and complex. It is also relatively complicated to implement in DSP software. But we can simplify it only approximating it as

$$\begin{aligned} L_{mn}^{(i)} &= 2\tanh^{-1}\left\{ \prod_{n' \in N(m)\backslash n} \tanh \frac{Z_{mn'}^{(i-1)}}{2} \right\} \\ &= \prod_{n' \in N(m)\backslash n} sgn(Z_{mn'}^{(i-1)}) f\left(\sum_{n' \in N(m)\backslash n} f\left(|Z_{mn'}^{(i-1)}| \right) \right) \\ &\approx \prod_{n' \in N(m)\backslash n} sgn(Z_{mn'}^{(i-1)}) f\left(f\left(\min_{n' \in N(m)\backslash n} |Z_{mn'}^{(i-1)}| \right) \right) \\ &= \prod_{n' \in N(m)\backslash n} sgn(Z_{mn'}^{(i-1)}) \min_{n' \in N(m)\backslash n} |Z_{mn'}^{(i-1)}|, \end{aligned} \tag{2.25}$$

where $f(x) = \log\frac{e^x+1}{e^x-1} = -\log(\tanh\frac{x}{2})$ is a fast decaying function as shown in Fig. 2.16. Therefore the second row in (2.25) can be approximated by the third row. Because the f function is it own inverse, we can simplify the third row to the fourth row.

This is a famous approximation called the min-sum or BP-based algorithm which only uses the signum and the minimum functions for check nodes processing. The processing at

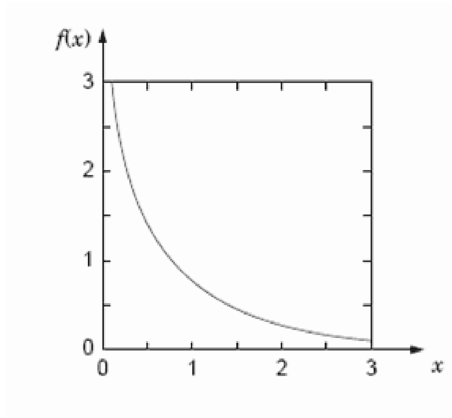


Figure 2.16: Fast decaying function $f(x) = \log\frac{e^x+1}{e^x-1}$.

the bit nodes is identical to that of BP decoding. But coming with the approximation at the check nodes is some performance degradation. We will see the effect later in the simulation results.

Balanced Belief Propagation Algorithm [18]

Observe that the conventional BP algorithm has unbalanced computation complexity between the check nodes operation (2.22) and the bit nodes operation (2.23). A modified version based on algorithmic transformation has been proposed in order to balance the computational load between the two decoding phases. The modified algorithm can be expressed as

$$L_{mn}^{(i)} = \prod_{n' \in N(m)\backslash n} sgn(Z_{mn'}^{(i-1)}) \sum_{n' \in N(m)\backslash n} f\left(|Z_{mn'}^{(i-1)}|\right), \tag{2.26}$$

$$Z_{mn}^{(i)} = L_n^{(0)} + \sum_{m' \in M(n)\backslash m} sgn(L_{m'n}^{(i)}) f\left(L_{m'n}^{(i)}\right). \tag{2.27}$$

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

two decoding phases.

Normalized BP-Based Algorithm

Let L_1 and L_2 represent the values of $L_{mn}^{(i)}$ computed by the BP algorithm and the BP-based algorithm with (2.22) and (2.25), respectively. It can be shown that L_1 and L_2 have the same sign, i.e., $\text{sgn}(L1) = \text{sgn}(L2)$ and L_2 has larger magnitude than L_1, i.e., $|L_2| > |L_1|$ [19]. According to [19], we can further modify (2.25) to let the BP-based algorithm obtain a BER vs. $\frac{Eb}{No}$ performance curve closer to the conventional BP algorithm.

Because $\text{sgn}(L1) = \text{sgn}(L2)$, 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,

$$\widehat{L_{mn}^{(i)}} \longleftarrow \frac{L_{mn}^{(i)}}{\alpha}, \tag{2.28}$$

where $\widehat{L_{mn}^{(i)}}$ is the output of the check node operation for normalized BP-based algorithm. The bit node operation stays unchanged. Ideally, α should vary with the signal-to-noise ratio (SNR) and with iterations to achieve the optimum performance. But it is kept constant for the sake of simplicity.

Offset BP-Based Algorithm

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

$$\widehat{L_{mn}^{(i)}} \longleftarrow sgn(L_{mn}^{(i)}) \max(|L_{mn}^{(i)}| - \beta, 0) \tag{2.29}$$

where $\widehat{L_{mn}^{(i)}}$ is the output from the check node operation for the offset BP-based algorithm. Again, the bit node operation stays the same. Also, β should vary with the signal-to-noise ratio (SNR) and with iterations to achieve the optimum performance. But it is kept constant for the sake of simplicity.

Table 2.5: Comparison of Main Operations of Different Decoding Algorithms

Decoding Algorithm	Main Operation
BP Decoding	$tanh$ and $tanh^{-1}$
Min-Sum Decoding	Minimum
Normalized BP-Based Decoding	Minimum and Division (or Multiplication)
Offset BP-Based Decoding	Minimum, Maximum and Substraction

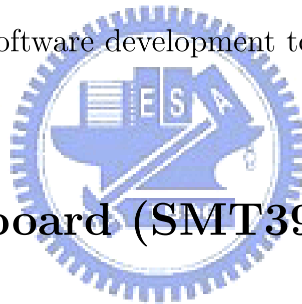
In summary, the BP decoding needs $tanh^{-1}$ and $tanh$ operations, the min-sum algorithm needs the minimum operation, the normalized BP-based algorithm needs minimum and division operations, and the offset BP-based algorithm needs minimum, maximum and substraction operations. A comparison of the different algorithms is given in Table 2.5.

Obviously, BP decoding is the most complex operation, and min-sum is the least. The two improved decoding methods are in between.

Chapter 3

DSP Implementation Environment

The DSP baseboard (SMT395) we used is Texas Instruments' TMS3200C6416T DSP chip and Xilinx Virtex-II Pro FPGA. In this chapter, our discussion will concentrate on the DSP system development environment, DSP chip and its features because our implementation is software-based on the DSP. The software development tool, Code Composer Studio (CCS), is also introduced.



3.1 The DSP Baseboard (SMT395)

The DSP card used in our implementation is Sundance's SMT395 shown in Fig. 3.1. It houses a 1 GHz 64-bit TMS320C6416T DSP of TI. The SMT395 is supported by the TI's Code Composer Studio and the 3L Diamond to enable multi-DSP systems with minimum efforts by the programmers.

Features of SMT395 board include:

- 1GHz TMS320C6416T fixed-point DSP processor with L1, L2 cache and SDRAM.
- 8000MIPS peak DSP performance.
- Xilinx Virtex II Pro FPGA XC2VP30-6 in FF896 package.

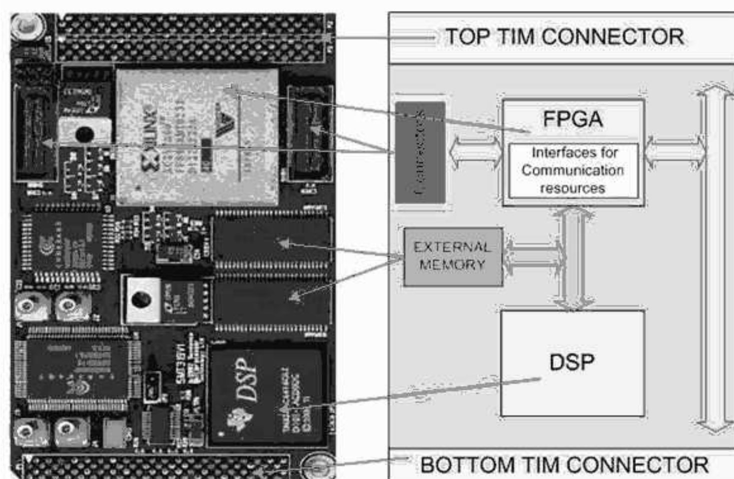


Figure 3.1: SMT395 Module.

- 256 Mbytes of SDRAM at 133MHz
- Eight 2Gbit/sec Rocket Serial Links (RSL) for inter module.
- Two Sundance High-speed Bus (50MHz, 100Mhz or 200MHz) ports at 32 bits width.
- 8 Mbytes flash ROM for configuration and booting.

3.2 The DSP Chip

The following text is mainly taken from references [21] and [22].

The TMS320C64x DSP is a fixed-point DSP in the TMS320C64x series of the TMS320C6000 DSP platform family. The TMS320C64x device is 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.
- 8/16/32/64-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.

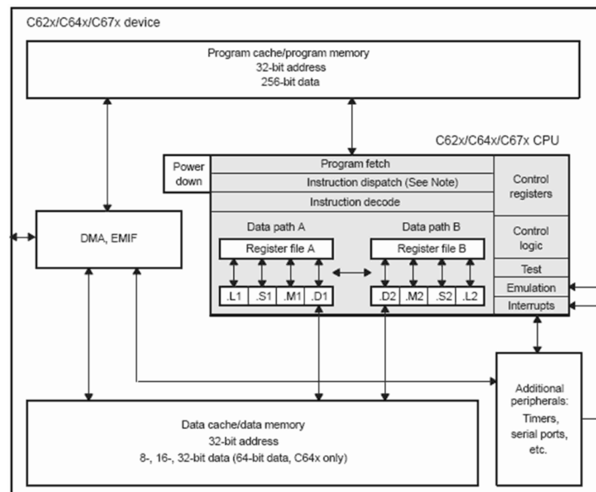


Figure 3.2: Block diagram of TMS320C6416 DSP (from [23]).

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

- 32x32-bit integer multiply with 32- or 64-bit result.

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.

The block diagram of the C6000 family is show in Fig. 3.2. The C6000 devices come with program memory, which, on some devices, can be used as a program cashe. The devices also have varying sizes of data memory. Peripherals such as a direct memory access (DMA) controller, power-down logic, and external memory interface (EMIF) usually come with the CPU, while peripherals such as serial ports and host ports are available only for certain model.

In the following subsections, the TMS320C64x DSP Chip is introduced in the two part: Central processing unit (CPU), Memory.

3.2.1 Central Processing Unit [23]

Besides the eight independent functional units and sixty-four general purpose 32-bit registers that has been mentioned before, the C64x CPU also consists of the program fetch unit, instruction dispatch unit (attached with advanced instruction packing), instruction decode unit, two data path (A and B, each with four functional units), test unit, emulation unit, interrupt logic, several control registers and two register files (A and B with respect to the two data paths).

The architecture is illustrated in more detail in Fig. 3.3. Compared with the other C6000 family DSP chip, the C64X DSP chip provides more available hardware resources.

The block diagram of C6416 DSP is shown in Fig. 3.2. 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

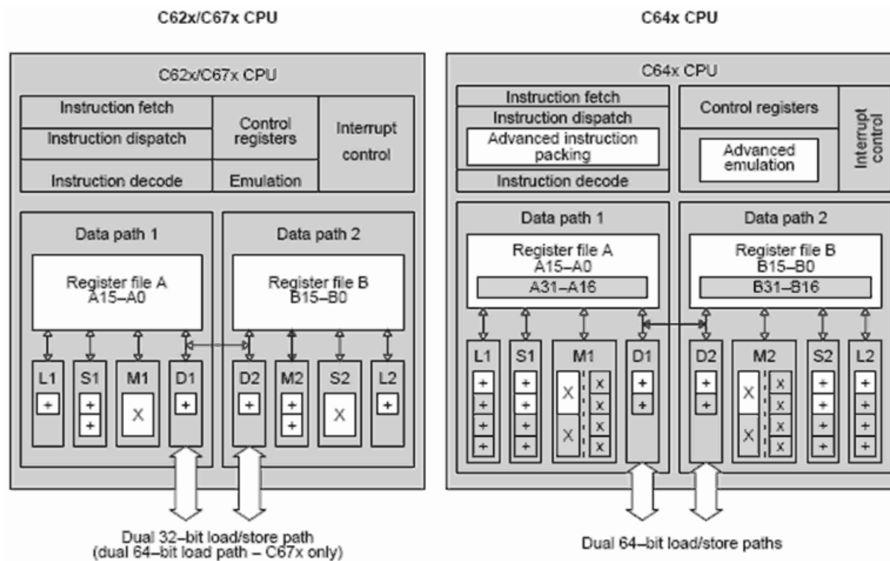


Figure 3.3: The TMS320C64x DSP chip architecture and comparison with earlier TMS320C62x/C67x chip (from [23]).

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.

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. 3.4.

Reference [23] 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 Fig. 3.5, which gives a mapping of operations occurring in each execution phase for the different instruction types. The delay slots associated with

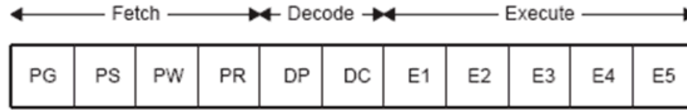


Figure 3.4: Pipeline phases of TMS320C6416 DSP (from [23]).

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 program fetch unit shown in the Fig. 3.3 could fetch eight 32-bit instructions (which implies 256-bit wide program data bus) every single cycle, and the instruction dispatch and decode units could also decode and arrange the eight instructions to eight functional units. The eight functional units in the C64x architecture could be further divided into two data paths A and B as shown in Fig. 3.3. Each path has one unit for multiplication operations (.M), one for logical and arithmetic operations (.L), one for branch, bit manipulation, and arithmetic operations (.S), and one for loading/storing, address calculation and arithmetic operations (.D). The .S and .L units are for arithmetic, logical, and branch instructions. All data transfers make use of the .D units. Two cross-paths (1x and 2x) allow functional units from one data path to access a 32-bit operand from the register file on the opposite side. There can be a maximum of two cross-path source reads per cycle. There are 32

		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‡

Figure 3.5: Execution stage length description for each instruction type (from [23]).

general purpose registers, but some of them are reserved for specific addressing or are used for conditional instructions.

The eight functional units in the C6000 data paths can be divided into two groups of 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 3.1.

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. 3.6). 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

Table 3.1: Functional Units and Operations Performed (from [23])

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

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.

3.2.2 Memory [24]

Internal Memory

The C64x DSP chip has a 32-bit, byte-addressable address space. Internal (on-chip) memory is organized in separate data and program spaces. When off-chip memory is used, these spaces are unified on most devices to a single memory space via the external memory interface (EMIF). The C64x has two 64-bit internal ports to access internal data memory and a single internal port to access internal program memory, with an instruction-fetch width of 256 bits

Memory Options

the C64x DSP Chip also provides a variety of memory options:

- Large on-chip RAM, up to 7M bits.
- Program cache.
- 2-level caches.
- 32-bit external memory interface supports SDRAM, SBSRAM, SRAM.

And other asynchronous memories for a broad range of external memory requirements and maximum system performance.

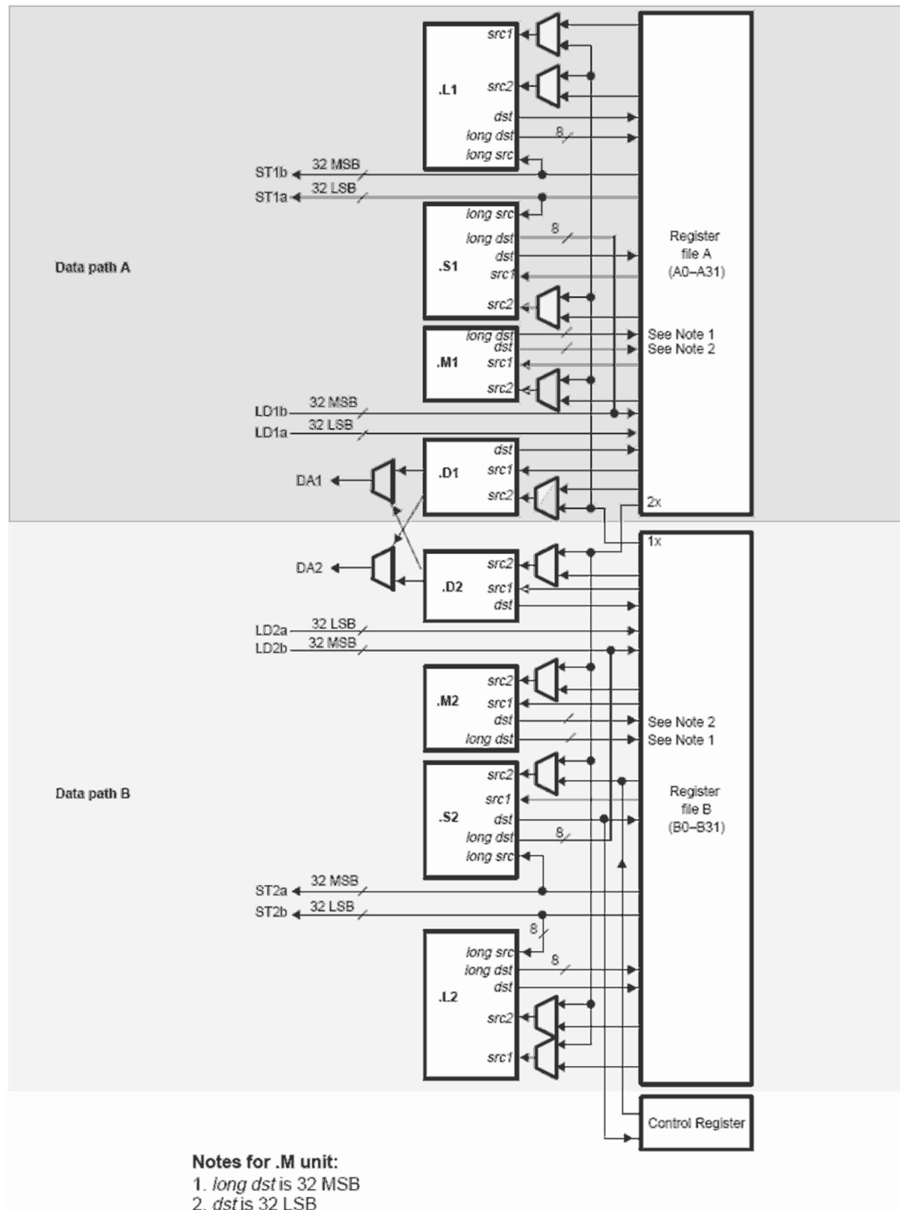


Figure 3.6: TMS320C64x CPU data paths (from [23]).

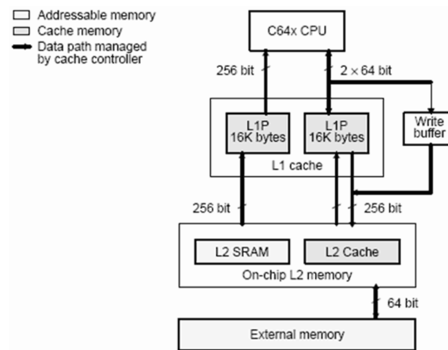


Figure 3.7: C64x cache memory architecture (from [24]).

Cache Memory

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. 3.7. 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 [24].

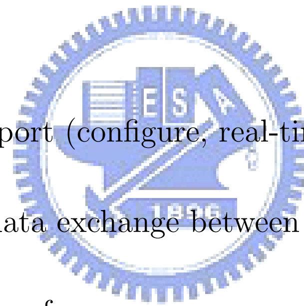
3.3 TI's Code Development Environment [25], [26]

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.
  - 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.

3.4 Code Development Flow [27]

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. 3.8. The tutorial section of the Programmers Guide [27] 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. 3.8.

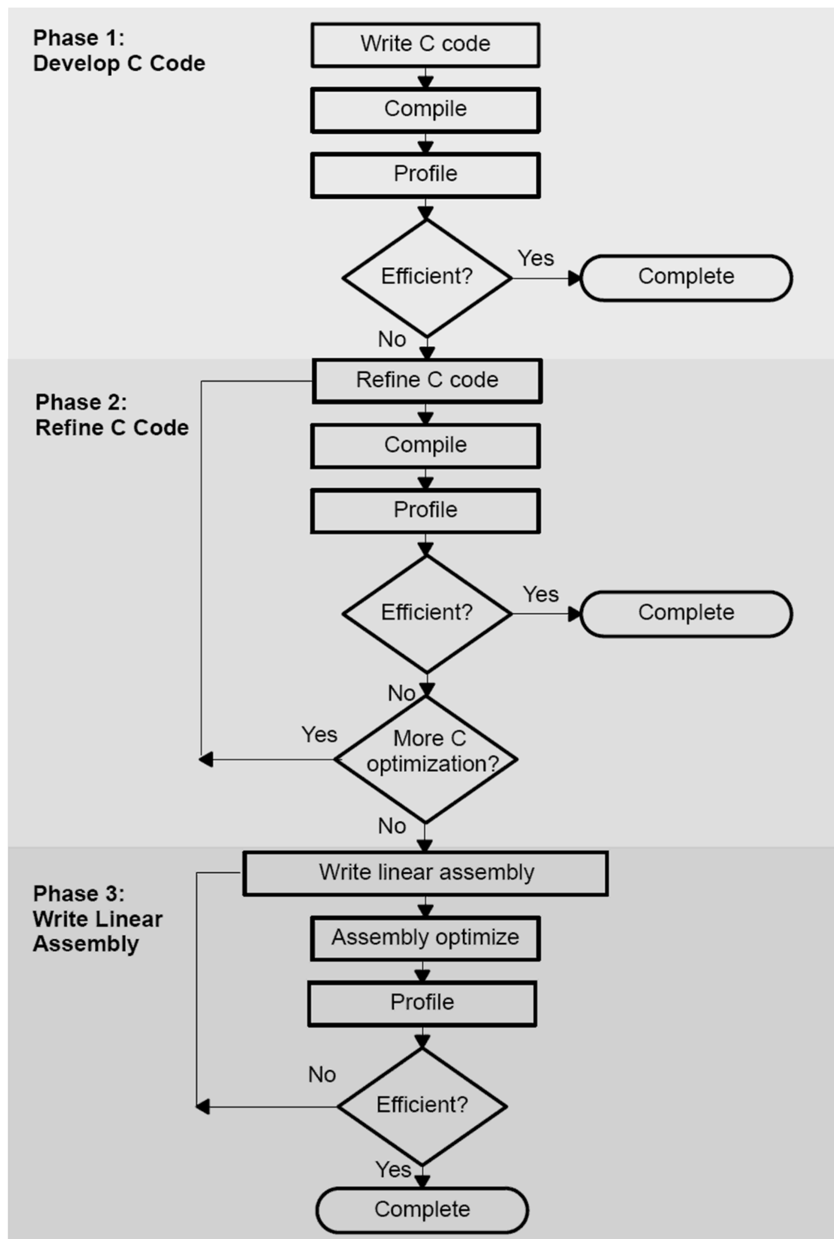
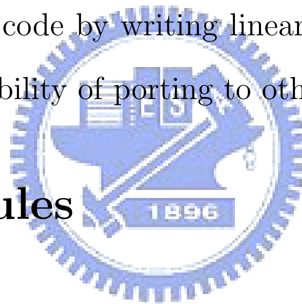


Figure 3.8: Code development flow for TI C6000 DSP (from [27]).

- 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 [27] 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.



3.5 Acceleration Rules

In this section, we describe several methods that can accelerate our code and reduce the execution time on the C64x DSP.

3.5.1 Compiler Optimization Options [27]

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 -on 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.

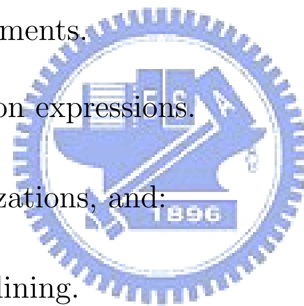


Table 3.2: Sizes of Different Data Types

Data type	Char	Short	Int	Long	Float	Double
Size (bits)	8	16	32	40	32	64

- 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.

3.5.2 Fixed–Point Coding

The C6000 compiler define a size for each data type as Table 3.2. The C64X DSP is a fixed-point processor, so it can only perform fixed-point operations. Although the C64X DSP can simulate floating-point processing, it takes many clock cycles to do the job. The "char", "short", "int" and "long" are fixed-point data types, and the "float" and "double" are floating-point data types.

3.5.3 Loop Unrolling

Loop unrolling unrolls the loops so the all iterations of the loop appear in the code. It often increases the number of instructions available to execute in parallel. It is also suitable for use with software pipelining. When our code has conditional instructions, sometimes the compiler may not be sure that the branch will occur or not. It needs more waiting time for

(a) Before Unrolling

```
int i,a=0,b=0
for (i=0,i<8,i++)
{
a+=i;
b+=i;
}
```

(b) After Unrolling

```
int i=0,a=0,b=0
a+=i;b+=i;i++;
a+=i;b+=i;i++;
a+=i;b+=i;i++;
a+=i;b+=i;i++;
a+=i;b+=i;i++;
a+=i;b+=i;i++;
a+=i;b+=i;i++;
a+=i;b+=i;i++;
```

Figure 3.9: Loop unrolling.

Table 3.3: Comparison Between Unrolled and not Unrolled

	Before Unrolling	After Unrolling
Execution Cycles	436	206
Code Size	116	479

the decision of branch operation. If we do loop unrolling, some of the overhead for branching instruction can be reduced. Fig. 3.9 is an example about loop unrolling and Table 3.3 shows the cycles and the code size with and without unrolling. We can see clearly that the clock cycles decrease after loop unrolling, but the code size has increased.

3.5.4 Packet Data Processing

Packet data processing means processing of several data together in one instruction. For example, we may use a single load or store instruction to access multiple data that are located consecutively in the memory. It can enhance data throughput. The technique is also called the single instruction multiple data (SIMD) method. For example, if we can place four 8-bit data (char) or two 16-bit data (short) in a 32-bit space, we may do four or two operations in one clock cycle. The code efficiency substantially. Some intrinsic functions

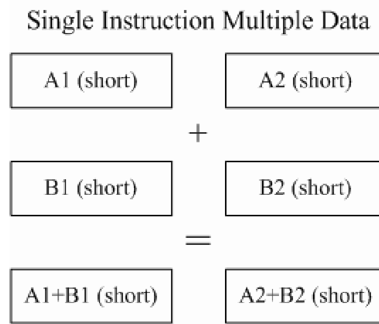


Figure 3.10: The block diagram of SIMD.

enhance the efficiency in a similar way. Fig. 3.10 shows an example that uses word access for adding short data.

3.5.5 Register and Memory Arrangement

When accessing in the external memory, it may tale more clock cycles than accessing on–chipdata. We can use registers to store data in order to reduce the transfer time. The C compiler has a pre–defined way of placing different code segments (such as variable pointers, malloc spaces, and the program code) in the memory. We can set up the link commend (.cmd) file to allocate the memory for different types of data for efficient data reading and writing. The key–words "CODE SECTION" and "DATA SECTION" can be used to put the program code or data in the internal memory for greater execution speed.

3.5.6 Software Pipelining

Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel. The compiler always attempts to software pipeline. In Fig. 3.11, illustrates a software pipelined loop. The stages of the loop are represented by A, B, C, D and E. In this figure, a maximum of five iterations of the loop can execute at one time. The shaded area represents the loop kernel. In the loop kernel, all five stages execute

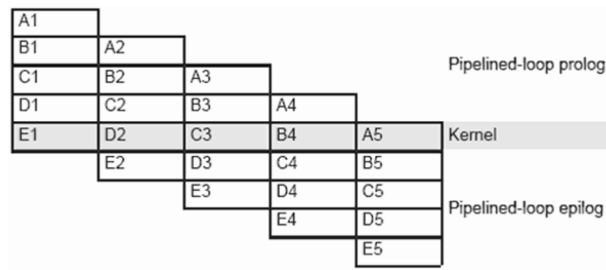


Figure 3.11: Software-pipelined loop.

in parallel. The area above the kernel is known as the pipelined loop prolog, and the area below the kernel is known as the pipelined loop epilog.

3.5.7 Macros and Intrinsic Functions

Because software-pipeline cannot contain function calls, it takes more clock cycles to complete function calls. Changing functions to macros under some conditions is a good way for code optimization. In addition, replacing functions with macros can cut down the code for initial function definition and reduce the number of branches. But macros are expanded each time they are called. Hence, they will increase the code size.

The TI C6000 compiler provides many special functions that map C codes directly to inlined C64x instructions, which increase C code efficiency. These special functions are called intrinsic functions. If some instructions have equivalent intrinsic functions, we can replace them by intrinsic functions and the execution time can be decreased.

3.5.8 Other Acceleration Rules

Other code Acceleration rules include reducing memory access, using bit shifts for multiplication or division, declaring constants as constants that not variable, access the memory sequentially, and minimizing use of conditional breaks or complex condition codes in loops.

Chapter 4

Simulation and DSP Implementation of Convolutional Encoder and Decoder

In this chapter, we present some floating-point simulation results for the convolution encoder and decoder. The simulation results provide information concerning proper choices of certain design parameters, such as α and β in tail-biting convolutional code decoder. We then present fixed-point simulation result and compare them with the floating-point results.

Then, we discuss the decoding algorithm of the IEEE802.16e OFDMA convolution codec on DSP. We base our implementation on modification of the code of Lee [29] 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 Coding Gain Analysis

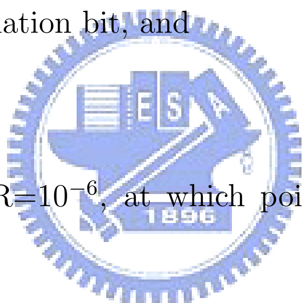
In this section, we analyse the convolutional coding gains to obtain to the reference to compare simulation results with. Coding gains are usually analyzed for AWGN channel. In AWGN channel, let the transmitted symbol energy $E_s = 1$. Then the relationship between

E_b/N_0 and the noise variance σ^2 is given by

$$\begin{aligned} \sigma^2 &= (\frac{E_s}{N_0})^{-1} \\ &= (\frac{N_b \cdot E_c}{N_0})^{-1} \\ &= (\frac{N_b \cdot R_c \cdot E_b}{N_0})^{-1} \end{aligned} \tag{4.1}$$

where

- E_s/N_0 is sometimes called SNR,
- N_b gives number of bits per symbol, which for QPSK, 16QAM, and 64QAM is 2, 4, and 6, respectively,
- $E_c = \frac{E_s}{N_b}$ is energy per code bit,
- $E_b = \frac{E_c}{R_c}$ is energy per information bit, and
- R_c is the code rate.



Crucial reference point is BER=10^{-6}, at which point the IEEE 802.16e specifies the performance requirement.

We investigate coding gains through several different views. First, we find the Shannon bounds on coding gain at different code rates specified in IEEE 802.16e. This helps us understand the limit in performance channel coding can provide. Then we estimate the coding gains of the convolutional codes based on minimum codeword distances.

The Shannon-Hartley law for the capacity of an AWGN channel is given by

$$CR_c = \log_2(1 + \frac{E_b C R_c}{N_0}), \tag{4.2}$$

where C is bit rate per Hz on channel and R_c is the code rate. As a result, the lower bound on E_b/N_0 is given by

$$\frac{E_b}{N_0} \geq \frac{2^{CR_c} - 1}{CR_c}. \tag{4.3}$$

Table 4.1: Coding Gain Upper-Bound in AWGN at BER = 10^{-6}

Modulation	Code Rate	Channel Bit Rate Under Minimum Bandwidth Design (C)	Shannon Bound (dB)	E_b/N_0 for Uncoded Transmission with Coherent Demodulation (dB)	Coding Gain Upper-Bound (dB)
QPSK	1/2	2	0	10.5	10.5
QPSK	3/4	2	0.86	10.5	9.64
16QAM	1/2	4	1.76	14.5	12.74
16QAM	3/4	4	3.68	14.5	10.82
64QAM	1/2	6	3.68	19.0	15.32
64QAM	2/3	6	5.74	19.0	13.26
64QAM	3/4	6	6.82	19.0	12.18

The coding gain upper-bound is the difference between the Shannon bound and the E_b/N_0 at BER = 10^{-6} for uncoded transmission with coherent demodulation. We list the coding gain upper-bound of the seven coding-modulation schemes in IEEE 802.16e in Table 4.1.

With BPSK or QPSK modulation, a rough estimate of convolutional coding gain in AWGN is

$$10\log_{10}(R_c \cdot d_{free}) \text{ dB}, \tag{4.4}$$

where R_c is the code rate and d_{free} is the free distance. This coding gain also assumes soft-decision decoding. For hard-decision decoding, the coding gain should be smaller by 2 to 3 dB. We conjecture that, for 16-QAM and 64-QAM with Gray-coded bit mapping, the coding gain will depend on how the coded bits are mapped into the different symbols. With sufficiently random interleaving, the estimate based on (4.4) may still apply. In Table 4.2, we list the coding gain estimates based on (4.4) for the seven convolutional coding schemes in IEEE 802.16e.

Table 4.2: Approximate Coding Gain Based on Analysis of Minimum Codeword Distance

Modulation	CC Code Rate	d_{free}	Soft-Decision CC Coding Gain (dB)
QPSK	1/2	10	6.99
QPSK	3/4	5	5.74
16QAM	1/2	10	6.99
16QAM	3/4	5	5.74
64QAM	1/2	10	6.99
64QAM	2/3	6	6.02
64QAM	3/4	5	5.74

4.2 Performance in AWGN with Floating-Point Processing

In this section, we discuss the floating-point simulation results of convolutional coding performance in AWGN based on the system structure shown in Fig. 2.1.

We discussed the importance of the parameters α and β. In Figs. 4.1 and 4.2 we show the floating-point simulation results of soft-decision decoding performance of the seven codes at different values of E_b/N_0 for different values of α and β between 0 and 96. From Fig. 4.1, we conclude that, for rate–1/2 and QPSK ,16QAM and 64QAM modulations, the performance is almost the same when $\alpha \geq 12$ and $\beta \geq 12$. From Fig. 4.2, we conclude that, for rate–2/3 and rate–3/4 coding and QPSK ,16QAM and 64QAM modulations, the performance is almost the same when $\alpha \geq 24$ and $\beta \geq 24$.

At rate 1/2, the codeword contains more parity check bits than at other rate. Then we can use smaller α and β to reduce the decoder complexity without performance loss. Considering the trade-off between performance and decoder complexity, we decide to let $\alpha = 48$ and $\beta = 48$ in our Viterbi decoder. A value of $\beta = 48$ is equivalent to a decoding

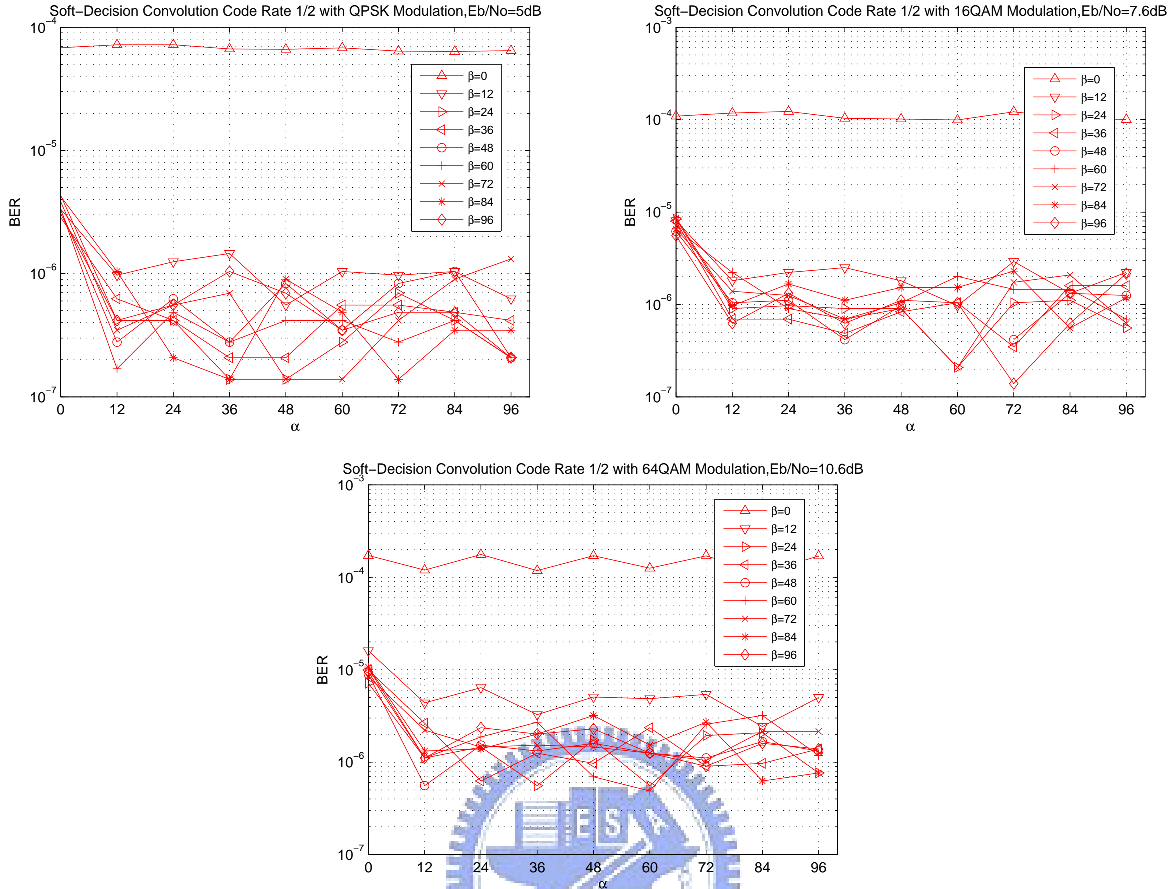


Figure 4.1: Soft-decision decoding performance of rate–1/2 coding in AWGN with different value of α and β employing floating-point computation.

delay of 48 bits. And it happens to be equal to 8 times the register length, a proper value of delay by experience.

To further confirm that we have made a proper choice of α and β, we run simulation under different E_b/N_0 value for $\alpha = \beta = 48$. The simulation results are depicted in Fig. 4.3. Table 4.3. lists the coding gains obtained from simulation with $\alpha = \beta = 48$ and compares them with the theoretic value obtained earlier. See that the coding gains obtained from simulation are only less than the theoretic value by approximately 1 dB or less. Therefore, suboptimal tail-biting Viterbi decoding and BICM de–interleaving provide acceptable performance at

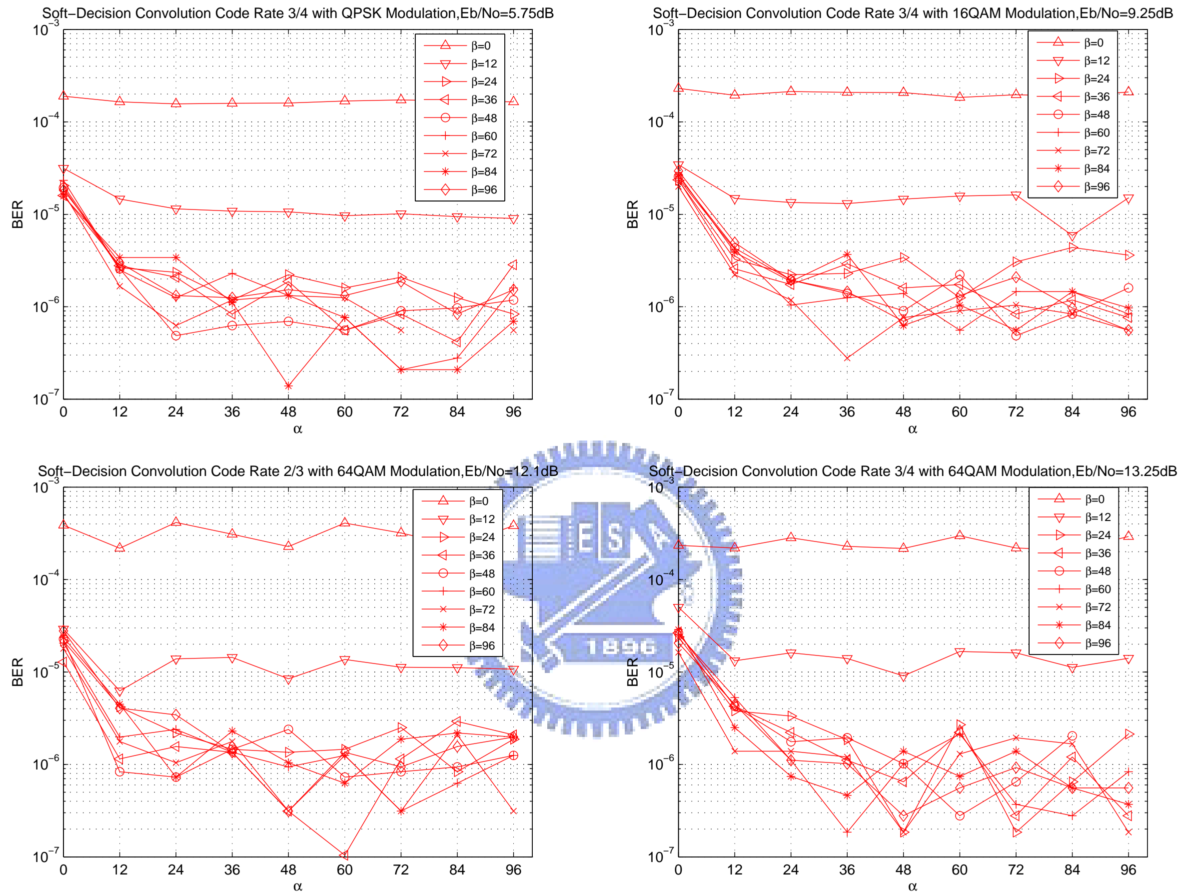


Figure 4.2: oft-decision decoding performance of rate–2/3 and rate–3/4 coding in AWGN with different value of α and β employing floating-point computation.

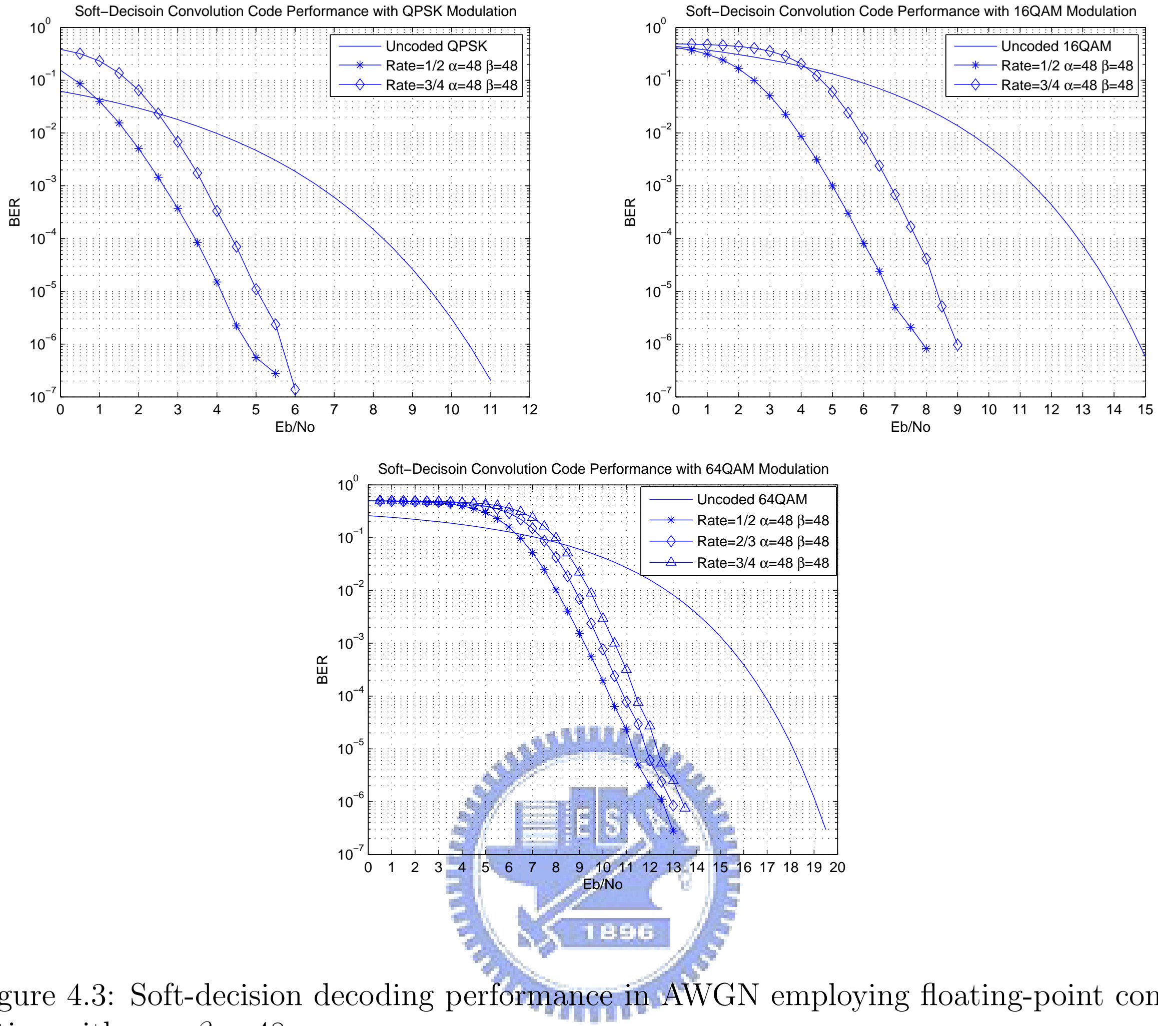


Figure 4.3: Soft-decision decoding performance in AWGN employing floating-point computation with $\alpha = \beta = 48$.

reasonable complexity.

4.3 Performance in AWGN with Fixed-Point Processing

In order to implement the encoder and decoder on DSP, we need to convert the floating-point processing to fixed-point processing. In this section we discuss the simulation results with fixed-point processing.

First, we convert floating-point values to the fixed-point ones. By multiplying the original

Table 4.3: Comparison of Convolutional Coding Gain froms in AWGN at BER = 10^{-6}

Modulation	CC Code Rate	Theoretic Soft-Decision CC Coding Gain (dB)	Soft-Decision CC Coding Gain from Simulation Using floating-point computation with $\alpha = \beta = 48$ (dB)
QPSK	1/2	6.99	5.62
QPSK	3/4	5.74	4.82
16QAM	1/2	6.99	6.28
16QAM	3/4	5.74	5.43
64QAM	1/2	6.99	6.35
64QAM	2/3	6.02	5.97
64QAM	3/4	5.74	5.64

floating-point values by 1000 and rounding the result to integer. Then we use 12 bits to represent this result.

Note that we only change the number of bit in the decoder input. We fix the integer part in 4 bit and change the fraction part bit numbers. But the precision of intermediate results decoding computation is still 16 bits. In Fig. 4.4, we see that, in QPSK we can use 4 bits to express the decoder input and achieve a performance less than 1 dB away from using 12 bits. In 16QAM and 64QAM, 4 bits are not good enough and we need at least 5 bits in 16QAM and 6 bits in 64QAM. In DSP implementation, we can only choose between char(8 bits) and short(16 bits). Thus we choose 16 bits as the decoder input width.

Next, we consider the impact of the parameters α and β in fixed-point processing. From Fig. 4.5, we see that, when $\alpha \geq 24$ and $\beta \geq 24$, the performance is almost the same as for floating-point processing. Therefore, $\alpha = 48$ and $\beta = 48$ is a suitable choice also for fixed-point processing.

Now we run simulation under different E_b/N_0 value for $\alpha = \beta = 48$. The simulation

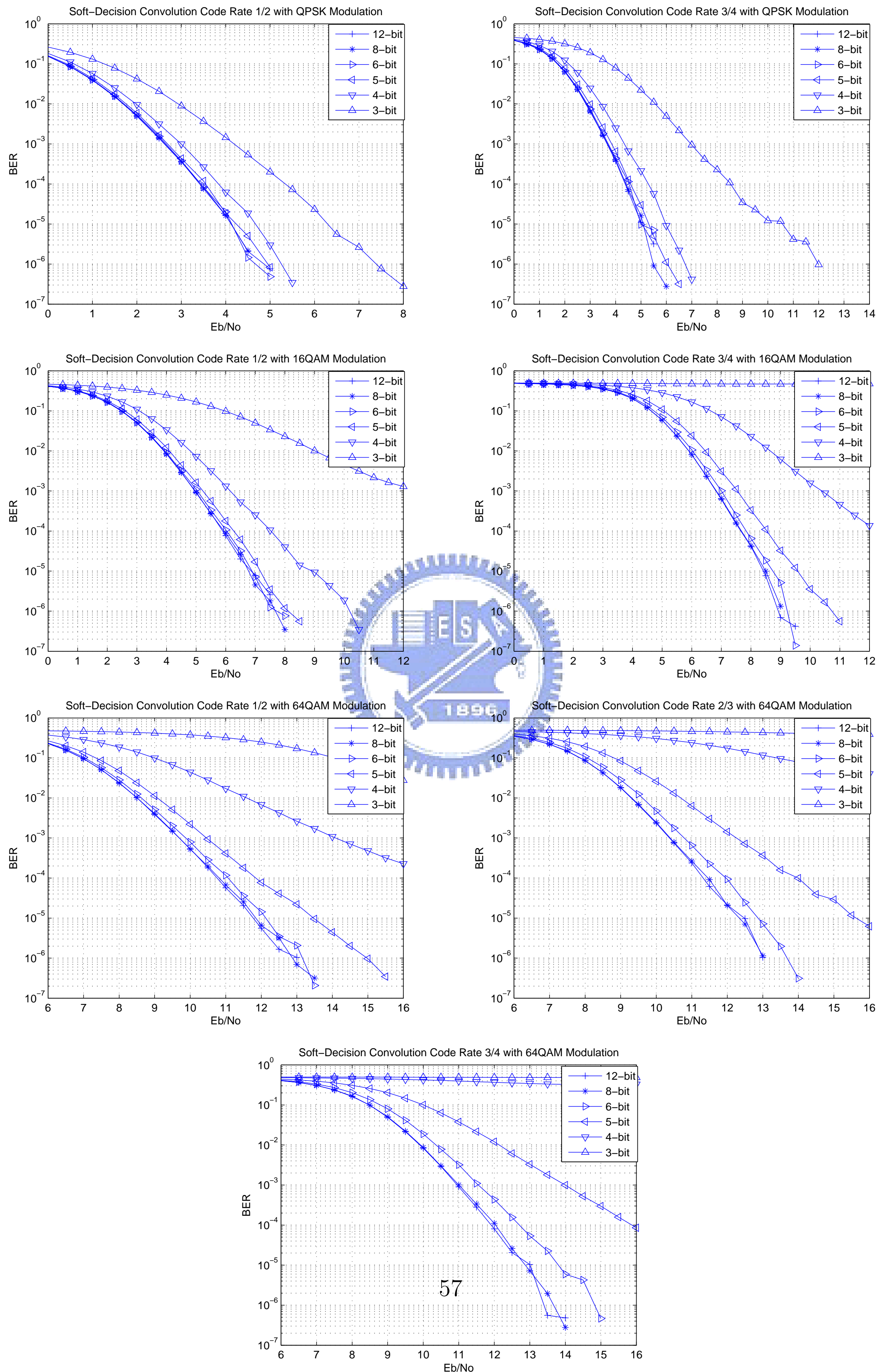


Figure 4.4: Soft-decision decoding performance in AWGN with different input precisions.

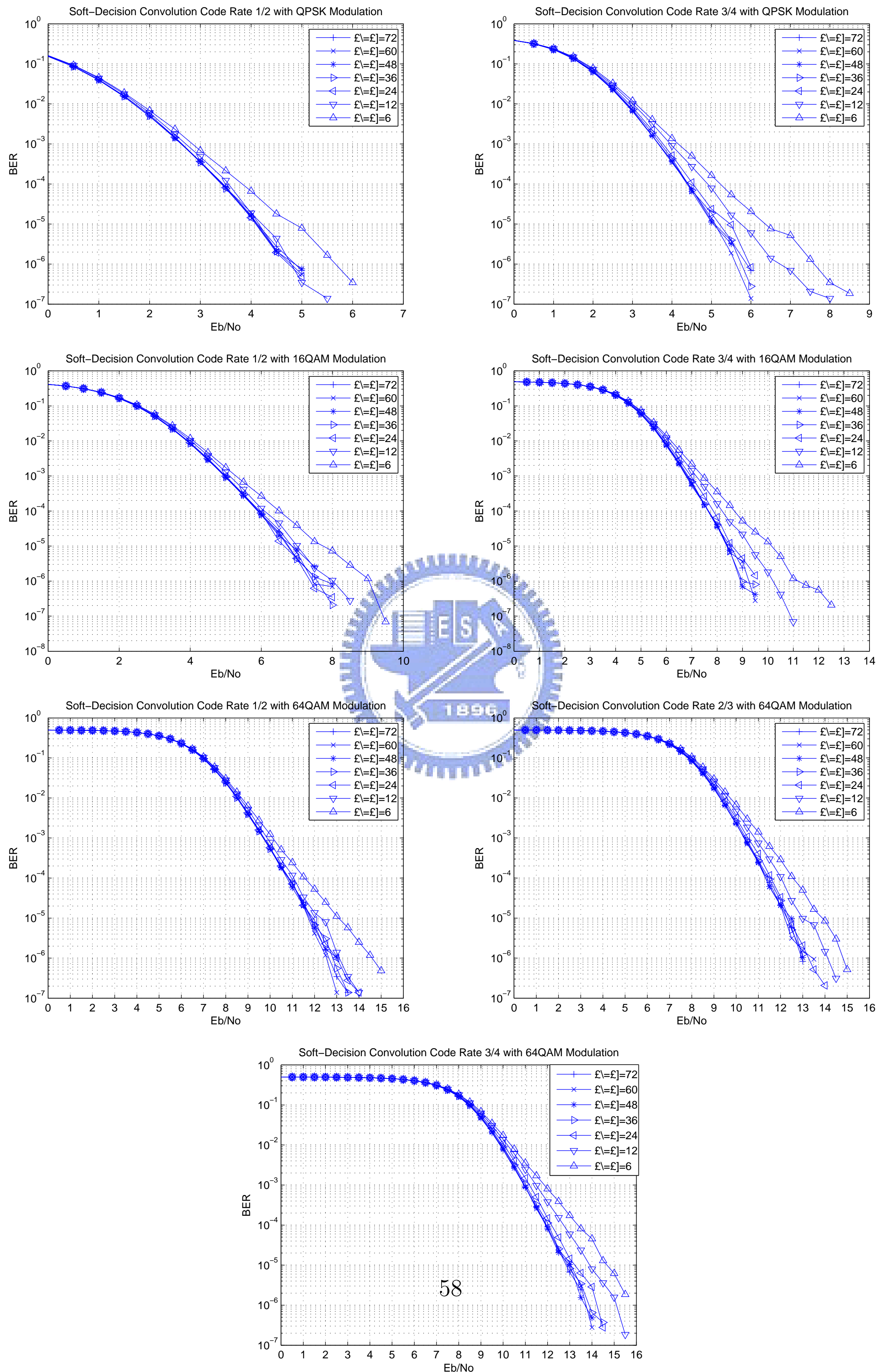


Figure 4.5: Soft-decision decoding performance employing fixed-point computation in AWGN with different value α and β.

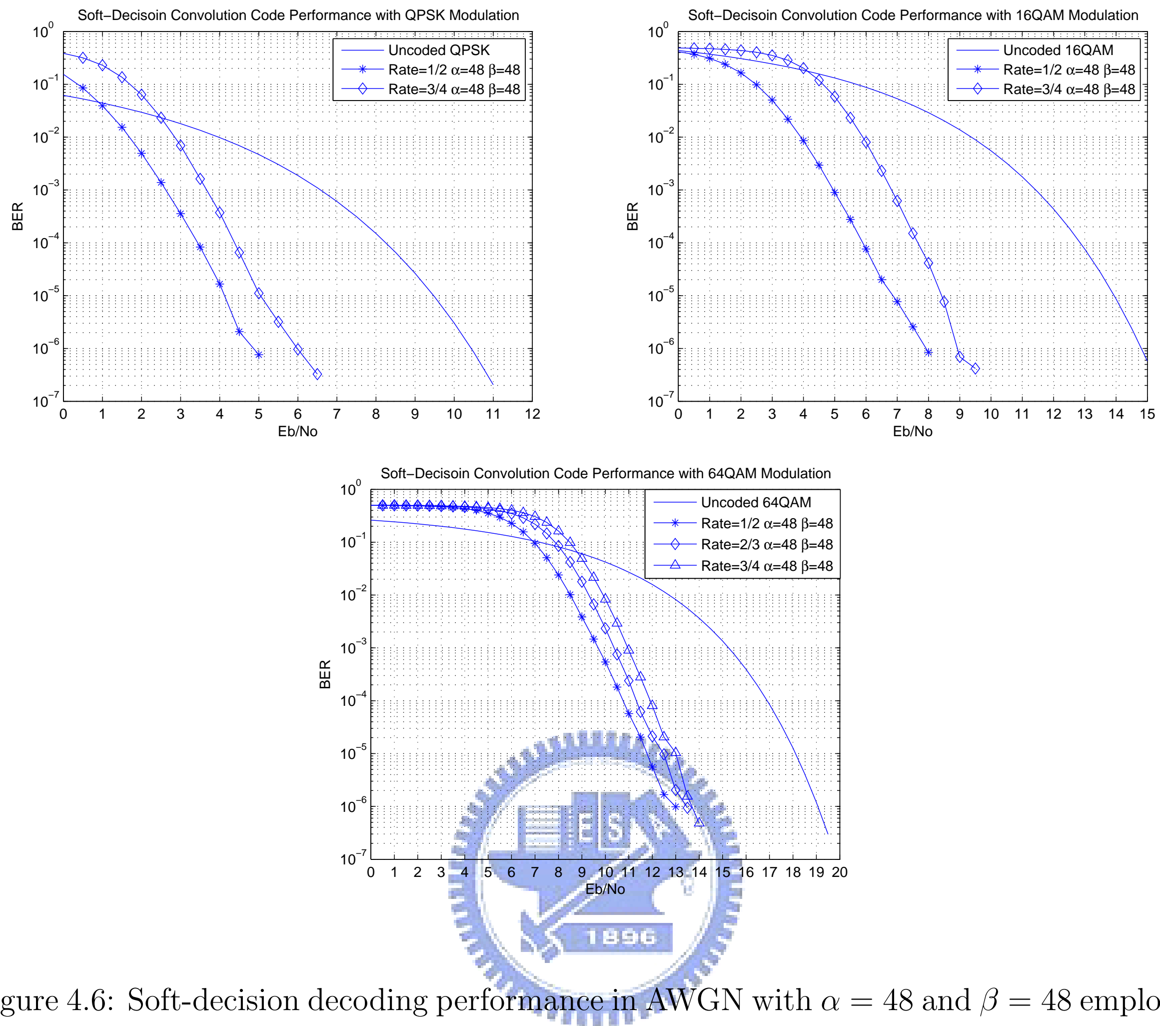


Figure 4.6: Soft-decision decoding performance in AWGN with $\alpha = 48$ and $\beta = 48$ employing fixed-point computation.

results are shown in Fig. 4.6. Table 4.4 compares the coding gain obtained from fixed-point computation with $\alpha = \beta = 48$ with the theoretic coding gains obtained previously.

In Fig. 4.7, we compare the simulation results of floating-point and fixed-point processing with $\alpha = \beta = 48$. The performance of floating-point and fixed-point computation is almost the same for every code rate and every modulation method.

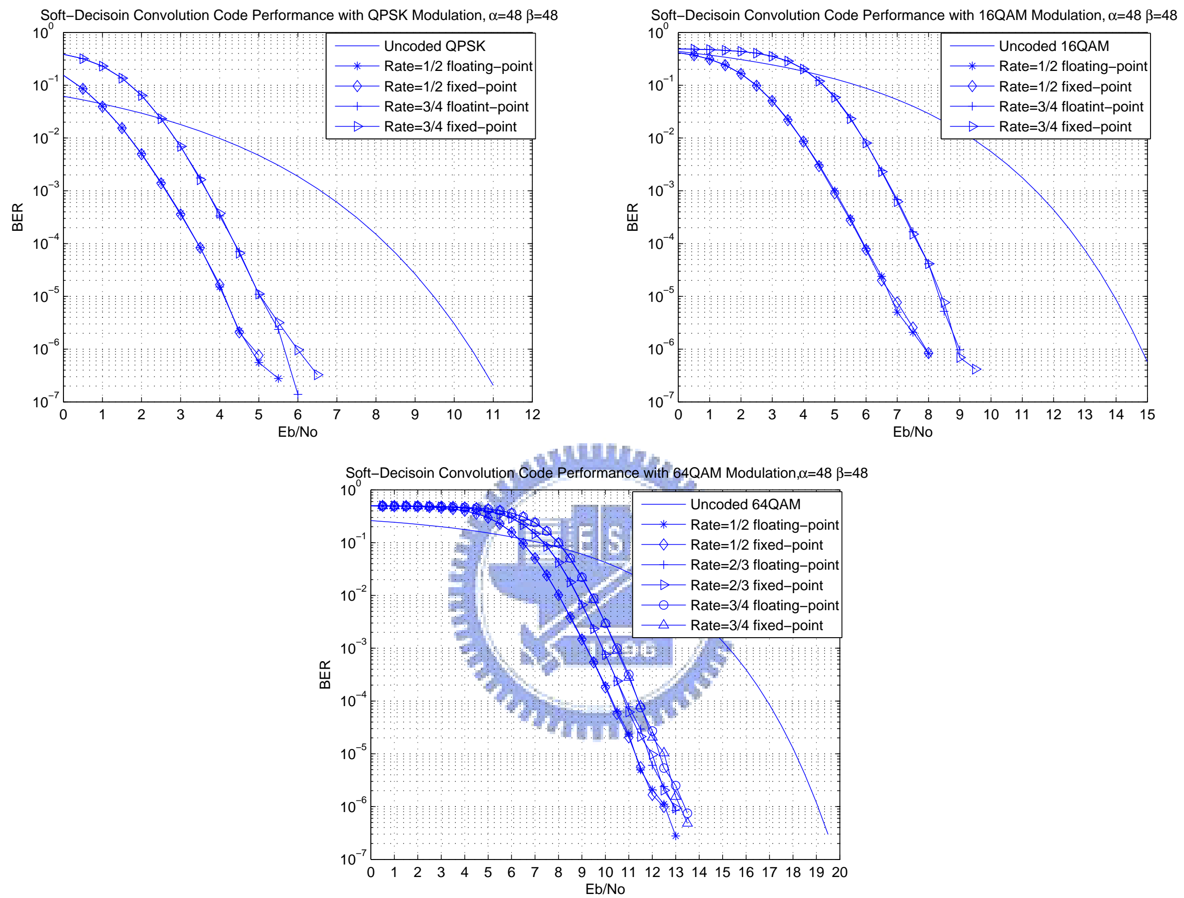


Figure 4.7: Comparison between soft-decision decoding performance in AWGN using floating-point computation and that using fixed-point computation.

Table 4.4: Soft-Decision Decoding Performance with $\alpha = 48$ and $\beta = 48$, in AWGN at BER $= 10^{-6}$ Employing Fixed-Point Computation

Modulation	CC Code Rate	Theoretic Soft-Decision CC Coding Gain (dB)	Soft-Decision CC Coding Gain from Simulation Employing Fixed-Point Computation (dB)	Soft-Decision CC Coding Gain from Simulation Employing Floating-Point Computation(dB)
QPSK	1/2	6.99	5.61	5.62
QPSK	3/4	5.74	4.51	4.82
16QAM	1/2	6.99	6.32	6.28
16QAM	3/4	5.74	5.41	5.43
64QAM	1/2	6.99	6.02	6.35
64QAM	2/3	6.02	5.62	5.97
64QAM	3/4	5.74	5.38	5.64

4.4 Implementation on DSP

We introduce the compiler options that control the operation of the compiler. CCS compiler offers high-level language support by transforming C/C++ code into more efficient assembly language source code. The compiler options can be used to optimize our code size and the executing performance.

The major compiler options we utilize are -o3, -pm -op2, no -ms.

- -pm -op2. In the CCS compiler option, -pm and -op2 are combined into one option:
  - -pm: Give the compiler global access to the whole program or module and allows it to be more aggressive in ruling out dependencies.
  - -op2: Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler. This improves

variable analysis and allowed assumptions.

- no -ms. Speed most critical.

4.4.1 Profile of the DSP code

In this section, we show the optimized profile of our convolution code, which concatenates the randomizer, interleaver and modulator. Table 4.5 show the execution speed of the final concatenated program for processing different data length define in IEEE 802.16e on DSP. In Table 4.5, we see that there is almost 97% execution time in the Viterbi decoder. In Fig. 4.8, we show the C code of Viterbi decoder. We do the full search to compare the Euclidean distance with the stage diagram every two bits. For coding rate 1/2 and codeword length 576, it needs to compare 768*64=49152 times. In the C code, there are 768*32=24576 loops when decoding one block. We show the assembly code in Fig. 4.9, 4.10, 4.11, 4.12 and 4.13. In these figures, we can see that the parallelism is good. In one cycle, it can execute more than five instructions usually.

Table 4.6 show the processing rate in different mandatory coding and modulation mode. In the encoder, we can approach the data rate more than 10M bps. But in the decoder, there is a bottleneck in viterbi algorithm. Viterbi algorithm need large operation and get high complexity. In our decoder, we can approach the data rate about only 8k bps.

Table 4.7 show the code size in different mandatory coding and modulation mode. The average code size is 2021 bytes in the encoder and is 5739 bytes in the decoder.

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 [30]. Using this co-processor may be helpful in raising the decoding speed. But its use requires

```
for (i=0;i<CC_Output_Expand_Block_rate;i++)
{
//////create the Trellis Tree //////

  //discard the first repeat_number metrice
  if(i==AA)
  {
   for(p=0;p<64;p++)
          {
    nod_prestate[0][p]=nod_prestate[AA][p];
          }
  }

  if(i>=AA)   j=i-AA;
  else j=i;

  k1=k;
  k=(k+1)%2;
          for (m=0; m<64; m+=2)
          {
                    tempA=-sta[m][1]*v[i<<1];
                    tempB=-sta[m][2]*v[(i<<1)+1];
                    tempC=-sta[m+1][1]*v[i<<1];
                    tempD=-sta[m+1][2]*v[(i<<1)+1];
                    tempE=-sta[m][4]*v[i<<1];
                    tempF=-sta[m][5]*v[(i<<1)+1];
                    tempG=-sta[m+1][4]*v[i<<1];
                    tempH=-sta[m+1][5]*v[(i<<1)+1];

                    metric0[m]=nod_distance[k][m]+tempA+tempB;
                    metric0[m+1]=nod_distance[k][m+1]+tempC+tempD;
                    metric1[m]=nod_distance[k][m]+tempE+tempF;
                    metric1[m+1]=nod_distance[k][m+1]+tempG+tempH;

                   if(metric0[m]<metric0[m+1]){
                      nod_distance[k1][sta[m][0]]=metric0[m];
                      nod_prestate[j+1][sta[m][0]]=m ;
                   }
                   else{
                      nod_distance[k1][sta[m][0]]=metric0[m+1];
                      nod_prestate[j+1][sta[m][0]]=m+1 ;
                   }

                   if(metric1[m]<metric1[m+1]){
                        nod_distance[k1][sta[m][3]]=metric1[m];
                        nod_prestate[j+1][sta[m][3]]=m;
                   }
                   else{
                        nod_distance[k1][sta[m][3]]=metric1[m+1];
                        nod_prestate[j+1][sta[m][3]]=m+1;
                   }

          }
                   //end switch
}
```


Figure 4.8: The C code of Viterbi decoder.

```
//////////////////////Vitarbi Decoder////////////////////////////////////////
k=0;
000829D8 0700A358             MVK.L1        0,A14
for (i=0;i<CC_Output_Expand_Block_rate;i++)
000829BC 006C0ADA             CMPLT.L2      0,B27,B0
000829C0 30EB4121      [!B0]  BNOP.S1       L184,2
000829C4 2215C22A  ||  [ B0]  MVK.S2        0x2b84,B4
000829C8 223C9C42      [ B0]  ADDAW.D2      SP,B4,B4
000829CC 00000000             NOP
000829D0 27901FD8      [ B0]  OR.L1X        0,B4,A15
000829DC 0D802029             MVK.S1        0x0040,A27
000829E0 0600A358  ||         MVK.L1        0,A12
000829E4          DW$L$_SoftCC_Decoder$90$B:
00082D60 06302058             ADD.L1        1,A12,A12
00082D64 0087E05A             SUB.L2        B1,1,B1
00082D68 4F21A120      [ B1]  BNOP.S1       DW$L$_SoftCC_Decoder$90$B,5
{
//////create the Trellis Tree //////

   //discard the first repeat_number metrics
   if(i==AA)
   000829D4 00EC0FDA             OR.L2         0,B27,B1
   000829E4          DW$L$_SoftCC_Decoder$90$B:
   000829E4 01801828             MVK.S1        0x0030,A3
   000829E8 000D8A78             CMPEQ.L1      A12,A3,A0
   000829EC D044A120      [!A0]  BNOP.S1       DW$L$_SoftCC_Decoder$92$B,5
   {
     for(p=0;p<64;p++)
     {
      nod_prestate[0][p]=nod_prestate[AA][p];
       000829F0          DW$L$_SoftCC_Decoder$91$B:
       000829F0 020DC82A             MVK.S2        0x1b90,B4
       000829F4 003C807A             ADD.L2        B4,SP,B0
       000829F8 020083E6             LDDW.D2T2     *+B0[0x4],B5:B4
       000829FC 0100E3E6             LDDW.D2T2     *+B0[0x7],B3:B2
       00082A00 030063E6             LDDW.D2T2     *+B0[0x3],B7:B6
       00082A04 0401E3E6             LDDW.D2T2     *+B0[0xF],B9:B8
       00082A08 0600A3E6             LDDW.D2T2     *+B0[0x5],B13:B12
       00082A0C 02901FD9             OR.L1X        0,B4,A5
       00082A10 0500C3E6  ||         LDDW.D2T2     *+B0[0x6],B11:B10
       00082A14 01941FD9             OR.L1X        0,B5,A3
       00082A18 020043E6  ||         LDDW.D2T2     *+B0[0x2],B5:B4
       00082A1C 0F0103E6             LDDW.D2T2     *+B0[0x8],B31:B30
       00082A20 0E0123E6             LDDW.D2T2     *+B0[0x9],B29:B28
       00082A24 0D0143E6             LDDW.D2T2     *+B0[0xA],B27:B26
       00082A28 0B0163E6             LDDW.D2T2     *+B0[0xB],B23:B22
       00082A2C 04901FD9             OR.L1X        0,B4,A9
       00082A30 0A0183E6  ||         LDDW.D2T2     *+B0[0xC],B21:B20
       00082A34 03941FD9             OR.L1X        0,B5,A7
       00082A38 020023E6  ||         LDDW.D2T2     *+B0[0x1],B5:B4
       00082A3C 0901A3E6             LDDW.D2T2     *+B0[0xD],B19:B18
       00082A40 0801C3E6             LDDW.D2T2     *+B0[0xE],B17:B16
       00082A44 0100F2FF             STW.D2T2      B2,*+SP[0xF2]
       00082A48 01180FDB  ||         OR.L2         0,B6,B2
       00082A4C 032416A2  ||         OR.S2X        0,A9,B6
       00082A50 0300E8FE             STW.D2T2      B6,*+SP[0xE8]
       00082A54 08901FD9             OR.L1X        0,B4,A17
       00082A58 040102FF  ||         STW.D2T2      B8,*+SP[0x102]
       00082A5C 04141FDA  ||         OR.L2X        0,A5,B8
       00082A60 04141FD9             OR.L1X        0,B5,A8
```

Figure 4.9: The assembly code of Viterbi decoder (1/5).

```
    00082A60 04141FD9              OR.L1X        0,B5,A8
    00082A64 020003E6   ||         LDDW.D2T2     *+B0[0x0],B5:B4
    00082A68 03441FDB              OR.L2X        0,A17,B6
    00082A6C 0B00FAFE   ||         STW.D2T2      B22,*+SP[0xFA]
    00082A70 0300E6FF              STW.D2T2      B6,*+SP[0xE6]
    00082A74 0B201FDA   ||         OR.L2X        0,A8,B22
    00082A78 0D00F8FF              STW.D2T2      B26,*+SP[0xF8]
    00082A7C 0D1C1FDA   ||         OR.L2X        0,A7,B26
    00082A80 0400ECFE              STW.D2T2      B8,*+SP[0xEC]
    00082A84 0200E4FE              STW.D2T2      B4,*+SP[0xE4]
    00082A88 0F00F4FF              STW.D2T2      B30,*+SP[0xF4]
    00082A8C 0F0C1FDA   ||         OR.L2X        0,A3,B30
    00082A90 0100EAFE              STW.D2T2      B2,*+SP[0xEA]
    00082A94 0600EEFE              STW.D2T2      B12,*+SP[0xEE]
    00082A98 0500F0FE              STW.D2T2      B10,*+SP[0xF0]
    00082A9C 0A00FCFE              STW.D2T2      B20,*+SP[0xFC]
    00082AA0 0900FEFE              STW.D2T2      B18,*+SP[0xFE]
    00082AA4 080100FE              STW.D2T2      B16,*+SP[0x100]
    00082AA8 0E00F6FF              STW.D2T2      B28,*+SP[0xF6]
    00082AAC 0E1C0FDA   ||         OR.L2         0,B7,B28
    00082AB0 0280E5FE              STW.D2T2      B5,*+SP[0xE5]
    00082AB4 0B00E7FE              STW.D2T2      B22,*+SP[0xE7]
    00082AB8 0D00E9FE              STW.D2T2      B26,*+SP[0xE9]
    00082ABC 0F00EDFE              STW.D2T2      B30,*+SP[0xED]
    00082AC0 0E00EBFE              STW.D2T2      B28,*+SP[0xEB]
    00082AC4 0680EFFE              STW.D2T2      B13,*+SP[0xEF]
    00082AC8 0580F1FE              STW.D2T2      B11,*+SP[0xF1]
    00082ACC 0180F3FE              STW.D2T2      B3,*+SP[0xF3]
    00082AD0 0F80F5FE              STW.D2T2      B31,*+SP[0xF5]
    00082AD4 0E80F7FE              STW.D2T2      B29,*+SP[0xF7]
    00082AD8 0D80F9FE              STW.D2T2      B27,*+SP[0xF9]
    00082ADC 0B80FBFE              STW.D2T2      B23,*+SP[0xFB]
    00082AE0 0A80FDFE              STW.D2T2      B21,*+SP[0xFD]
    00082AE4 0980FFFE              STW.D2T2      B19,*+SP[0xFF]
    00082AE8 088101FE              STW.D2T2      B17,*+SP[0x101]
    00082AEC 048103FE              STW.D2T2      B9,*+SP[0x103]
  }
}

if(i>=AA)   j=i-AA;
00082AF0           DW$L$_SoftCC_Decoder$92$B:
00082AF0 01801829               MVK.S1        0x0030,A3
00082AF4 027FE82A  ||           MVK.S2        0xffffffd0,B4
00082AF8 000D8AF8               CMPLT.L1      A12,A3,A0
00082AFC C3301FDA       [ A0]   OR.L2X        0,A12,B6
00082B00 D330907A       [!A0]   ADD.L2X       B4,A12,B6
00082B04 06B80FD8               OR.L1         0,A14,A13
else j=i;

k1=k;
k=(k+1)%2;
00082B08 01B42059               ADD.L1        1,A13,A3
00082B0C 0215A12B  ||           MVK.S2        0x2b42,B4
00082B10 0FBD8A64  ||           LDW.D1T1      *+A15[A12],A31
00082B14 018FE9A1               SHRU.S1       A3,0x1f,A3
00082B18 023C9C43  ||           ADDAW.D2      SP,B4,B4
00082B1C 0415812A  ||           MVK.S2        0x2b02,B8
00082B20 018DA079               ADD.L1        A13,A3,A3
00082B24 0380C42B  ||           MVK.S2        0x0188,B7
```

Figure 4.10: The assembly code of Viterbi decoder (2/5).

```
00082B28 0D3D1C42  ||         ADDAW.D2     SP,B8,B26
00082B2C 018C2059             ADD.L1       1,A3,A3
00082B30 0F1016A1  ||         OR.S1X       0,B4,A30
00082B34 02B51CA2  ||         SHL.S2X      A13,0x8,B5
00082B38 026FCF59             AND.L1       -2,A3,A5
00082B3C 0498ECA2  ||         SHL.S2       B6,0x7,B9
00082B40 0195A0F8             SUB.L1       A13,A5,A3
00082B44 070C2058             ADD.L1       1,A3,A14
00082B48 01B90CA0             SHL.S1       A14,0x8,A3
00082B4C 00000000             NOP
00082B50 01BD1059             ADD.L1X      8,SP,A3
00082B54 020DF07A  ||         ADD.L2X      SP,A3,B4
00082B58 0D90E07A             ADD.L2       B7,B4,B27
 for (m=0; m<64; m+=2)
 00082B5C 088D6059             ADD.L1       11,A3,A17
 00082B60 010403E3  ||         MVC.S2       CSR,B2
 00082B64 00002041  ||         MVK.D1       1,A0
 00082B68 080D705B  ||         ADD.L2X      11,A3,B16
 00082B6C 0181C829  ||         MVK.S1       0x0390,A3
 00082B70 0A000042  ||         MVK.D2       0,B20
 00082B74 08C74059             SUB.L1       A17,6,A17
 00082B78 020BCF5B  ||         AND.L2       -2,B2,B4
 00082B7C 05BC7AB1  ||         ADD.D1X      A3,SP,A11
 00082B80 0181C829  ||         MVK.S1       0x0390,A3
 00082B84 CA4020A7  || [ A0]   LDB.D2T2     *-B16[0x1],B20
 00082B88 0A80002A  ||         MVK.S2       0x0000,B21
 00082B8C 09446025             LDB.D1T1     *-A17[0x3],A18
 00082B90 009003A3  ||         MVC.S2       B4,CSR
 00082B94 0980A359  ||         MVK.L1       0,A19
 00082B98 0B800029  ||         MVK.S1       0x0000,A23
--------------------------------------------------------------------
 00082B9C 0400A35B  ||         MVK.L2       0,B8
 00082BA0 096C33E6  ||         LDDW.D2T2    *++B27[0x1],B19:B18
 00082BA4 0200C82B             MVK.S2       0x0190,B4
 00082BA8 C9C48025  || [ A0]   LDB.D1T1     *-A17[0x4],A19
 00082BAC 053C7079  ||         ADD.L1X      A3,SP,A10
 00082BB0 0180C829  ||         MVK.S1       0x0190,A3
 00082BB4 0B00A35B  ||         MVK.L2       0,B22
 00082BB8 0E202942  ||         ADD.D2       B8,0x1,B28
 00082BBC 0FBC807B             ADD.L2       B4,SP,B31
 00082BC0 0201C82B  ||         MVK.S2       0x0390,B4
 00082BC4 CBC46225  || [ A0]   LDB.D1T1     *+A17[0x3],A23
 00082BC8 0180C829  ||         MVK.S1       0x0190,A3
 00082BCC 013C7078  ||         ADD.L1X      A3,SP,A2
 00082BD0 0F3C807B             ADD.L2       B4,SP,B30
 00082BD4 0200C82B  ||         MVK.S2       0x0190,B4
 00082BD8 CAC42027  || [ A0]   LDB.D1T2     *-A17[0x1],B21
 00082BDC 0D00A359  ||         MVK.L1       0,A26
 00082BE0 CD4080A5  || [ A0]   LDB.D2T1     *-B16[0x4],A26
 00082BE4 0A800028  ||         MVK.S1       0x0000,A21
 00082BE8 0EBC807B             ADD.L2       B4,SP,B29
 00082BEC 01BC7079  ||         ADD.L1X      A3,SP,A3
 00082BF0 0E5421A1  ||         ADD.S1       1,A21,A28
 00082BF4 CB4196A6  || [ A0]   LDB.D2T2     *B16++[0xC],B22
 00082BF8 0A141FD8             OR.L1X       0,B5,A20
 00082BFC 0B241FD8             OR.L1X       0,B9,A22
 00082C00          DW$L$_SoftCC_Decoder$94$B:
 00082C00 0201C82B             MVK.S2       0x0390,B4
 00082C04 03C59627  ||         LDB.D1T2     *A17++[0xC],B7
 00082C08 03CBE480  ||         MPYHL.M1     A31,A18,A7
 00082C0C 0B8C807B             ADD.L2       B4,SP,B23
```

Figure 4.11: The assembly code of Viterbi decoder (3/5).

```
00082C10 02FE6C80  ||        MPY.M1       A19,A31,A5
00082C14 045FE481            MPYHL.M1     A31,A23,A8
00082C18 027E9C82  ||        MPY.M2X      B20,A31,B4
00082C1C 027EBC83            MPY.M2X      B21,A31,B4
00082C20 02FF4C81  ||        MPY.M1       A26,A31,A5
00082C24 029652F8  ||        SUB.L1X      B18,A5,A5
00082C28 027ED883            MPYLH.M2X    B22,A31,B4
00082C2C 031260FB  ||        SUB.L2       B19,B4,B6
00082C30 0C1CA0F8  ||        SUB.L1       A5,A7,A24
00082C34 00E006A1            OR.S1        0,A24,A1
00082C38 027CF883  ||        MPYLH.M2X    B7,A31,B4
00082C3C 089240FB  ||        SUB.L2       B18,B4,B17
00082C40 029672F8  ||        SUB.L1X      B19,A5,A5
00082C44 0390C0FB            SUB.L2       B6,B4,B7
00082C48 0CA0A0F8  ||        SUB.L1       A5,A8,A25
00082C4C 0C6833C5            STDW.D2T1    A25:A24,*++B26[0x1]
00082C50 031220FB  ||        SUB.L2       B17,B4,B6
00082C54 00670AF8  ||        CMPLT.L1     A24,A25,A0
00082C58 049C1FD9            OR.L1X       0,B7,A9
00082C5C 026416A3  ||        OR.S2X       0,A25,B4
00082C60 001CCAFB  ||        CMPLT.L2     B6,B7,B0
00082C64 DCC2E0A7  || [!A0]  LDB.D2T2     *-B16[0x17],B25
00082C68 CEC62024  || [ A0]  LDB.D1T1     *-A17[0x11],A29
00082C6C 03783347            STDW.D1T2    B7:B6,*++A30[0x1]
00082C70 384280A4  || [!B0]  LDB.D2T1     *-B16[0x14],A16
00082C74 2C4280A6     [ B0]  LDB.D2T2     *-B16[0x14],B24
00082C78 00002000            NOP          2
00082C7C 03973C43            ADDAW.D2     B5,B25,B7
00082C80 02DBBA40  ||        ADDAH.D1     A22,A29,A5
00082C84 02956079            ADD.L1       A11,A5,A5
00082C88 03DA1A41  ||        ADDAH.D1     A22,A16,A7
00082C8C 08A73A43  ||        ADDAH.D2     B9,B25,B17
00082C90 039FE07A  ||        ADD.L2       B31,B7,B7
00082C94 0C1D4079            ADD.L1       A10,A7,A24
00082C98 03D3BC41  ||        ADDAW.D1     A20,A29,A7
00082C9C 0247C07B  ||        ADD.L2       B30,B17,B4
00082CA0 02804051  ||        ADDK.S1      128,A5
00082CA4 D21C02F6  || [!A0]  STW.D2T2     B4,*+B7[0x0]
00082CA8 0C004051            ADDK.S1      128,A24
00082CAC 039C4079  ||        ADD.L1       A2,A7,A7
00082CB0 04521C41  ||        ADDAW.D1     A20,A16,A8
00082CB4 03971C43  ||        ADDAW.D2     B5,B24,B7
00082CB8 02004052  ||        ADDK.S2      128,B4
00082CBC 01A06079            ADD.L1       A3,A8,A3
00082CC0 0AD441A1  ||        ADD.S1       2,A21,A21
00082CC4 CA940255  || [ A0]  STH.D1T1     A21,*+A5[0x0]
00082CC8 039FA07B  ||        ADD.L2       B29,B7,B7
00082CCC 08A71A42  ||        ADDAH.D2     B9,B24,B17
00082CD0 006EAAF9            CMPLT.L1     A21,A27,A0
00082CD4 C09C0275  || [ A0]  STW.D1T1     A1,*+A7[0x0]
00082CD8 DE1002D7  || [!A0]  STH.D2T2     B28,*+B4[0x0]
00082CDC 0246E07A  ||        ADD.L2       B23,B17,B4
00082CE0 348C0275     [!B0]  STW.D1T1     A9,*+A3[0x0]
00082CE4 CFFFE413  || [ A0]  B.S2         DW$L$_SoftCC_Decoder$94$B
00082CE8 231C02F6  || [ B0]  STW.D2T2     B6,*+B7[0x0]
00082CEC 02004053            ADDK.S2      128,B4
00082CF0 0181C829  ||        MVK.S1       0x0390,A3
00082CF4 C96C33E7  || [ A0]  LDDW.D2T2    *++B27[0x1],B19:B18
00082CF8 C9446024  || [ A0]  LDB.D1T1     *-A17[0x3],A18
00082CFC 0420405B            ADD.L2       2,B8,B8
```

Figure 4.12: The assembly code of Viterbi decoder (4/5).

```
00082C88 03DA1A41  ||         ADDAH.D1      A22,A16,A7
00082C8C 08A73A43  ||         ADDAH.D2      B9,B25,B17
00082C90 039FE07A  ||         ADD.L2        B31,B7,B7
00082C94 0C1D4079             ADD.L1        A10,A7,A24
00082C98 03D3BC41  ||         ADDAW.D1      A20,A29,A7
00082C9C 0247C07B  ||         ADD.L2        B30,B17,B4
00082CA0 02804051  ||         ADDK.S1       128,A5
00082CA4 D21C02F6  || [!A0]   STW.D2T2      B4,*+B7[0x0]
00082CA8 0C004051             ADDK.S1       128,A24
00082CAC 039C4079  ||         ADD.L1        A2,A7,A7
00082CB0 04521C41  ||         ADDAW.D1      A20,A16,A8
00082CB4 03971C43  ||         ADDAW.D2      B5,B24,B7
00082CB8 02004052  ||         ADDK.S2       128,B4
00082CBC 01A06079             ADD.L1        A3,A8,A3
00082CC0 0AD441A1  ||         ADD.S1        2,A21,A21
00082CC4 CA940255  || [ A0]   STH.D1T1      A21,*+A5[0x0]
00082CC8 039FA07B  ||         ADD.L2        B29,B7,B7
00082CCC 08A71A42  ||         ADDAH.D2      B9,B24,B17
00082CD0 006EAAF9             CMPLT.L1      A21,A27,A0
00082CD4 C09C0275  || [ A0]   STW.D1T1      A1,*+A7[0x0]
00082CD8 DE1002D7  || [!A0]   STH.D2T2      B28,*+B4[0x0]
00082CDC 0246E07A  ||         ADD.L2        B23,B17,B4
00082CE0 348C0275     [!B0]   STW.D1T1      A9,*+A3[0x0]
00082CE4 CFFFE413  || [ A0]   B.S2          DW$L$_SoftCC_Decoder$94$B
00082CE8 231C02F6  || [ B0]   STW.D2T2      B6,*+B7[0x0]
00082CEC 02004053             ADDK.S2       128,B4
00082CF0 0181C829  ||         MVK.S1        0x0390,A3
00082CF4 C96C33E7  || [ A0]   LDDW.D2T2     *++B27[0x1],B19:B18
00082CF8 C9446024  || [ A0]   LDB.D1T1      *-A17[0x3],A18
00082CFC 0420405B             ADD.L2        2,B8,B8
00082CEC 02004053             ADDK.S2       128,B4
00082CF0 0181C829  ||         MVK.S1        0x0390,A3
00082CF4 C96C33E7  || [ A0]   LDDW.D2T2     *++B27[0x1],B19:B18
00082CF8 C9446024  || [ A0]   LDB.D1T1      *-A17[0x3],A18
00082CFC 0420405B             ADD.L2        2,B8,B8
00082D00 241002D7  || [ B0]   STH.D2T2      B8,*+B4[0x0]
00082D04 0181C829  ||         MVK.S1        0x0390,A3
00082D08 05BC7079  ||         ADD.L1X       A3,SP,A11
00082D0C 0200C82B  ||         MVK.S2        0x0190,B4
00082D10 C9C48024  || [ A0]   LDB.D1T1      *-A17[0x4],A19
00082D14 0180C829             MVK.S1        0x0190,A3
00082D18 053C7079  ||         ADD.L1X       A3,SP,A10
00082D1C 0201C82B  ||         MVK.S2        0x0390,B4
00082D20 0FBC807B  ||         ADD.L2        B4,SP,B31
00082D24 CBC46225  || [ A0]   LDB.D1T1      *+A17[0x3],A23
00082D28 CA4020A6  || [ A0]   LDB.D2T2      *-B16[0x1],B20
00082D2C 0180C829             MVK.S1        0x0190,A3
00082D30 0200C82B  ||         MVK.S2        0x0190,B4
00082D34 013C7079  ||         ADD.L1X       A3,SP,A2
00082D38 0F3C807B  ||         ADD.L2        B4,SP,B30
00082D3C CAC42027  || [ A0]   LDB.D1T2      *-A17[0x1],B21
00082D40 CD4080A4  || [ A0]   LDB.D2T1      *-B16[0x4],A26
00082D44 3E600255     [!B0]   STH.D1T1      A28,*+A24[0x0]
00082D48 0E20205B  ||         ADD.L2        1,B8,B28
00082D4C 01BC7079  ||         ADD.L1X       A3,SP,A3
00082D50 0EBC81E3  ||         ADD.S2        B4,SP,B29
00082D54 0E5421A1  ||         ADD.S1        1,A21,A28
00082D58 CB4196A6  || [ A0]   LDB.D2T2      *B16++[0xC],B22
00082D5C                 DW$L$_SoftCC_Decoder$96$B:
00082D5C 008803A2             MVC.S2        B2,CSR
```

Figure 4.13: The assembly code of Viterbi decoder (5/5).

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 334
;*      Loop opening brace source line   : 335
;*      Loop closing brace source line   : 368
;*      Known Minimum Trip Count         : 32
;*      Known Maximum Trip Count         : 32
;*      Known Max Trip Count Factor      : 32
;*      Loop Carried Dependency Bound(^) : 17
;*      Unpartitioned Resource Bound     : 16
;*      Partitioned Resource Bound(*)    : 16
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     2        1
;*      .S units                     6        7
;*      .D units                    15       16*
;*      .M units                     4        4
;*      .X cross paths               8        5
;*      .T address paths            11       12
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           2        0     (.L or .S unit)
;*      Addition ops (.LSD)         14       15     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             5        4
;*      Bound(.L .S .D .LS .LSD)    13       13
;*
;*      Searching for software pipeline schedule at ...
```

Figure 4.14: Software pipeline information for Viterbi decoder.

Table 4.5: Final Profile of Convolution Code (Cycles)

Fucntion	QPSK rate 1/2 36 bytes	QPSK rate 3/4 36 bytes	16QAM rate 1/2 36 bytes	16QAM rate 3/4 36 bytes	64QAM rate 1/2 36 bytes	64QAM rate 2/3 24 bytes	64QAM rate 3/4 27 bytes
Randomizer	4447	4433	4453	4443	4448	3009	3352
Encoder	3124	4133	3137	4139	3130	2954	3110
Interleaver	3526	2360	4747	3169	19678	9854	9836
Modulator	3845	2556	10141	6993	9281	5066	4846
TX total	14942	13482	22478	18744	36537	20883	21144
De-modulator	762	462	4394	2926	2569	1627	1629
De-interleaver	3590	2396	4163	2786	6487	3243	3243
Decoder	337017	337571	336995	337635	337111	254305	275019
De-randomizer	4492	4508	4496	4499	4494	3010	3382
RX total	345861	344937	350048	347846	350661	262185	283273

Table 4.6: Final Profile of Convolution Code (Processing Rate)

Processing Rate (kbps)	QPSK rate 1/2	QPSK rate 3/4	16QAM rate 1/2	16QAM rate 3/4	64QAM rate 1/2	64QAM rate 2/3	64QAM rate 3/4
Encoder	19724	21361	12812	15364	7882	9194	10215
Decoder	832	835	822	828	821	732	762

Table 4.7: Final Profile of Convolution Code (Code Size)

Code Size (byte)	QPSK rate 1/2	QPSK rate 3/4	16QAM rate 1/2	16QAM rate 3/4	64QAM rate 1/2	64QAM rate 2/3	64QAM rate 3/4
Encoder	1868	1684	2164	1980	2176	2284	1992
Decoder	4648	5012	5152	5516	6400	6684	6764

ES 1896

study and testing of the "enhanced direct memory access (EDMA)" mechanism of the C64x chips, which is bypassed in the present study.

Chapter 5

Simulation and DSP Implementation of LDPC Encoder and Decoder

In this chapter, we present some simulation results for the LDPC codec in IEEE 802.16e. It contains floating-point, fixed-point simulation and DSP implementation.

5.1 Performance in AWGN Channel with Floating-Point Processing

5.1.1 Number of Iterations

The iteration number is a most important factor in the decoding algorithm. This number affects the decoding accuracy and system complexity. A larger iteration number usually leads to better performance. But the complexity and the and the latency are increased. We compare the performance with iteration numbers between 10 and 70 for BP decoding of the rate 1/2, length 576 code with QPSK modulation in Fig. 5.1.

In Fig. 5.1, the performance at 10 iterations is obviously inferior to other choice. We can see that if the iteration number is more than 20, then the BER curves are almost the same. To limit the decoding complexity and maintain a reasonable performance, we use 20 as the iteration number in other simulations.

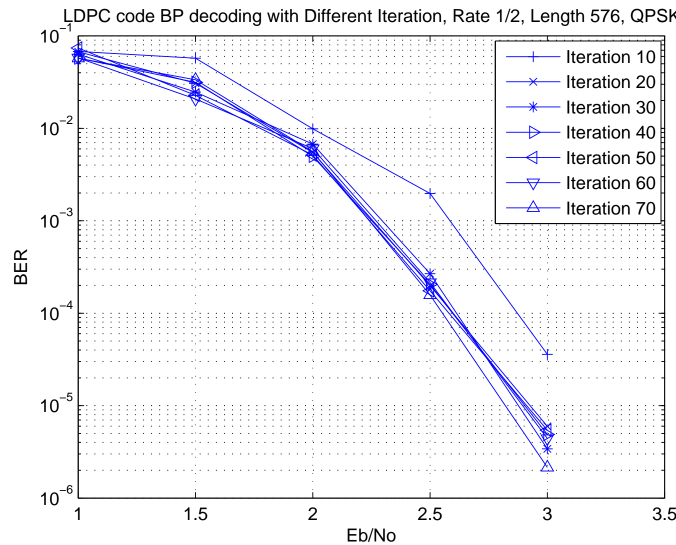


Figure 5.1: LDPC decoding performance in different iteration numbers with floating-point computation.

5.1.2 Performance at Different Codeword Lengths

In convolutional coding the codeword length does not affect the performance. But LDPC code is different. Fig. 5.2 shows the performance at four different codeword lengths at code rate 1/2 with QPSK modulation and BP decoding with 20 iterations. In Fig. 5.2, as the codeword length becomes longer, the improved performance is obtained. The result is not surprising, because at medium or short code lengths, the BP algorithm is not optimum, owing to correlation among messages passed during iterative decoding [17]. For the codeword length 2304, the coding gain is about 8 dB at BER 10^{-6}. This coding gain value is several dB higher than convolutional coding as obtained in the last chapter.

5.1.3 Performance with Different Modulations

We compare the performance of QPSK, 16QAM and 64QAM at rate 1/2, codeword length 576, with BP decoding with 20 iterations. In Fig. 5.3, the coding gains of QPSK, 16QAM,

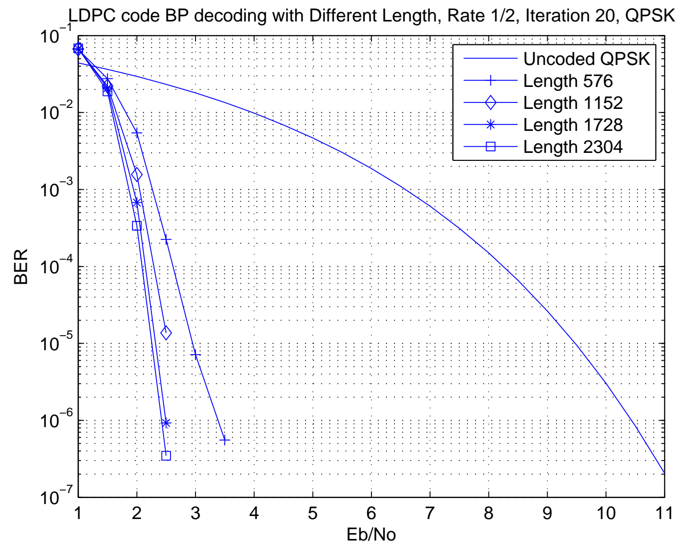


Figure 5.2: LDPC decoding performance in different codeword length with floating-point computation.

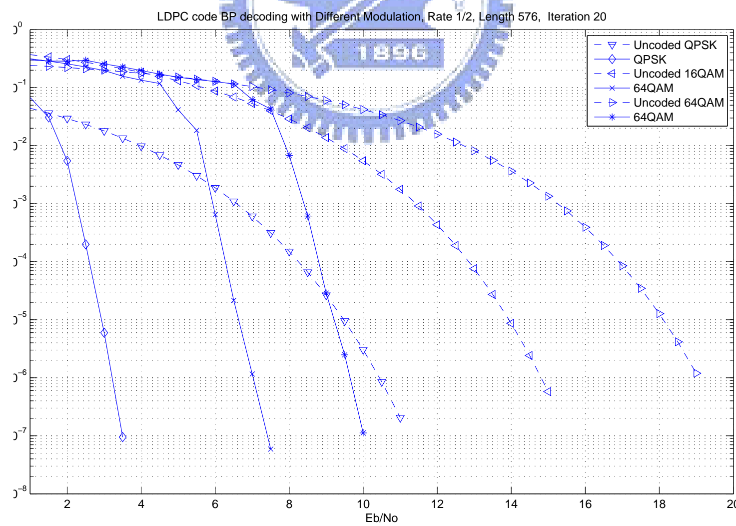


Figure 5.3: LDPC decoding performance with different modulation employing floating-point computation.

Table 5.1: Comparison of Coding Gain Between LDPC Codes and Convolutional Codes at Code Rate 1/2 in AWGN at BER = 10^{-6}

Modulation Type	Convolutional Code	LDPC Code
QPSK	5.62	7.31
16QAM	6.28	7.43
64QAM	6.35	9.32

64QAM are 7.31, 7.43 and 9.32 dB, respectively, at BER=10^{-6}.

In Table 5.1, we compare the coding gains of LDPC codes and convolutional codes. The LDPC codes are obviously better. They are close to Shannon limit [12].

5.1.4 Performance at Different Coding Rates

In IEEE 802.16e, six coding rate are defined for LDPC code, namely $\frac{1}{2}$, $\frac{2}{3}A$, $\frac{2}{3}B$, $\frac{3}{4}A$, $\frac{3}{4}B$ and $\frac{5}{6}$. In Fig. 5.4, we compare their performance under QPSK, code length 576 and BP decoding with 20 iteration. As mat be anticipated, the best performance is obtained at rate 1/2. As the code rate gets higher, the performance gets worse. Note that the two 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 point of view of "threshold" [32]. 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. In Table 5.2, the threshold of $\frac{2}{3}A$ is only larger than that of $\frac{2}{3}B$ by 0.012 dB. So, 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 method, we also easily explain the relationship between the two BER curves of $\frac{3}{4}A$ and $\frac{3}{4}B$.

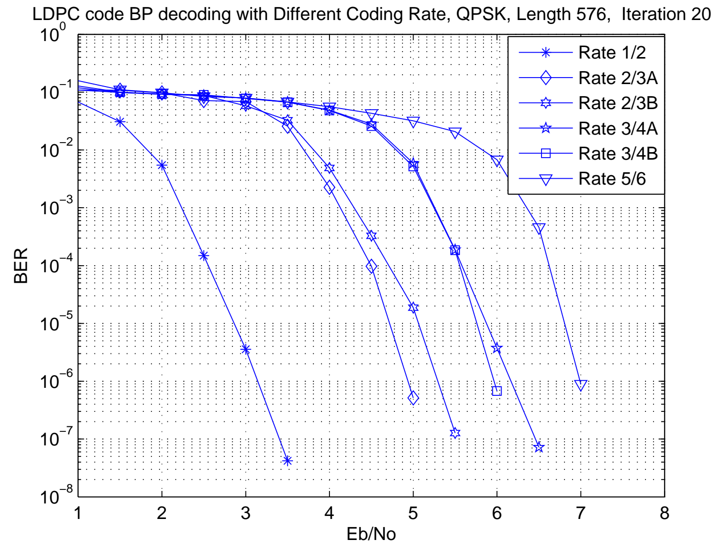


Figure 5.4: LDPC Decoding Performance in Different Coding Rate (floating-point).

Table 5.2: Threshold for Each Code Rate under BPSK Modulation in AWGN Channel [20].

Code Rate	Threshold
$1/2$	0.9273
$2/3A$	0.7282
$2/3B$	0.7163
$3/4A$	0.6358
$3/4B$	0.6446
$5/6$	0.5607

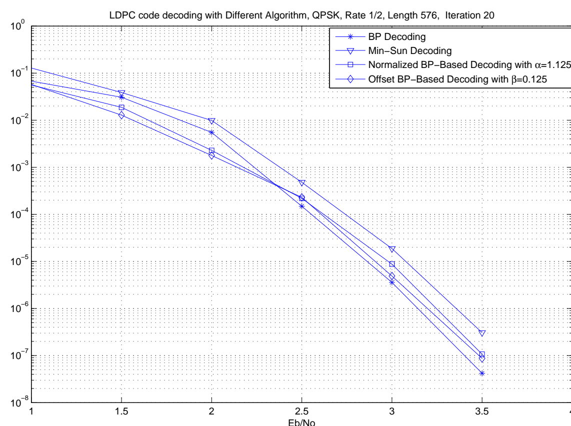


Figure 5.5: LDPC decoding performance using different decoding algorithm employing floating-point computation.

5.1.5 Performance of Reduced-Complexity Algorithm

In chapter 2, we discuss some decoding algorithms with reduced complexity than the BP algorithm. In Fig. 5.5, we compare the performance of four algorithms at code rate 1/2, length 576, QPSK modulation with 20 iterations.

As expected, the min-sun algorithm is obviously worse than the other algorithms. The reason also been discussed previously in chapter 2.

The other two reduced-complexity algorithms, offset BP-based and normalized BP-based, 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 which depends 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 the correlations [17]. The normalized BP-based algorithm slightly outperforms the offset BP-based algorithm, but may also be slightly more complex to im-

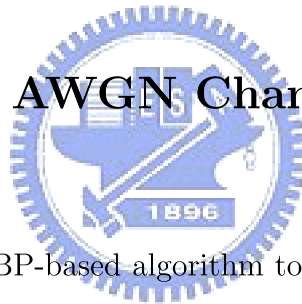
Table 5.3: LDPC Coding Gain between Floating-point and Fixed-point in AWGN at BER $= 10^{-5}$.

Modulation	Floating-Point Coding Gain (dB)	Fixed-Point 12 bit Coding Gain (dB)	Fixed-Point 6 bit Coding Gain (dB)
QPSK	6.58	6.17	5.17
16QAM	7.36	5.89	4.66
64QAM	8.75	6.71	4.65

plement.

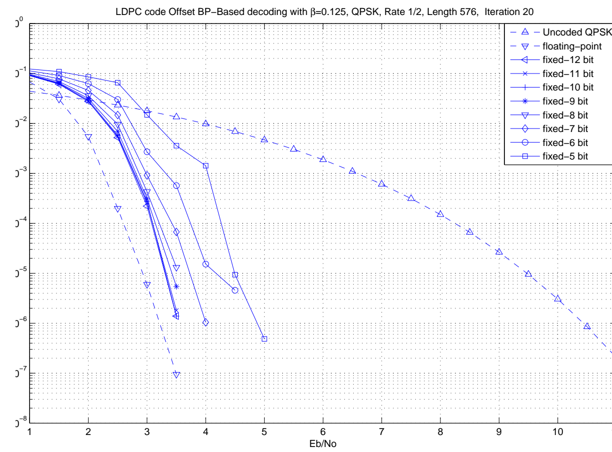
As a result, we choose the offset BP-Based algorithm for DSP implementation. The structure of the algorithm also makes the conversion from floating-point to fixed-point computation easier.

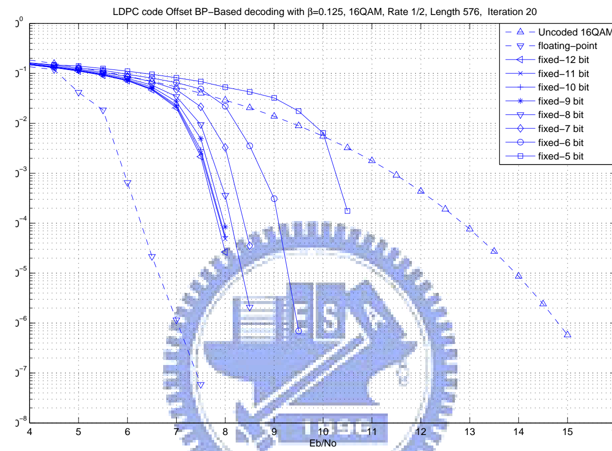
5.2 Performance in AWGN Channel with Fixed-Point Processing



In the above, we select the offset BP-based algorithm to convert the floating-point value to the fixed-point value. By multiplying the original floating-point values by 1000 and rounding the result to integer. Then we use 12 bits to represent this result. Note that we only change the number of bit in the decoder input. We fix the integer part in 4 bit and change the fraction part bit numbers. But the precision of intermediate results decoding computation is still 16 bits.

In Fig. 5.6, we compare the performance when bit number used in decoder is between 5 to 12 for offset BP-based decoding at rate 1/2, length 576 and three different modulations. When we use 8 to 12 bits, the BER curves are almost the same for QPSK, 16QAM and 64QAM. For QPSK, the BER curve when we use 5 bit, is in our acceptable bound. But in





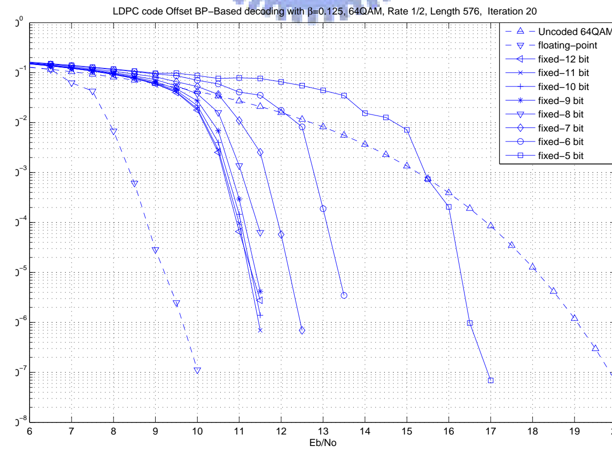


Figure 5.6: LDPC decoding performance at different bit numbers with different modulations employing fixed-point computation.

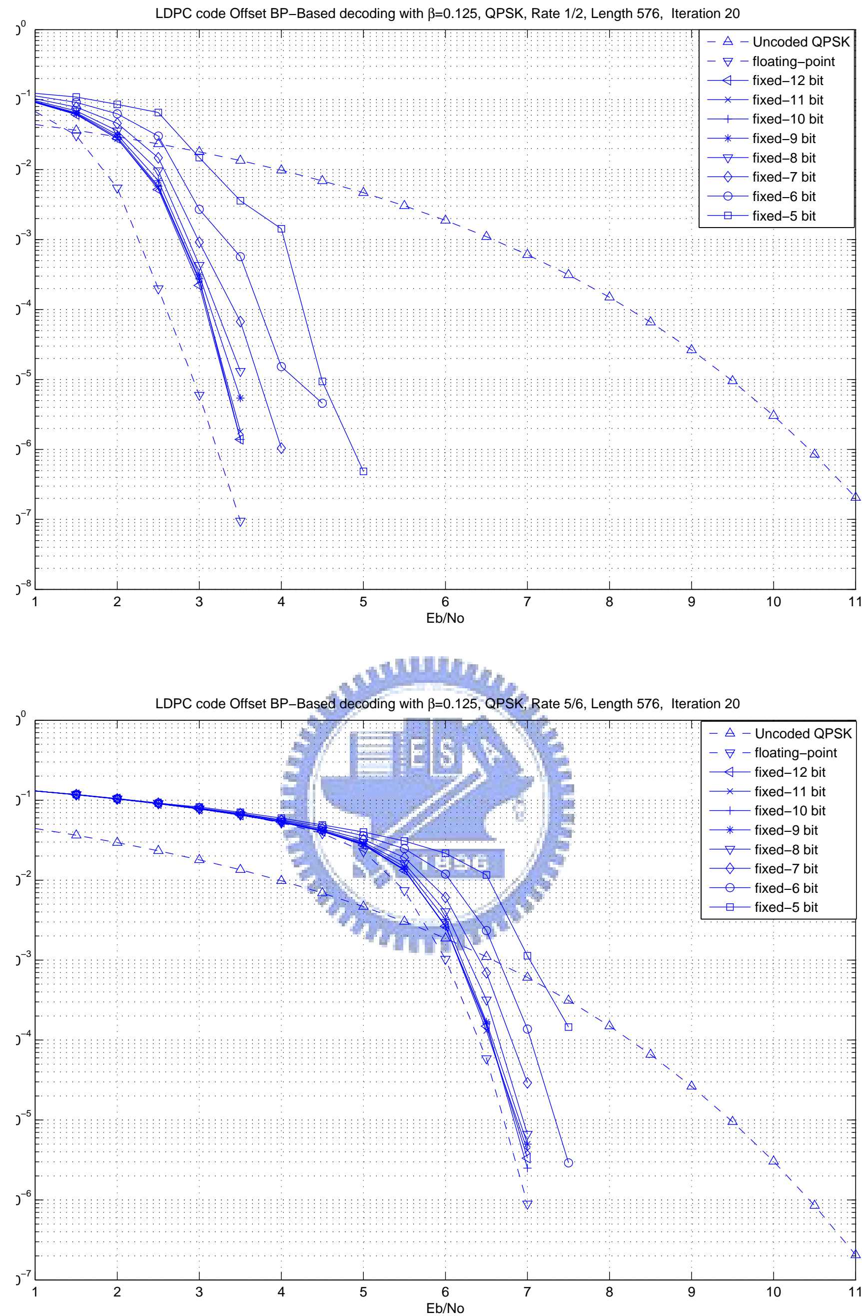
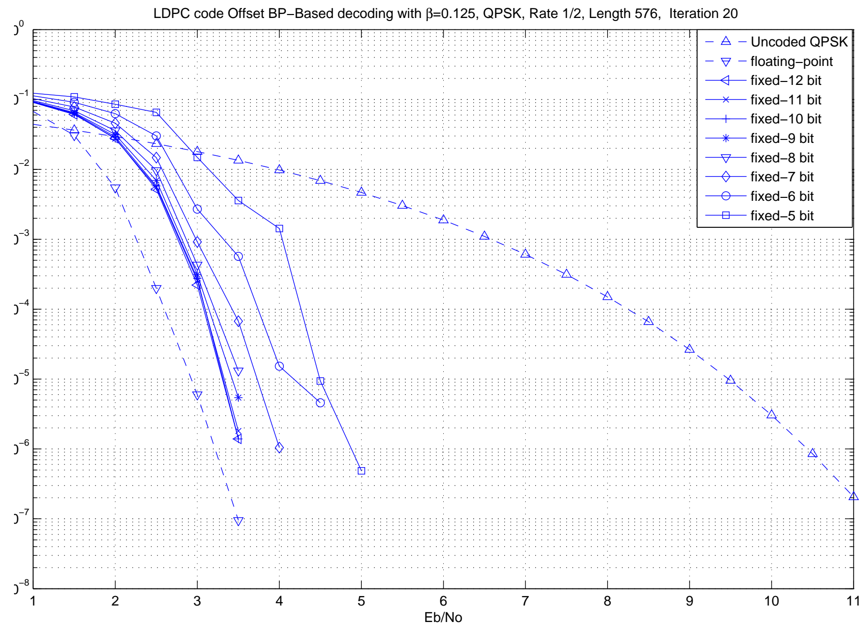


Figure 5.7: LDPC decoding performance at different bit numbers at two different coding rate employing fixed-point computation.



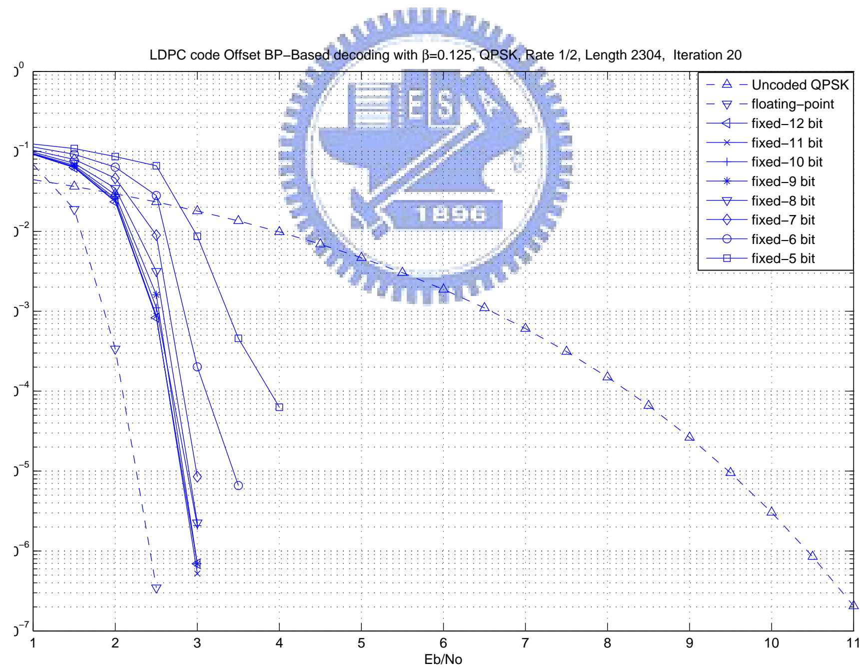


Figure 5.8: LDPC decoding performance at different bit numbers at two different codeword lengths employing fixed-point computation.

16QAM and 64QAM, 6 bit is the limit that we can acceptable. Table 5.3 shows the coding gain between floating-point and fixed-point.

In Fig. 5.7, we compare the performance between coding rates 1/2 and 5/6. When coding rate is 5/6, the SNR need at least 6.5 dB to keep the performance better than uncoded QPSK if we use 7–12 bit. 6 bit is the boundary that we can accept, that the SNR need more than 7 dB to keep the performance better than uncoded QPSK.

In Fig. 5.8, we compare the performance between codeword lengths 576 and 2304. As we discuss above, when length is 576, 5 bit is not enough. But in long codeword length, the BP-Based algorithm is optimum. For the codeword length 2304, it has very good performance. Then, we can also use 5 bit to implement our decoder. The performance just less 1 dB than we use 6 bit to implement.

5.2.1 Profile of the DSP code

Encoder

First, we optimize our code and show the profile. In the case, codeword length 512 and code rate 1/2, it need 21715443 cycles to encode one block. However, the speed performance is awful. As we discuss in chapter 2, LDPC encoder needs to compute the shift size and do the circular shift. Coding one block, it uses circular shift 1002 times at rate 1/2 and codeword length 576. In Table 5.4, 96.3% execution time expends on doing circular shift. In Fig. 5.9, we show the C codes of circular shift. In Fig. 5.10 and 5.11, show the assembly codes of circular shift. At every codeword length, the value of "ZZ" is known. Then we can calculate the circular shift value and compute the circular shifted matrix by ourself. Now, we reduce the C code about circular shift and compute the $p(f, i, j)$ initially. We write the circular shifted matrix into a table. The encoder reads circular shifted matrix from the table. In Table 5.5, it just needs 812491 cycles to encode one block.

Table 5.4: Original Profile of LDPC Encoder (Cycles)

Areas	Cycles	Percentage (%)	Processing Rate (kbits/sec)
LDPC Encoder	21684592	100	13.3
Circular Shift	20881567	96.3	

Table 5.5: Profile of LDPC Encoder with Matrix Table (Cycles)

Areas	Cycles	Processing Rate (kbits/sec)	Improvement (%)
LDPC Encoder (Original)	21684592	13.3	N/A
LDPC Encoder (with Table)	812491	354.5	96.3

Table 5.6: Profile of LDPC Encoder with Different Coding Rates

Profile	Coding Rate					
	1/2	$2/3A$	$2/3B$	$3/4A$	$3/4B$	5/6
Cycle	812491	482041	477639	316731	319386	1748774
Processing Rate (Kbps)	354.5	597.5	603.1	909.3	901.7	1646.9

```
void circular_shift(int I[][zz],int HH[][zz], int shift_size)
{
    int row,colu;
    int temp=0;

    for(row=0;row<zz;row++)
    {
        for(colu=0;colu<zz;colu++)
        {
            if(shift_size==-1)
                HH[row][colu]  =0;
            else
            {
                temp=colu+shift_size;
                while(temp%zz<0)     temp=temp+zz;
                while(temp >= zz)   temp = temp-zz;
                HH[row][temp]=I[row][colu];
            }
        }
    }
}
```

Figure 5.9: The C codes of circular shift.

Table 5.7: Profile of LDPC Decoder with different Coding Rate

Profile	Coding Rate					
	1/2	2/3A	2/3B	3/4A	3/4B	5/6
Cycle	37714286	56177704	58141064	76294841	85273741	93146880
Processing Rate (Kbps)	7.6	5.1	5.0	3.8	3.4	3.1

In Table 5.6, we show the profile with different coding rate when codeword length is 576. In this case, when the coding rate is higher the cycle number is more. Because it need to compute more parity bit, and it need more computation complexity.

Decoder

In this subsection, we show the profile of the LDPC decoder when codeword length 576 and iteration 20. Table 5.7 shows the execution speed and the processing rate of our LDPC decoder on DSP. In advance, the LDPC decoder is more complex.

In our code, coding rate 1/2 and codeword length 576, doing one iteration need the loop:

```
void circular_shift(int I[][zz],int HH[][zz], int shift_size)
{
000D0400          circular_shift:
000D0400 01BC54F6           STW.D2T2      B3,*SP--[0x2]
000D0404 0980A359           MVK.L1        0,A19
000D0408 02981FDB  ||       OR.L2X        0,A6,B5
000D040C 041016A1  ||       OR.S1X        0,B4,A8
000D0410 049008F0  ||       OR.D1         0,A4,A9
    int row,colu;
    int temp=0;

    for(row=0;row<zz;row++)
    000D0414 0880A358           MVK.L1        0,A17
    000D0418 03800C2A           MVK.S2        0x0018,B7
    000D041C 03441FDA           OR.L2X        0,A17,B6
    000D0528          DW$L$_circular_shift__FPA24_iT1i$17$E:
    000D0528 08803050           ADDK.S1       96,A17
    000D052C 001FF059           SUB.L1X       B7,1,A0
    000D0530 039FE05A  ||       SUB.L2        B7,1,B7
    000D0534 CFC0A121     [ A0] BNOP.S1       L1,5
    000D0538 C3441FDA  || [ A0] OR.L2X        0,A17,B6
    {
        for(colu=0;colu<zz;colu++)
        000D0420          L1:
        000D0420 0380A358           MVK.L1        0,A7
        000D0514          DW$L$_circular_shift__FPA24_iT1i$17$B:
        000D0514 039C2059           ADD.L1        1,A7,A7
        000D0518 01800C29  ||       MVK.S1        0x0018,A3
        000D051C 0318805A  ||       ADD.L2        4,B6,B6
        000D0520 000CEAF8           CMPLT.L1      A7,A3,A0
        000D0524 CFC1A120     [ A0] BNOP.S1       DW$L$_circular_shift__FPA24_iT1i$2$E,5
        {
            if(shift_size==-1)
            000D0424          DW$L$_circular_shift__FPA24_iT1i$2$E:
            000D0424 0017EA5A           CMPEQ.L2      -1,B5,B0
            000D0428 203BA120     [ B0] BNOP.S1       DW$L$_circular_shift__FPA24_iT1i$16$B,5
        //  if(shift_size <0)
            {
                HH[row][colu]  =0;
                000D050C          DW$L$_circular_shift__FPA24_iT1i$16$B:
                000D050C 01991078           ADD.L1X       A8,B6,A3
                000D0510 098C0274           STW.D1T1      A19,*+A3[0x0]
            }

            else
            {
                temp=colu+shift_size;
                000D042C          DW$L$_circular_shift__FPA24_iT1i$3$E:
                000D042C 09193078           ADD.L1X       A9,B6,A18
                000D0430 0814F078           ADD.L1X       A7,B5,A16

                //如果直接用(colu+shift_size)%zz 會出現負值，用這行就一定是正值
                while(temp%zz<0)     temp=temp + zz;
                000D0434 000F5C11           B.S1          _remi
                000D0438 02400FD8  ||       OR.L1         0,A16,A4
                000D043C 02000C2A           MVK.S2        0x0018,B4
                000D0440 01816162           ADDKPC.S2     RL0,B3,3
                000D0444          RL0:
                000D0444 001008D8           CMPGT.L1      0,A4,A0
                000D0448 D00EA120     [!A0] BNOP.S1       DW$L$_circular_shift__FPA24_iT1i$8$E,5
                000D044C          DW$L$_circular_shift__FPA24_iT1i$5$E:
                000D044C 000F5813           B.S2          _remi
```

Figure 5.10: The assembly codes of circular shift (1/2).

```
000D0450 08000C50  ||          ADDK.S1       24,A16
000D0454 00004000              NOP           3
000D0458              DW$L$_circular_shift__FPA24_iT1i$6$E:
000D0458 01890162              ADDKPC.S2     RL1,B3,0
000D045C 02000C28              MVK.S2        0x0018,B4
000D0460 02400FD8  ||          OR.L1         0,A16,A4
000D0464              RL1:
000D0464 001008D8              CMPGT.L1      0,A4,A0
000D0468 CFFE2123     [ A0]    BNOP.S2       DW$L$_circular_shift__FPA24_iT1i$6$E,1
000D046C C8000C50  || [ A0]    ADDK.S1       24,A16
000D0470 C00F5410     [ A0]    B.S1          _remi
000D0474 00004000              NOP           3

while(temp >= zz)   temp = temp-zz;
000D0478              DW$L$_circular_shift__FPA24_iT1i$8$E:
000D0478 01800C28              MVK.S1        0x0018,A3
000D047C 000E0AF8              CMPLT.L1      A16,A3,A0
000D0480 C01EA120     [ A0]    BNOP.S1       DW$L$_circular_shift__FPA24_iT1i$15$B,5
000D0484              DW$L$_circular_shift__FPA24_iT1i$9$E:
000D0484 02400FD9              OR.L1         0,A16,A4
000D0488 040403E2  ||          MVC.S2        CSR,B8
000D048C 04A3CF5B              AND.L2        -2,B8,B9
000D0490 021309C0  ||          SUB.D1        A4,0x18,A4
000D0494 000C8BF9              CMPLTU.L1     A4,A3,A0
000D0498 021309C1  ||          SUB.D1        A4,0x18,A4
000D049C 00A403A2  ||          MVC.S2        B9,CSR
000D04A0              DW$L$_circular_shift__FPA24_iT1i$10$E:
000D04A0 D0000813     [!A0]    B.S2          L6
000D04A4 021309C1  ||          SUB.D1        A4,0x18,A4
000D04A8 D00C8BF8  || [!A0]    CMPLTU.L1     A4,A3,A0
000D04AC D0000813     [!A0]    B.S2          L6
000D04B0 021309C1  ||          SUB.D1        A4,0x18,A4
000D04B4 D00C8BF8  || [!A0]    CMPLTU.L1     A4,A3,A0
000D04B8 D0000813     [!A0]    B.S2          L6
000D04BC 021309C1  ||          SUB.D1        A4,0x18,A4
000D04C0 D00C8BF8  || [!A0]    CMPLTU.L1     A4,A3,A0
000D04C4 D0000413     [!A0]    B.S2          L6
000D04C8 021309C1  ||          SUB.D1        A4,0x18,A4
000D04CC D00C8BF8  || [!A0]    CMPLTU.L1     A4,A3,A0
000D04D0 02401FDB              OR.L2X        0,A16,B4
000D04D4 D0000413  || [!A0]    B.S2          L6
000D04D8 021309C1  ||          SUB.D1        A4,0x18,A4
000D04DC D00C8BF8  || [!A0]    CMPLTU.L1     A4,A3,A0
000D04E0              L6:
000D04E0 021309C3              SUB.D2        B4,0x18,B4
000D04E4 D0000013  || [!A0]    B.S2          L6
000D04E8 D00C8BF9  || [!A0]    CMPLTU.L1     A4,A3,A0
000D04EC 021309C0  ||          SUB.D1        A4,0x18,A4
000D04F0              DW$L$_circular_shift__FPA24_iT1i$14$B:
000D04F0 00A003A2              MVC.S2        B8,CSR
000D04F4 08101FD8              OR.L1X        0,B4,A16

 HH[row][temp]=I[row][colu];
 000D04F8              DW$L$_circular_shift__FPA24_iT1i$15$B:
 000D04F8 000D4121              BNOP.S1       DW$L$_circular_shift__FPA24_iT1i$17$B,2
 000D04FC 01C80264  ||          LDW.D1T1      *+A18[0x0],A3
 000D0500 02461C40              ADDAW.D1      A17,A16,A4
 000D0504 02110078              ADD.L1        A8,A4,A4
 000D0508 01900274              STW.D1T1      A3,*+A4[0x0]
      }
    }
  }
000D053C              DW$L$_circular_shift__FPA24_iT1i$18$E:
000D053C 01BC52E6              LDW.D2T2      *++SP[0x2],B3
000D0540 00006000              NOP           4
000D0544 008CA362              BNOP.S2       B3,5
```

Figure 5.11: The assembly codes of circular shift (2/2).

```
for (i=0; i<N; i++)
  for (j=0; j<code_node_size[i]; j++)
    {

    code_node_pi1[i][j] = 0;

    for (l=0; l<code_node_size[i]; l++)
      {
      aux = code_node_index[i][l]-1;

      if ( aux != (code_node_index[i][j]-1) )
        {

        // Compute index "m" of message from children
        m = 0;
        while (  ( (check_node_index[aux][m]-1) != i )
                      && ( m < check_node_size[aux])  ) m++;

        code_node_pi1[i][j] += check_node_lambda1[aux][m];
        }
      }

    code_node_pi1[i][j] += llrp1[i];


    if (code_node_pi1[i][j] < -30*offset)
      code_node_pi1[i][j] = -30*offset;

    }
```

Figure 5.12: The C code of computing form check nodes to bit nodes.

- 288*$N(m)$*$N(m)$.
- 576*$M(n)$*$M(n)$.
- 576*$M(n)$.

In Fig. 5.12, we show the C code which computing the value from check nodes to bit nodes. In the code. There are 576*$M(n)$*$M(n)$ loops. 576 is the number of bit nodes and 288 is the numbers of check node.$M(n)$ means the number of check nodes connected to bit node n and $N(m)$ denotes the number of bit nodes connected to the check nodes. The loop in one iteration is depended on $M(n)$ and $N(m)$. However, in coding rate 1/2, $M(n)$ is more than 6 and $N(m)$ is more than 3. It means that there is more than $288*6*6+576*3*3+576*3 = 17280$ loops in one iteration. In one loop, it should execute the bit node value or check node value, read value from memory, and a little stall or NOP cycle. We approximate 90 cycles in one loop and it need about $90*17280 = 1555200$ cycles in one iteration. So, it needs about

Table 5.8: Final Profile of LDPC Code (Code Size).

	Code Size (byte)
Encoder	6028
Decoder	2688

31104000 cycles after 20 iterations.

In Fig. 5.13, 5.14 and 5.15, we see the assembly code which computing the value from check nodes to bit nodes. The parallelism is not good. We see that in our code, the value in check nodes and bits nodes are read from memory many times. Reading memory costs several cycles and reduces the code parallelism.

In Table 5.8, we just show the code size of encoder and decoder without randomizer, interleaver and modulator.

```
for (i=0; i<N; i++)
000D0A38            DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$40$E:
000D0A38 097FF22B                   MVK.S2        0xffffffe4,B18
000D0A3C 0D440029  ||               MVK.S1        0xffff8800,A26
000D0A40 0900A358  ||               MVK.L1        0,A18
000D0C48            DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$55$E:
000D0C48 0C608059                   ADD.L1        4,A24,A24
000D0C4C 01848029  ||               MVK.S1        0x0900,A3
000D0C50 09482940  ||               ADD.D1        A18,0x1,A18
000D0C54 000F0AF8                   CMPLT.L1      A24,A3,A0
000D0C58 CF818121      [ A0]  BNOP.S1       DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_
000D0C5C D204842B  || [!A0]  MVK.S2        0x0908,B4
000D0C60 D4BD005A  || [!A0]  ADD.L2        8,SP,B9
000D0C64 D43C807A     [!A0]  ADD.L2        B4,SP,B8
  for (j=0; j<code_node_size[i]; j++)
  000D0A44            DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$41$E:
  000D0A44 018484EC                   LDW.D2T1      *+SP[0x484],A3
  000D0A48 00004000                   NOP           3
  000D0A4C 0F8486EC                   LDW.D2T1      *+SP[0x486],A31
  000D0A50 018E4A64                   LDW.D1T1      *+A3[A18],A3
  000D0A54 00006000                   NOP           4
  000D0A58 01800C29                   MVK.S1        0x0018,A3
  000D0A5C 000C0AD8  ||               CMPLT.L1      0,A3,A0
  000D0A60 D07A6121      [!A0]  BNOP.S1       DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA
  000D0A64 0B486571  ||               MPYLI.M1      A3,A18,A23:A22
  000D0A68 C23F1078  || [ A0]  ADD.L1X       A24,SP,A4
  000D0A6C 01DBE078                   ADD.L1        A31,A22,A3
  000D0A70 00000000                   NOP
  000D0A74            DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$42$E:
  000D0A74 0C910059                   ADD.L1        8,A4,A25
  000D0A78 0BD806A1  ||               OR.S1         0,A22,A23
  000D0A7C 040F905A  ||               SUB.L2X       A3,4,B8
  000D0A80 0880A35A                   MVK.L2        0,B17
  000D0C28 0B588058                   ADD.L1        4,A22,A22
  000D0C2C 018484ED                   LDW.D2T1      *+SP[0x484],A3
  000D0C30 08C4205A  ||               ADD.L2        1,B17,B17
  000D0C34 00006000                   NOP           4
  000D0C38 018E4A64                   LDW.D1T1      *+A3[A18],A3
  000D0C3C 00006000                   NOP           4
  000D0C40 000E3AFA                   CMPLT.L2X     B17,A3,B0
  000D0C44 2F91A120      [ B0]  BNOP.S1       DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA
    {

    code_node_pil[i][j] = 0;
    000D0A84            DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$43$E:
    000D0A84 0200A35A                   MVK.L2        0,B4
    000D0A88 022032F6                   STW.D2T2      B4,*++B8[0x1]

    for (l=0; l<code_node_size[i]; l++)
    000D0A8C 018484EC                   LDW.D2T1      *+SP[0x484],A3
    000D0A90 020485EC                   LDW.D2T1      *+SP[0x485],A4
    000D0A94 00004000                   NOP           3
    000D0A98 018E4A64                   LDW.D1T1      *+A3[A18],A3
    000D0A9C 00000000                   NOP
    000D0AA0 0292E078                   ADD.L1        A23,A4,A5
    000D0AA4 00000000                   NOP
    000D0AA8 0817905A                   SUB.L2X       A5,4,B16
    000D0AAC 000C0AD8                   CMPLT.L1      0,A3,A0
    000D0AB0 D05BA121      [!A0]  BNOP.S1       DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2
    000D0AB4 CD92C078  || [ A0]  ADD.L1        A22,A4,A27
    000D0AB8            DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$44$E:
    000D0AB8 0480A35B                   MVK.L2        0,B9
```

Figure 5.13: The assembly code of computing form check nodes to bit nodes (1/3).

```
000D0ABC 0E00A358  ||        MVK.L1       0,A28
000D0C00         DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$53$E:
000D0C00 0E702058            ADD.L1       1,A28,A28
000D0C04 007068F8            CMPGT.L1     A3,A28,A0
000D0C08 CFB0A120     [ A0]  BNOP.S1      DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2
  {
  aux = code_node_index[i][1]-1;
  000D0AC0         DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$45$E:
  000D0AC0 024032E6            LDW.D2T2     *++B16[0x1],B4
  000D0AC4 00006000            NOP          4
  000D0AC8 0313E05A            SUB.L2       B4,1,B6

  if ( aux != (code_node_index[i][j]-1) )
  000D0ACC 026C0264            LDW.D1T1     *+A27[0x0],A4
  000D0AD0 00006000            NOP          4
  000D0AD4 00109A78            CMPEQ.L1X    A4,B4,A0
  000D0AD8 C050A120     [ A0]  BNOP.S1      DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_i
    {

    // Compute index "m" of message from children
    m = 0;
    000D0ADC         DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$46$E:
    000D0ADC 0380A35A            MVK.L2       0,B7
    while (  ( (check_node_index[aux][m]-1) != i )
    000D0AE0 0290ACA2            SHL.S2       B4,0x5,B5
    000D0AE4 02149CC2            SUBAW.D2     B5,B4,B4
    000D0AE8 0212407A            ADD.L2       B18,B4,B4
    000D0AEC 0211807A            ADD.L2       B12,B4,B4
    000D0AF0 021002E6            LDW.D2T2     *+B4[0x0],B4
    000D0AF4 00006000            NOP          4
    000D0AF8 01C892F8            SUB.L1X      B4,A18,A3
    000D0AFC 000C2A58            CMPEQ.L1     1,A3,A0
    000D0B00 C0358121     [ A0]  BNOP.S1      DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6
    000D0B04 D8800041  || [!A0]  MVK.D1       0,A17
    000D0B08 D9981FD9  || [!A0]  OR.L1X       0,B6,A19
    000D0B0C D2ACCAE6  || [!A0]  LDW.D2T2     *+B11[B6],B5
    000D0B10 0094EAFA            CMPLT.L2     B7,B5,B1
    000D0B14         DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$47$E:
    000D0B14 0800A359            MVK.L1       0,A16
    000D0B18 031C18F1  ||        OR.D1X       0,B7,A6
    000D0B1C 48CCACA1  || [ B1]  SHL.S1       A19,0x5,A17
    000D0B20 020491EC  ||        LDW.D2T1     *+SP[0x491],A4
    000D0B24 431C3059     [ B1]  ADD.L1X      1,B7,A6
    000D0B28 48467CC1  || [ B1]  SUBAW.D1     A17,A19,A16
    000D0B2C 04800028  ||        MVK.S1       0x0000,A9
    000D0B30 44C0DC41     [ B1]  ADDAW.D1     A16,A6,A9
    000D0B34 0400A358  ||        MVK.L1       0,A8
    000D0B38 44313078     [ B1]  ADD.L1X      A9,B12,A8
    000D0B3C         DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$48$E:
    000D0B3C 0380A359            MVK.L1       0,A7
    000D0B40 43A00264  || [ B1]  LDW.D1T1     *+A8[0x0],A7
    000D0B44 00000000            NOP
    000D0B48 038403E2            MVC.S2       CSR,B7
    000D0B4C 0004A35B            MVK.L2       1,B0
    000D0B50 0A301FD9  ||        OR.L1X       0,B12,A20
    000D0B54 031FC7A2  ||        AND.S2       -2,B7,B6
    000D0B58 0280A359            MVK.L1       0,A5
    000D0B5C 0104A35B  ||        MVK.L2       1,B2
    000D0B60 0A800029  ||        MVK.S1       0x0000,A21
    000D0B64 019808F1  ||        OR.D1        0,A6,A3
    000D0B68 021818F3  ||        OR.D2X       0,A6,B4
    000D0B6C 009803A2  ||        MVC.S2       B6,CSR
```

Figure 5.14: The assembly code of computing form check nodes to bit nodes (2/3).

```
    000D0B70          L25:
    000D0B70 03040A5B               CMPEQ.L2      0,B1,B6
    000D0B74 42C8E0F8  || [ B1]  SUB.L1        A7,A18,A5
    000D0B78 42142A59     [ B1]  CMPEQ.L1      1,A5,A4
    000D0B7C 008CB8FA  ||         CMPGT.L2X     B5,A3,B1
    000D0B80 2A900FD9     [ B0]  OR.L1         0,A4,A21
    000D0B84 48CCACA0  || [ B1]  SHL.S1        A19,0x5,A17
    000D0B88 0010DFFB               OR.L2X        B6,A4,B0
    000D0B8C 43182059  || [ B1]  ADD.L1        1,A6,A6
    000D0B90 48467CC0  || [ B1]  SUBAW.D1      A17,A19,A16
    000D0B94 2100A35B     [ B0]  MVK.L2        0,B2
    000D0B98 44C0DC40  || [ B1]  ADDAW.D1      A16,A6,A9
    000D0B9C 6FFFFE13     [ B2]  B.S2          L25
    000D0BA0 62181FDB  || [ B2]  OR.L2X        0,A6,B4
    000D0BA4 44268079  || [ B1]  ADD.L1        A20,A9,A8
    000D0BA8 70800042  || [!B2]  MVK.D2        0,B1
    000D0BAC 43A00264     [ B1]  LDW.D1T1      *+A8[0x0],A7
    000D0BB0 00002000               NOP           2
    000D0BB4 00080FDB               OR.L2         0,B2,B0
    000D0BB8 01980FD8  ||         OR.L1         0,A6,A3
    000D0BBC 00000000               NOP
    000D0BC0          DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$50$E:
    000D0BC0 034C1FDB               OR.L2X        0,A19,B6
    000D0BC4 009C03A2  ||         MVC.S2        B7,CSR
    000D0BC8 03900FDB               OR.L2         0,B4,B7
    000D0BCC 0A8491FD  ||         STW.D2T1      A21,*+SP[0x491]
    000D0BD0 065016A2  ||         OR.S2X        0,A20,B12
              && ( m < check_node_size[aux])  ) m++;

    code_node_pi1[i][j] += check_node_lambda1[aux][m];
    000D0BD4          DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$52$E:
    000D0BD4 0218ACA2               SHL.S2        B6,0x5,B4
    000D0BD8 0210DCC2               SUBAW.D2      B4,B6,B4
    000D0BDC 0210FC42               ADDAW.D2      B4,B7,B4
    000D0BE0 018484EC               LDW.D2T1      *+SP[0x484],A3
    000D0BE4 0211A07A               ADD.L2        B13,B4,B4
    000D0BE8 021002E6               LDW.D2T2      *+B4[0x0],B4
    000D0BEC 00002000               NOP           2
    000D0BF0 018E4A64               LDW.D1T1      *+A3[A18],A3
    000D0BF4 00000000               NOP
    000D0BF8 04A4807A               ADD.L2        B4,B9,B9
    000D0BFC 04A002F6               STW.D2T2      B9,*+B8[0x0]
    }
  }

code_node_pi1[i][j] += llrp1[i];
000D0C0C          DW$L$_LDPC_Decoder__FiPiN21fT5PsT7T2PA6_iPA6_iPA6_iT2PA7_iPA7_iPA7_i$54$E:
000D0C0C 022002E7               LDW.D2T2      *+B8[0x0],B4
000D0C10 01E40264  ||         LDW.D1T1      *+A25[0x0],A3
000D0C14 00006000               NOP           4
000D0C18 01907078               ADD.L1X       A3,B4,A3


if (code_node_pi1[i][j] < -30*offset)
000D0C1C 01A002F5               STW.D2T1      A3,*+B8[0x0]
000D0C20 00686AF8  ||         CMPLT.L1      A3,A26,A0
  code_node_pi1[i][j] = -30*offset;
  000D0C24 CD2002F4     [ A0]  STW.D2T1      A26,*+B8[0x0]

}
```

Figure 5.15: The assembly code of computing form check nodes to bit nodes (3/3).

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 167
;*      Loop opening brace source line   : 168
;*      Loop closing brace source line   : 168
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 11
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 4
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     1        2
;*      .S units                     1        1
;*      .D units                     3        0
;*      .M units                     0        0
;*      .X cross paths               0        3
;*      .T address paths             1        0
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          6        8     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        2
;*      Bound(.L .S .D .LS .LSD)     4*       4*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 11 Schedule found with 2 iterations in parallel
```

Figure 5.16: Software pipeline information for LDPC decoder.

Chapter 6

Conclusion and Future Work

This work contained two parts of IEEE 802.16e. One was the research in convolution code and implementation on DSP of 802.16e for WirelessMAN-OFDMA. And the other was the reduced-complexity decoding research of the LDPC code.

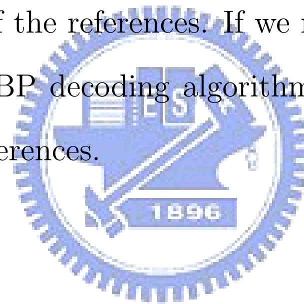
In the first part work, we first analyzed the Shannon bounds on coding gain and theoretic coding gain based on minimum codeword distance in AWGN. In our implementation, the convolution coding gain in AWGN was less than theoretic value by 1 dB. When we converted the floating-point to fixed-point, the performance was almost the same and we could just use 6 bits to implement the decoder. Finally, the convolutional decoder required multiple DSPs to run in parallel to handle the data rate under a 10 MHz transmission bandwidth. But encoder could achieve the data rate 10 Mhz.

In the second part work, we first evaluated the performance of LDPC code and compared the results with the numerical results. The coding gain of LDPC code was much better than convolution code. Then we focused on several complexity-reducing decoding algorithm. Therefore, these simplified reduced-complexity decoding schemes could outperform the BP decoding algorithm. Then we converted the floating-point to fixed-point, and we could use 6 bits to implement the decoder. In the DSP implementation, it could not achieve

the bandwidth 10 MHz both in encoder and decoder. LDPC code is more complex than convolution code in our DSP implementation.

In the future work, we need to revise the coding algorithms to be fixed-point to reduce the complexity for actual DSP implementation. In convolution code, the C64x is equipped with a Viterbi decoder co-processor [30]. Using this co-processor may be helpful in raising the decoding speed. But it use requires study and testing of the "enhanced direct memory access (EDMA)" mechanism of the C64x chips.

In LDPC code, there are two possible methods to enhance our DSP implementation. First, we may rewrite our code. We discuss in chapter 5, in our code, there are too many loops to execute. These cost too many cycles and must read the memory many times. Second, we have find some references. If we need further reducing complexity by other decoding algorithms, [34] 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, [35] is one of the references.



Bibliography

[1] IEEE Std 802.16e, *IEEE Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Fixed Broadband Wireless Access Systems. Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands and Corrigendum 1.* New York: IEEE, Feb. 2006.

[2] Y.-P. Ho, "Study on OFDM Signal Description and Channel Coding in the IEEE 802.16a TDD OFDMA Wireless Communication Standard," M.S. thesis, National Chiao Tung University, Dep. of Electronics Eng., Hsinchu, Taiwan, R.O.C., June 2003.

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

[4] H. H. Ma and J. K. Wolf, "On tail biting convolutional codes," *IEEE Trans. Commun.*, vol. 34, pp. 104–111, 1986.

[5] Y.-P. E. Wang and R. Ramésh, "To bite or not to bite — a study of tail bits versus tail-biting," in *Proc. IEEE Int. Symp. Personal Indoor Mobile Radio Commun.*, vol. 2, Oct. 1996, pp. 317–321.

[6] W. Sung and I.-K. Kim, "Performance of a fixed delay decoding scheme for tail biting convolutional codes," in *IEEE Asilomar Signals Sys. Computers Conf. Rec.*, vol. 1, Oct. 1996, pp. 704–708.

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

[8] M. Speth, A. Senst, and H. Meyr, "Low complexity space-frequency MLSE for multi-user COFDM," in *IEEE Global Telecommun. Conf. Rec.*, vol. 5, 1999, pp. 2395–2399.

[9] F. Tosato and P. Bisaglia, "Simplified soft-output demapper for binary interleaved COFDM with application to HIPERLAN/2," in *IEEE Int. Conf. Commun. Conf. Rec.*, vol. 2, 2002, pp. 664–668.

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

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

[12] 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.

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

[14] 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 Communication Letter*, vol. 5, no. 2, pp. 58–60, Feb. 2001.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Y.-C. Chen, "Research in Channel Coding Techniques and DSP Implementation for IEEE 802.16e OFDM and OFDMA," M.S. thesis, National Chiao Tung University, Dep. of Electronics Eng., Hsinchu, Taiwan, R.O.C., June 2006.

[21] *Quixote Data Sheet.* http://www.innovative-dsp.com/support/datasheets/quixote.pdf.

[22] 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.

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

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

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

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

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

[28] Texas Instruments, *TMS320C6000 Optimizing Compiler User's Guide.* Literature number SPRU187L, May. 2004.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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：吳柏昇 (Po-Sheng Wu)

生日：1982 年 12 月 21 日

出生地：台中縣

學歷：交通大學電信工程系學士(2001.9~2005.6)

交通大學電子研究所碩士(2005.9~2007.6)

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 OFDMA 通道編碼技術與數位訊號處理器實現之研究

(Research in and DSP Implementation of Channel Coding Techniques for IEEE 802.16e OFDMA)