國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

WiMAX通道編碼技術與

數位訊號處理器實現之探討

Study in WiMAX Channel Coding Techniques and Associated Digital Signal Processor Implementation

研 究 生：陳佳楓

指導教授：林大衛 博士

中 華 民 國 九 十 七 年 六 月


E S A
1896

WiMAX 通道編碼技術與

數位訊號處理器實現之探討

Study in WiMAX Channel Coding Techniques and

Associated Digital Signal Processor Implementation

研究生：陳佳楓　　Student: Jia-Fong Chen

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

國 立 交 通 大 學

電子工程學系　電子研究所碩士班

碩士論文

A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Electronics Engineering

June 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年六月


E S A
1896

WiMAX 通道編碼技術與

數位信號處理器實現之探討

研究生:陳佳楓　　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(咬尾)系統並且實現在德州儀器公司所發展數位訊號處理器(DSP) TMS320C6416 上的維特比解碼協處理器(VCP)並針對咬尾編碼的特性，中斷服務常式(ISR)以及增強型直接記憶體存取(EDMA)進行研究。此外我們也利用 3L Diamond 的 EDMA 進行 VCP 在多個 DSP 運算處理的應用。在論文中，我們利用 C 語言所模擬的迴旋碼在加成性白色高斯通道下和利用 VCP 應用於迴旋碼進行效能及速度上的比較。在效能錯誤率上，受限於實點數及 VCP 輸入位元數的硬體條件下，若以相同條件比較而言，兩者的效能是接近的。而在速度方面，經過在 DSP 平台上最佳化我們的程式後，分別於 CCS 模擬器和 3L 測量上，迴旋編碼的編碼器部份，可以到每秒 16,667K 和 3,764K 位元的處理速度，而在 VCP 方面解碼器的部份可以達到每秒 7,897K 和 2,997K 位元的處理速度，C 語言模擬方面則可以達到每秒 805K 和 632K 位元的處理速度。簡而說之，若以解碼器觀點而言，VCP 提升了速度為 9.8 和 4.7 倍，分別針對 CCS 模擬器和 3L Diamond 測量而得到數據。

本論文後半部份重點，研究 IEEE 802.16e OFDMA 所訂定的渦輪迴旋碼(CTC)系統並且實現在數位訊號處理器。闡明渦輪迴旋碼的雙二位元循環遞迴系統迴旋(duo-binary CRSC)編碼與最大對數事後機率(max-log MAP)解碼演算法。我們利用 C 語言驗證系統演算法上的正確性，並在加成性白色高斯通道下模擬了各種調變。接著在 TI C6416 DSP 平台實現，於 3L Diamond 測量上方面，編碼器部份可以到每秒 8,223 位元的處理速度，而解碼器的部份僅可以達到每秒 30K 位元的處理速度。之後我們對於解碼器做了一些最佳化的改善，使解碼器的速度增進約 10 倍，進而可以達到每秒 300K 位元的處理速度。

Study in WiMAX Channel Coding Techniques and Associated Digital Signal Processor Implementation

Student: Jia-Fong Chen Advisor: Dr. David W. Lin

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

Abstract

In the IEEE 802.16e wireless communication standard, a *f*orward *e*rror *c*orrection (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 (CC) with tail biting defined in IEEE 802.16e OFDMA standard and implement the project on Viterbi-decoder coprocessor (VCP) of the Texas Instruments (TI)'s TMS320C6416T digital signal processor (DSP) and also sturdy for tail-biting encoding property, interrupt service routine (ISR) and enhanced direct memory access (EDMA). Besides, we also employ the EDMA under 3L Diamond real-time operating system (RTOS) for the VCP applications of multi-DSP operation. We compare CC in AWGN channel on the C program to CC on the VCP applications for BER performance and processing rate. In BER performance, the simulation is limited to the hardware fixed-point and VCP branch metric input bit numbers; however, if we utilize the same condition to compare them, we can find their performance are close. In processing rate, after optimizing the programs on the DSP platform, encoder can achieve two data processing rates of 16,667 Kbps and 3,764 Kbps, the VCP decoder can achieve two processing rates of 7,897 Kbps and 2,997 Kbps and the C program decoder can achieve two processing rates of 805 Kbps and 632 Kbps, respectively on the C6416 CCS simulator and 3L Diamond. In short, we utilize the CCS and 3L to measure, finding decoding processing rate can be improve significantly about 9.8 and 4.7 times, respectively.

The focus of second part is the research of the convolutional turbo code (CTC) defined in IEEE 802.16e OFDMA and implement on the C6416 DSP. We explain the duo-binary circular recursive systematic convolutional encoding (duo-binary CRSC) and the max-log MAP decoding algorithm. We employ the C program to insure the correctness of our

algorithm and simulate the CTC for different modulation in AWGN; then, we implement on TI C6416 DSP. The encoder can achieve a data processing rate of 8,223 Kbps and the decoder can achieve a processing rate of 30 Kbps on the 3L. Then we utilize some optimized techniques to improve the decoder's speed, which is approximately 10 times speeded up in decoding rate. Therefore, the decoder can achieve a further data processing rate of 300 Kbps.


E S A
1896

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。謝謝你們熱心有耐性地幫我解決了許多通訊方面相關的疑問。也要感謝 3L Diamond Engineer Peter Robertson，謝謝你不嫌棄我的英文，透過不斷往來的英文書信，仍然很熱心很積極的幫我解決一些 3L EDMA 的問題。另外還要感謝「數位信號處理平台在嵌入式系統的應用」一書作者，交大電信所畢業的盧怡仁學長，學長已經畢業學校很多年，因緣巧合和學長搭上了線，感謝您幫我釐清一些 DSP 問題的觀念。

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 94 級柏昇、順成兩位學長的指導，以及 95 級昀澤、光中、婉清、奕安、威年、尚諭、衛川、紹唐等實驗室成員，平日和我一起唸書，一起討論，也一起打混，讓我的研究生涯充滿歡樂又有所成長。期待大家畢業之後都能有不錯的發展。

最後，要感謝的是我的家人，他們的支持讓我能夠心無旁騖的從事研究工作。

謝謝所有幫助過我、陪我走過這一段歲月的師長、同儕與家人。謝謝！

誌於 2008.6 故鄉風城 交大

佳楓


E
S
A
1896

Contents

1 Introduction **1**

1.1 Scope of the Work 1

1.2 Organization of This Thesis 3

2 Overview of CC and CTCs in IEEE 802.16e OFDMA **4**

2.1 Tail-Biting Convolutional Code Specifications [1] 4

2.1.1 Randomizer [1] 5

2.1.2 Convolutional Encoder [1] 7

2.1.3 Interleaver [1] 10

2.1.4 Modulation [1] 11

2.2 Decoding of CC 11

2.2.1 Demodulation for Bit-Interleaved Coded Modulation [9] 12

2.3 Convolutional Turbo Codes Specifications [1] 16

2.3.1 CTC Encoder in IEEE 802.16e OFDMA [1] 17

2.3.2 CTC Interleaver [1] 20

2.3.3 CTC Tail-Biting [1], [10] 20

2.3.4 Subpacket Generation (Channel Interleaver or Interleaver and Puncturing) [1] . . . 23

2.4 Decoding of CTC . . . 29

2.4.1 The Turbo Decoding Algorithm [11] . . . 29

2.4.2 Decoding Rule for CRSC Codes with Non-binary Trellis [14] . . . 31

2.4.3 Simplified Max-Log-MAP Algorithm for Double-Binary CTC [14] . . . 33

3 DSP Implementation Environment **37**

3.1 The DSP Baseboard . . . 37

3.2 The Viterbi-Decoder Coprocessor (VCP) [19] . . . 38

3.2.1 Overview of VCP [19], [21], [22] . . . 38

3.2.2 VCP Inputs (Brach Metrics and VCP Input Configuration) [19], [22] 43

3.2.3 VCP Output (Decisions) [19] . . . 46

3.2.4 Sliding Windows Processing [19] . . . 48

3.3 VCP Programming [19], [21] . . . 49

3.3.1 Prepare Input Configuration, Initialize Input Buffers, and Allocate Output Buffers [21] . . . 51

3.3.2 EDMA Resource [19] . . . 52

3.3.3 VCP Procedure [21] . . . 52

3.4 EDMA under the Code Composer Studio (CCS) [23] . . . 54

3.4.1 EDMA Control Registers [23] . . . 55

3.4.2 Parameter RAM (PaRAM) [23] . . . 58

3.4.3 EDMA Transfer Parameter Entry [23] 60
3.4.4 Initiating an EDMA Transfer [23] 60
3.4.5 Linking EDMA Transfers [23] 61
3.4.6 EDMA Interrupt Generation [23], [24] 62
3.5 EDMA under the 3L Diamond Real-Time Operating System 64
3.5.1 Introduction to 3L Diamond 64
3.5.2 SC6xEDMA [26] 64
3.5.3 EDMA Channel Availability [26] 67
3.5.4 SC6xEDMAChannel Functions [26] 67

4 DSP Implementation of Convolutional Encoder and Decoder 70
4.1 VCP Parameter Setting 70
4.1.1 Generator Polynomials 71
4.1.2 EDMA Setting 71
4.1.3 Tail-Biting 72
4.2 Coding Gain Analysis [3] 73
4.3 Comparison of Performance in AWGN of VCP and Wu's Viterbi Decoder . . 75
4.4 VCP Operation Under 3L Diamond 84

5 Simulation and DSP Implementation of CTC Encoder and Decoder 89
5.1 Performance in AWGN Channel with Floating-Point Processing 89
5.2 Performance in AWGN Channel with Fixed-Point Processing 91

5.2.1 Speed Performance of the DSP Code 97
5.2.2 Improving CTC Decoding Speed 101
5.3 Comparison of Speed of Current Codes 113
5.3.1 The Views of Block Decoder for Processing Rate 113
5.3.2 Comparison of Tail-Biting CC and CTC for Adders and Multipliers . 114
5.3.3 Comparison of Decoder Speed for Tail-Biting CC, CTC, and LDPC . 116

6 Conclusion and Future Work **118**

Bibliography **120**

List of Figures

2.1 Structure of convolutional coding in transmitter (top path) and decoding in receiver (bottom path). 5

2.2 PRBS for data randomization (from [1]). 7

2.3 Convolutional encoder of rate 1/2 (from [1]). 8

2.4 The second permutation of interleaver. 11

2.5 QPSK, 16-QAM, and 64-QAM constellations (from [1]). 12

2.6 Metric partitions of the 16-QAM constellation (from [9]). 15

2.7 CTCs coding block diagram (from [1]). 16

2.8 CTC encoder (modified from [1]). 18

2.9 CTC rate 1/3 encoder flow chart. 19

2.10 CTC encoding slot concatenation for different rate (modified from [1]). . . . 20

2.11 CTC channel coding per modulation (modified from [1]). 21

2.12 CTC interleaver in two steps (modified from [1]). 22

2.13 Block diagram of CTC channel interleaving scheme (from [1]). 26

2.14 Block diagram of a turbo decoder (from [11]). 29

2.15 CTC trellis structure of duo-binary convolutional code with feedback encoder (from [14]). 31

3.1 Sundance's SMT395 module (from [18]). 38

3.2 VCP block diagram (modified from [19]). 39

3.3 DSP chip architecture (from [20]). 40

3.4 DSP chip die (from [20]). 40

3.5 Convolutional encoder example, where $K = 3$, $R = 1/3$, $G0 = (100)_8$, $G1 = (101)_8$, $G2 = (111)_8$ (from [19]). 42

3.6 Convolutional code trellis example (from [19]). 43

3.7 Example of survivor path and associated decoded sequence (from [21]). . . . 44

3.8 VCP input FIFO (modified from [19]). 45

3.9 VCP registers (modified from [19]). 46

3.10 VCP configuration structure (modified from [22]). 47

3.11 VCP output FIFO (modified from [19]). 47

3.12 VCP tailed traceback mode (from [19]). 49

3.13 VCP frame, reliability, and convergence length limitations(modified from [19]). 50

3.14 VCP EDMA parameters structure (from [19]). 53

3.15 EDMA control (from [23]). 55

3.16 EDMA parameter RAM contents (modified from [23]). 59

3.17 EDMA channel parameters (from [23]). 60

3.18 Example of linked EDMA transfers (from [23]). 62

4.1 CC encoding and decoding with VCP. 71
4.2 VCP parameter setting (modified from [21]). 72
4.3 Tail-biting CC decoding employing (modified from [3]). 73
4.4 VCP decoding performance in AWGN with different BM truncation precisions (1/3). 78
4.5 VCP decoding performance in AWGN with different BM truncation precisions (2/3). 79
4.6 VCP decoding performance in AWGN with different BM truncation precisions (3/3). 80
4.7 Effect of CSB values in VCP-based decoding in AWGN at different coding-modulation settings (1/3). 81
4.8 Effect of CSB values in VCP-based decoding in AWGN at different coding-modulation settings (2/3). 82
4.9 Effect of CSB values in VCP-based decoding in AWGN at different coding-modulation settings (3/3). 83
4.10 How VCP operates under 3L Diamond and TI CCS. 84

5.1 Performance of CTC at different iteration counts under different modulations. 90
5.2 CTC decoding performance with different modulations employing floating-point computation at 4 iterations. 91
5.3 CTC fixed-point truncation parameters. 93
5.4 CTC fixed-point truncation parameters flow chart. 93
5.5 CTC at different bit numbers with different modulations. 94

5.6 Performance with scaling of various quantities in CTC decoding to avoid overflow at high SNR. 95

5.7 BER performance of CTC decoding with fixed-point computation vs. floating-point computation. 96

5.8 Function *Gamma*() (1/3). 103

5.9 Function *Gamma*() (2/3). 104

5.10 Function *Gamma*() (3/3). 105

5.11 The assembly code of *Gamma*() function (1/6). 106

5.12 The assembly code of *Gamma*() function (2/6). 107

5.13 The assembly code of *Gamma*() function (3/6). 108

5.14 The assembly code of *Gamma*() function (4/6). 109

5.15 The assembly code of *Gamma*() function (5/6). 110

5.16 The assembly code of *Gamma*() function (6/6). 111

List of Tables

2.1 Mandatory Channel Coding Schemes for Each Modulation Method 6

2.2 The Convolutional Code with Puncturing Configuration 8

2.3 Bit Interleaved Block Sizes and Modulos 10

2.4 Bit Metric for Method-ML and Method-LLR 14

2.5 Circulation State Look-Up Table (S_{C1} and S_{C2}) 23

2.6 Parameters for the Subblock Interleavers 26

3.1 Branch Metrics for Rate-1/2 Code 44

3.2 VCP Required EDMA Links Per User Channel 50

3.3 EDMA Control Registers (Modified from [23]) 56

4.1 Coding Gain Upper Bounds in AWGN at BER = 10^{-6} 75

4.2 Approximate Coding Gains Based on Analysis of Minimum Codeword Distance 76

4.3 Comparison of Soft-Decision Decoding Performance, in AWGN at BER = 10^{-6} 77

4.4 Speed of Overall Decoder from 3L-Measured Execution Time 85

4.5 CCS Profile of CC Coding and Decoding with VCP (Cycles) 86

4.6 3L-Measured Execution Time of CC Coding and Decoding with VCP (ms) . 87

4.7 Information Data Processing Rate Calculated from CCS Profile of CC with VCP 87

4.8 Information Data Processing Rate Calculated from 3L-Measure Execution Time of CC with VCP 88

5.1 Comparison of Coding Gains of CTC and Tail-biting CC in AWGN at BER = 10^{-6} 92

5.2 Coding Gain Performance of Rate-1/3 CTC in AWGN at BER = 10^{-5} with Floating-Point and Fixed-Point Computation 96

5.3 CTC Rate-1/3 Encoder Execution Times Measured under 3L and the Corresponding Data Rate with 480-Bit Information Data Blocks 98

5.4 Profile of *CTC_Encoder* with QPSK Modulation for One Data Block 98

5.5 CTC Rate-1/3 Decoder Executive Times for 480 Information Bits Measured under 3L 99

5.6 Corresponding Processing Rates of CTC Rate-1/3 Decoder Based on 3L-Measured Execution Times for One Information Data Block of 480 Bits . . . 100

5.7 Profile of *CTC_Decoder* with QPSK Modulation for One Data Block in One Iteration 100

5.8 Profile of *Duo_Binnary_CRSC_Decoder* 100

5.9 Rate-1/3 CTC Processing Rate with 4 iterations in Decoding 101

5.10 Profile of Improved *Duo_Binnary_CRSC_Decoder* 102

5.11 Speed up in Decoding of One Data Block with QPSK Modulation for One Iteration 105

5.12 Improved CTC Decoding Speed Base on 3L-Measured Execution Times for One Information Data Block of 480 Bits 112

5.13 CTC Code Sizes 112

5.14 CC and CTC for Adder and Multiplier (Numbers) 115

5.15 Information Data Processing Rate Calculated from CCS for One Information Data Block of 480 Bits 116

5.16 Comparison of Decoder Speed for Tail-Biting CC, CTC, and LDPC Calculated from CCS 116

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.

CC (convolutional code) with tail-biting and CTCs (convolutional turbo codes) comprise

the mandatory channel coding schemes in Mobile WiMAX. A growing number of research studies are now available to shed some light on the convolution code and turbo code. A number of studies have been conducted using viterbi algorithm as the convolution decoding and BCJR algorithm as the turbo decoding. There have been numerous studies in the literature dealing with different decoding algorithms.

However we need to reduce the complexity for actual DSP implementation. In convolution code, the TI C6416 is equipped with a Viterbi-decoder coprocessor (VCP) [19]. Using this coprocessor can be helpful in raising the decoding speed. Furthermore, we also consider runnig the VCP under 3L Diamond RTOS for more digital signal processors (DSPs) applications. In addition, We also discuss the CTC in IEEE 802.16e for OFDMA. It uses a double binary circular recursive systematic convolutional (CRSC) code, which makes CTC efficient for coding of data cells in blocks. Note that "circular" can be equated with tail-biting, which means the initial state of the encoding start frame to be the same as the end state of the encoding end frame.

In this thesis, my work can be summarized as following:

- Study IEEE 802.16e specifications.
- Study tail-biting CC.
  - Study TI Viterbi-decoder coprocessor (VCP) and 3L Diamond EDMA.
  - Design tail-biting CC with VCP.
- Study CTCs.
  - Design rate-1/3 CTC floating-point and fixed-point versions.
  - Compare the performance and complexity.
  - Use optimization methods to implement CTC on DSP.

1.2 Organization of This Thesis

This thesis is organized as follows.

- Chapter 2 introduces CC with tail-biting and the CTC (convolution turbo code) of IEEE 802.16e specifications.
- Chapter 3 describes the DSP implementation environment, which is composed of the VCP, TI EDMA, and 3L EDMA.
- Chapter 4 discusses simulation and the DSP implementation of the convolutional decode with VCP.
- Chapter 5 discusses simulation and the DSP implementation of the CTC encoder and decoder.
- Chapter 6 contains the conclusion and points out some future work.

Chapter 2

Overview of CC and CTCs in IEEE 802.16e OFDMA

Convolutional code with tail-biting and convolutional turbo codes comprise the mandatory channel coding schemes in Mobile WiMAX. In this chapter, we introduce their specifications in IEEE 802.16e and their decoding methods.

2.1 Tail-Biting Convolutional Code Specifications [1]

The contents of this section have been taken to a large extent from [2], [3].

The mandatory channel coding scheme used in IEEE 802.16e OFDMA is as shown in Fig. 2.1. The input data stream is processed by the randomizer to clean up the bit correlation, and then each data block is encoded by the convolutional encoder with tail-biting, which means the encoder starts in the same state as it ends up after encoding. The block-by-block coding makes the convolutional code effectively a block code.

However, we do not implement the repetition block, which can be used to further increase signal margin over the modulation and FEC mechanisms, for the channel coding procedures in IEEE 802.16e. As the repetition block can be applied only to QPSK modulation, we bypass it in the present study. The reader interested in the repetition block can refer to

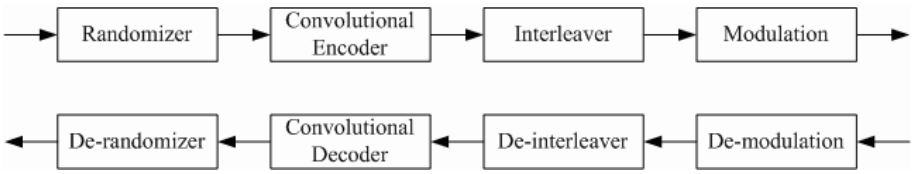


Figure 2.1: Structure of convolutional coding in transmitter (top path) and decoding in receiver (bottom path).

relevant material in [1].

We note again that our study concerns convolutional code with tail-biting, because an optional channel coding scheme of IEEE 802.16e is convolutional code with zero-tailing, which means the encoder is forced to return to the all-zero state after encoding. The two can be confused easily.

Between the convolutional encoder 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 [4].

To make the system more flexibly adaptable to the channel condition, 19 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 defined by the polynomial $1 + X^{14} + X^{15}$, as depicted in Fig. 2.2. Data randomization is performed on all data transmitted on the downlike (DL) and uplink (UL), expect the frame control header (FCH). The randomization is initialized on each FEC block.

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

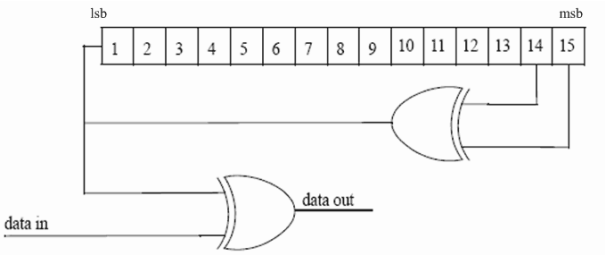


Figure 2.2: PRBS for data randomization (from [1]).

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. Here, the amount of data allocated means the amount of data that corresponds to the amount of slots $\lfloor N_s/R \rfloor$, where N_s is the number of the slots allocated for the data burst and R is the repetition factor used.

Each data byte to be transmitted shall enter sequentially into randomizer, msb first, to make the "0" and "1" bits in the input data streams well-distributed and hence improve the coding performance. The randomization is applied only to information bits. Preambles are not randomized. In both UL and DL, the randomizer is initialized with the vector

(LSB) 0 1 1 0 1 1 1 0 0 0 1 0 1 0 1 (MSB).

As we do not implement the HARQ mechanism, we bypass it in the present study. Note that the randomizer can be initialized with different vector for HARQ required, which can refer to [1] in detail.

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, as depicted in Fig. 2.3.

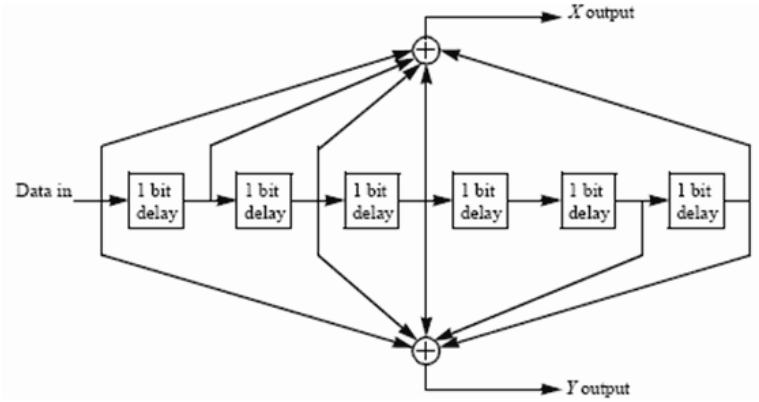


Figure 2.3: Convolutional encoder of rate 1/2 (from [1]).

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	$X_1Y_1Y_2$	$X_1Y_1Y_2X_3$

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 6 data bits of the block.

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 [8, Chapter 8].

Tail-Biting

The CC in IEEE 802.16e is terminated in a block; it therefore becomes a block code. In general, there are three methods to achieve code termination [5]. For ease of understanding, we describe these methods in terms of a binary (n, k, m) CC (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 lower 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 $n(L+m)/k$. 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 decoding 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 CC and does not lose rate compared with zero-tail CC. Nevertheless, we pay the price of a complex decoder. The optimal decoder of tail-biting convolutional code, as suggested in [5], 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

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

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 [6] and [7] and will be introduced later.

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.

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

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

QPSK

0	18	36		
1	19	37		
2	20	38		
3	21	39		
4	22	40		
16	34	52		
17	35	53		

16QAM

0	33	64		
1	32	65		
2	35	66		
3	34	67		
4	37	68		
5	36	69		
30	63	94		
31	62	95		

64QAM

0	56 (2)	109 (1)	162	
1	54 (0)	110 (2)	163	
2	55 (1)	108 (0)	164	
3	59 (2)	112 (1)	165	
4	57 (0)	113 (2)	166	
5	58 (1)	111 (0)	167	
51	107	160 (1)		
52	105	161 (2)		
53	106	159 (0)		

Figure 2.4: The second permutation of interleaver.

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.

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 of CC

In this section, we introduce the decoding method for CC. As there is a bit interleaver between the convolutional encoder and the modulator in the transmitter, the decoder should be based on the super-trellis combining the convolutional code, the interleaver, and the QAM

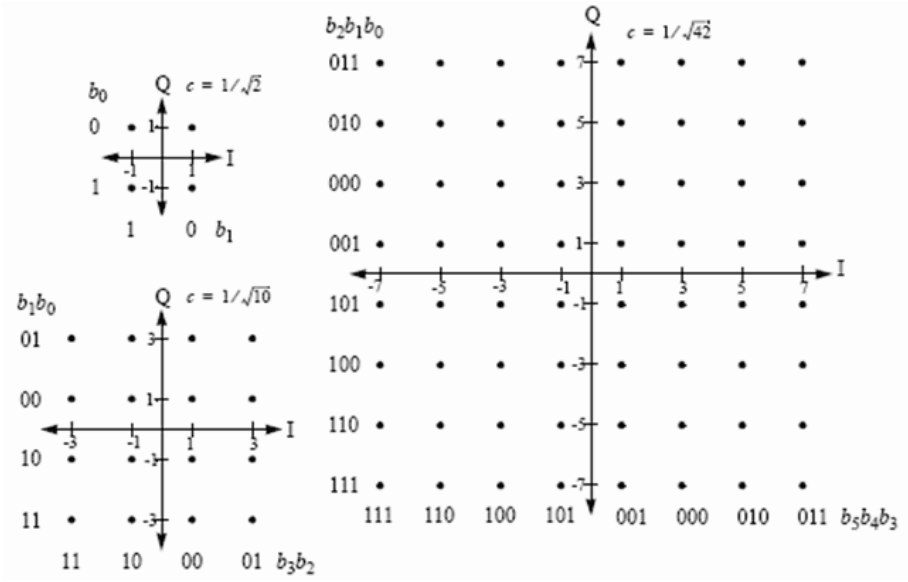


Figure 2.5: QPSK, 16-QAM, and 64-QAM constellations (from [1]).

modulator. So we mainly introduce the demodulation for bit-interleaved modulation in the section. For decoding of CC with tail-biting, we discuss it in chapter 3 along with the VCP and discuss how to do tail-biting with the VCP in chapter 4.

2.2.1 Demodulation for Bit-Interleaved Coded Modulation [9]

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, we can write the received signal of the sub-carrier as

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

where $G_{ch}[i]$ is the complex channel frequency response at 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 independent and identically distributed (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 (2.35) are equivalent to

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

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	$[\frac{1}{4}(m_0 - m_1) - 1)]^2$

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 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 in 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 of $(S_{I,k}^{(0)}, S_{I,k}^{(1)})$ for the generic bit $b_{I,k}$ in the case of 16-QAM.

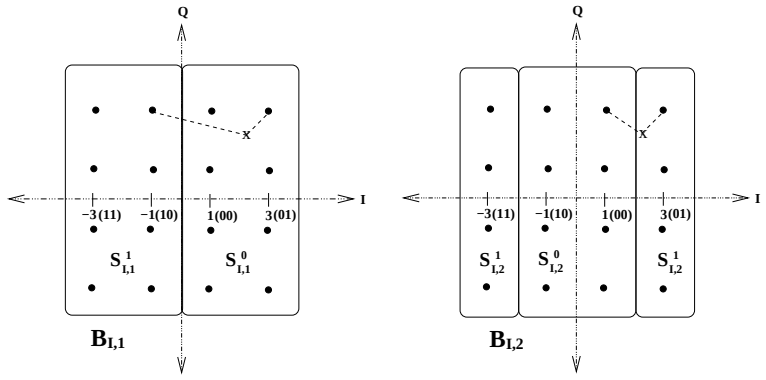


Figure 2.6: Metric partitions of the 16-QAM constellation (from [9]).

As a consequence,

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

can be simplified as follows.

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

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

The same observation holds for QPSK and 64-QAM constellations. For QPSK, $D_I = -y_I[i]$. For 64-QAM,

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

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

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

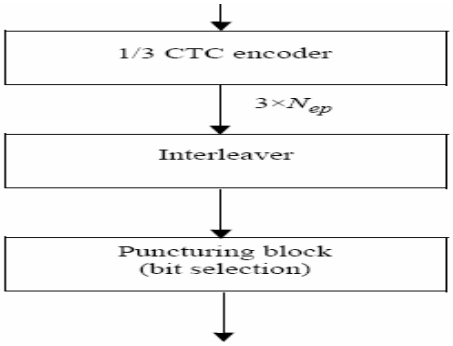


Figure 2.7: CTCs coding block diagram (from [1]).

2.3 Convolutional Turbo Codes Specifications [1]

The convolution turbo codes (CTCs) defined in IEEE 802.16e OFDMA is shown in Fig. 2.7. The input data are first encoded by the CTC encoder. Then, they are interleaved by the interleaving block and followed by puncturing. Likewise, there are three different modulation types. Note that the interleaving and the puncturing are also called subpacket generation. CTC is not only defined in IEEE 802.16e OFDMA but also in IEEE 802.16e OFDM. They are differentiated by their puncturing mechanism and subpacket generation.

Overview of CTC

Turbo code is first presented for error correction coding in 1993, which has provided for very long codewords with only modest decoding complexity.

In later years, researchers have shown that non-binary circular Turbo codes can offer many advantages in comparison to the classical single binary Turbo codes. Hence they have been used as one of FEC options in some recent satellite and mobile communication standards, in particular, DVB-RCS (Digital Video Broadcasting—Return Channel via Satellite) and WiMAX (IEEE 802.16e).

The Double-Binary Code Advantages [17]

- Better convergence: The advantage is well marked when replacing binary codes by double-binary code. The gain is less noticeable for inputs > 2.
- Larger minimum distance.
- Less sensitivity to puncturing patterns.
- Reduced latency.
  - As data are processed using symbols of 2 bits and ignoring the side effects, latency is divided by 2, from both coding and decoding viewpoints.
  - The trellis contains half as many states as a binary code of identical constraint length and the decoding hardware can be clocked at half the rate as a binary code [16, Chapter 12].
- Robustness of the decoder.
- Better performance for max-log-MAP algorithm: The duo-binary code can be decoded with max-log-MAP algorithm, which loses only about 0.1–0.2 dB relative to the optimal log-MAP algorithm. This is in contrast to binary codes, which lose about 0.3–0.4 dB when decoded with the max-log-MAP algorithm [16, Chapter 12].

A more detailed understanding of this relationship can be gained form [17].

2.3.1 CTC Encoder in IEEE 802.16e OFDMA [1]

The CTC encoder, including its constituent encoder, is shown in Figure 2.8. It uses a double binary circular recursive systematic convolutional (CRSC) code. The bits of the data to be encoded are alternately fed to A and B, starting with the MSB of the first byte being fed to

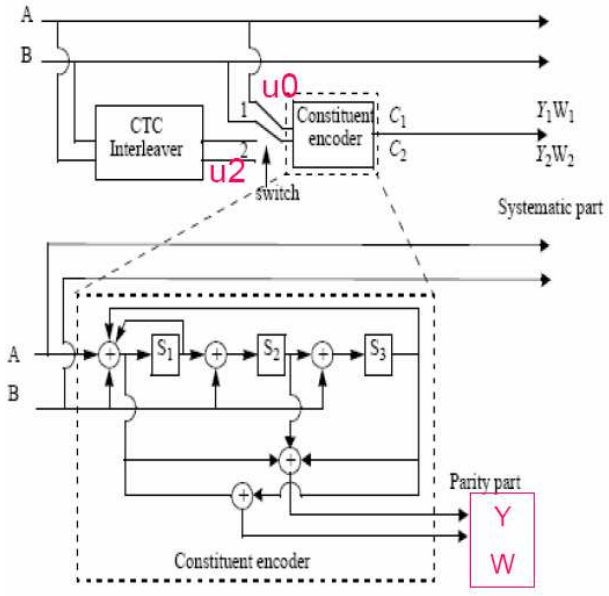


Figure 2.8: CTC encoder (modified from [1]).

A. The encoder is fed by blocks of k bits or N couples ($k = 2 \times N$ bits). For all the frame sizes, k is a multiple of 8 and N is a multiple of 4. Further, N is limited to $8 \leq N/4 \leq 1024$.

The polynomials defining the connections are described in octal and symbol notations as follows:

- For the feedback branch: 0xB, equivalently $1 + D + D^3$.
- For the Y parity bit: 0xD, equivalently $1 + D^2 + D^3$.
- For the W parity bit: 0x9, equivalently $1 + D^3$.

First, the encoder (after initialization by the circulation state S_{C1}) is fed the sequence in the natural order (position 1) with the incremental address $i = 0, \ldots, N-1$, which is called C_1 encoding. Second, the encoder (after initialization by the circulation state S_{C2}) is fed the sequence in the natural order (position 2) with the incremental address $j = 0, \ldots, N-1$,

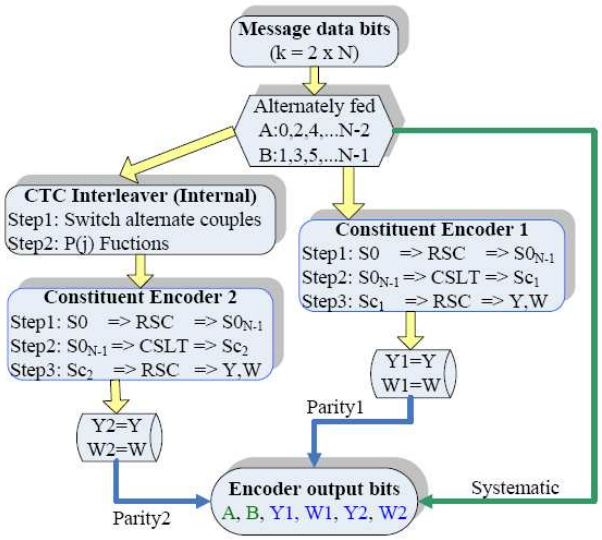


Figure 2.9: CTC rate 1/3 encoder flow chart.

which is called C_2 encoding. The order in which the encoded bits are fed into the subpacket generation block is A, B, Y_1, Y_2, W_1, W_2 =

$$A_0, A_1, ..., A_{N-1}, B_0, B_1, ..., B_{N-1},$$
$$Y_{1,0}, Y_{1,1}, ..., Y_{1,N-1}, Y_{2,0}, Y_{2,1}, ..., Y_{2,N-1},$$
$$W_{1,0}, W_{1,1}, ..., W_{1,N-1}, W_{2,0}, W_{2,1}, ..., W_{2,N-1}.$$

However, we can represent the above rule with the flow chart shown as Fig. 2.9. Note that CSLT express the *c*irculation *s*tate *l*ook-up *t*able, as shown in Table2.5.

The encoding block size shall depend on the number of slots allocated and the modulation specified for the current transmission. Concatenation of a number of slots can be performed in order to make larger blocks of coding where it is possible, with the limitation of not exceeding the largest supported block size for the applied modulation and coding.

There are 32 different block sizes as shown in Fig. 2.10. The specification for QPSK-1/2 may be in error, which should be 9 rather than 10. The concatenation rule shall not be used

Modulation and rate	j
QPSK-1/2	10
QPSK-3/4	6
16-QAM-1/2	5
16-QAM-3/4	3
64-QAM-1/2	3
64-QAM-2/3	2
64-QAM-3/4	2
64-QAM-5/6	2

Should be 9

Figure 2.10: CTC encoding slot concatenation for different rate (modified from [1]).

when using IR HARQ (incremental redundancy hybrid automatic repeat request).

2.3.2 CTC Interleaver [1]

The interleaver requires the parameters P_0, P_1, P_2, and P_3 shown in Fig. 2.11, which gives the block sizes, code rates, channel efficiency, and code parameters for different modulation and coding schemes.

The two-step interleaver can be performed as shown in Fig. 2.12, where two possible errors in the draft standard is indicated.

2.3.3 CTC Tail-Biting [1], [10]

For recursive encoders, tail-biting is not as easy as it is for non-recursive encoders. To ensure that the starting state is the same as the ending state, which is called circulation state, for recursive encoders an initial encoding of the whole sequence has to be performed [10].

The initial encoding is started in the all-zero state and depending on the information sequence it ends up in a special state, S_{end}. Based on this ending state, the circulation state

Modulation	Data block size (bytes)	Encoded data block size (bytes)	Code rate	N bits	P_0	P_1	P_2	P_3
QPSK	6	12	1/2	24	5	0	0	0
QPSK	12	24	1/2	48	13	24	0	24
QPSK	18	36	1/2	72	11	6	0	6
QPSK	24	48	1/2	96	7	48	24	72
QPSK	30	60	1/2	120	13	60	0	60
QPSK	36	72	1/2	144	17	74	72	2
QPSK	48	96	1/2	192	11	96	48	144
QPSK	54	108	1/2	216	13	108	0	108
QPSK	60	120	1/2	240	13	120	60	180
QPSK	9	12	3/4	36	11	18	0	18
QPSK	18	24	3/4	72	11	6	0	6
QPSK	27	36	3/4	108	11	54	56	2
QPSK	36	48	3/4	144	17	74	72	2
QPSK	45	60	3/4	180	11	90	0	90
QPSK	54	72	3/4	216	13	108	0	108
16-QAM	12	24	1/2	48	13	24	0	24
16-QAM	24	48	1/2	96	7	48	24	72
16-QAM	36	72	1/2	144	17	74	72	2
16-QAM	48	96	1/2	192	11	96	48	144
16-QAM	60	120	1/2	240	13	120	60	180
16-QAM	18	24	3/4	72	11	6	0	6
16-QAM	36	48	3/4	144	17	74	72	2
16-QAM	54	72	3/4	216	13	108	0	108
64-QAM	18	36	1/2	72	11	6	0	6
64-QAM	36	72	1/2	144	17	74	72	2
64-QAM	54	108	1/2	216	13	108	0	108
64-QAM	24	36	2/3	96	7	48	24	72
64-QAM	48	72	2/3	192	11	96	48	144
64-QAM	27	36	3/4	108	11	54	56	2
64-QAM	54	72	3/4	216	13	108	0	108
64-QAM	30	36	5/6	120	13	60	0	60
64-QAM	60	72	5/6	240	13	120	60	180

Figure 2.11: CTC channel coding per modulation (modified from [1]).

Step 1: Switch alternate couples

Let the sequence $u_0 = [(A_0, B_0), (A_1, B_1), (A_2, B_2), (A_3, B_3), \ldots, (A_{N-1}, B_{N-1})]$ be the input to first encoding C_1.

for $i = 0 \ldots N-1$

if (i mod 2), let $(A_i, B_i) \rightarrow (B_i, A_i)$ (i.e., switch the couple)

This step gives a sequence $u_1 = [(A_0, B_0), (B_1, A_1), (A_2, B_2), (B_3, A_3), \ldots, (B_{N-1}, A_{N-1})] = [u_1(0), u_1(1), u_1(2), u_1(3), \ldots, u_1(N-1)]$.

Step 2: *P(j)*

The function $P(j)$ provides the address of the couple of the sequence u_1 that shall be mapped onto the address j of the interleaved sequence (i.e., $u_2(j) = u_1(P(j))$).

for $j = 0 \ldots N-1$

switch (j mod 4):

case 0: $P(j) = (P_0 \cdot j + 1) \bmod N$

case 1: $P(j) = (P_0 \cdot j + 1 + N/2 + P_1) \bmod N$

case 2: $P(j) = (P_0 \cdot j + 1 + P_2) \bmod N$

case 3: $P(j) = (P_0 \cdot j + 1 + N/2 + P_3) \bmod N$

This step gives a sequence $u_2 = [u_1(P(0)), u_1(P(1)), u_1(P(2)), u_1(P(3)), \ldots, u_1(P(N-1))] = [(B_{P(0)}, A_{P(0)}), (A_{P(1)}, B_{P(1)}), (B_{P(2)}, A_{P(2)}), (A_{P(3)}, B_{P(3)}), \ldots, (A_{P(N-1)}, B_{P(N-1)})]$. Sequence u_2 is the input to the second encoding C_2.

Reverse error ?

Figure 2.12: CTC interleaver in two steps (modified from [1]).

can be computed using linear algebra methods based on the state space description of the encoder. In order to eliminate this linear algebra computation, the IEEE 802.16 provides a so-called circulation state look-up table, where the correspondence between the final state S_{end} of the initial encoding process and the circulation state as a function of the information sequence length is listed in Table 2.5.

Afterwards, the real encoding can be started, whereby the encoder state is initialized now with the circulation state. Hence, a tail-biting encoder needs two complete encoding processes, which adds complexity to the encoder. Complexity is also added to the decoder of the constituent code. The complexity added to the decoder compared to the case where the starting and ending state is known to the decoder is in the additional wrap-around for the forward and backward recursion of the MAP decoder. Since the wrap-around length can be kept small, the additional complexity is quite small [10].

Table 2.5: Circulation State Look-Up Table (S_{C1} and S_{C2})

$N \bmod 7$	$S0_{N-1}$							
	0	1	2	3	4	5	6	7
1	0	6	4	2	7	1	3	5
2	0	3	7	4	5	6	2	1
3	0	5	3	6	2	7	1	4
4	0	4	1	5	6	2	7	3
5	0	2	5	7	1	3	4	6
6	0	7	6	1	3	4	5	2

Determination of CTC Circulation States [1]

The state of the encoder is denoted S $(0 \leq S \leq 7)$ with $S = 4S_1 + 2S_2 + S_3$, as shown in Fig. 2.8. The circulation states S_{C1} and S_{C2} are determined by the following operations:

- Initialize the encoder with state 0.
- Encode the sequence in the natural order for the determination of S_{C1} or in the interleaved order for determination of S_{C2}. Let the final state in each case be denoted $S0_{N-1}$.
- According to the length N of the sequence, use Table 2.5 to find S_{C1} and S_{C2}.

2.3.4 Subpacket Generation (Channel Interleaver or Interleaver and Puncturing) [1]

The proposed FEC structure punctures the mother codeword to generate a subpacket with various coding rates. The framework consists of the following:

- bit separation,

- subblock interleaving,
- bit grouping, and
- bit selection.

The subpacket is also used in HARQ packet transmission. Figure 2.7 shows the block diagram of subpacket generation. A rate-1/3 CTC encoded codeword goes through interleaving and the puncturing. Figure 2.13 shows the block diagram of the interleaving block. The puncturing is performed to select a consecutive interleaved bit sequence that starts at some point of whole codeword.

For the first transmission, the subpacket is generated to select the consecutive interleaved bit sequence that starts from the first bit of the systematic part of the mother codeword. The length of the subpacket is chosen according to the needed coding rate reflecting the channel condition. The first subpacket can also be used as a codeword with the needed coding rate for a burst where HARQ is not applied.

Bit Separation

All of the encoded bits can be demultiplexed into six subblocks denoted A, B, $Y1$, $Y2$, $W1$, and $W2$. The encoder output bits are sequentially distributed into the six subblocks with the first N bits going to the A subblock, the second N to the B subblock, the third N to the $Y1$ subblock, the fourth N to the $Y2$ subblock, the fifth N to the $W1$ subblock, and the sixth N to the $W2$ subblock.

Subblock Interleaving

The six subblocks can be interleaved separately. The interleaving is performed in unit of bits. The sequence of interleaver output bits for each subblock can be generated by the

procedure described below. The entire subblock of bits to be interleaved is written into an array at addresses from 0 to the number of the bits minus one $(N-1)$, and the interleaved bits are read out in a permuted order with the ith bit being read from the address AD_i $(i = 0, \ldots, N-1)$, as follows:

1. Determine the subblock interleaver parameters, m and J. Table 2.6 gives these parameters.

2. Initialize i and k to 0.

3. Form a tentative output address T_k according to

$$T_k = 2^m(k \bmod J) + BRO_m(\lfloor k/J \rfloor) \tag{2.16}$$

   where $BRO_m(y)$ indicates the bit-reversed m-bit value of y (e.g.,$BRO_3(6) = 3$).

4. If T_k is less than N, $AD_i = T_k$ and increment i and k by 1. Otherwise, discard T_k and increment k only.

5. Repeat steps 3 and 4 until all N interleaver output addresses are obtained.

Bit Grouping

The channel interleaver output sequence can consist of the interleaved A and B subblock sequences, followed by a bit-by-bit multiplexed sequence of the interleaved $Y1$ and $Y2$ subblock sequences, followed by a bit-by-bit multiplexed sequence of the interleaved $W1$ and $W2$ subblock sequences.

The bit-by-bit multiplexed sequence of interleaved $Y1$ and $Y2$ subblock sequences can consist of the first output bit from the $Y1$ subblock interleaver, the first output bit from the $Y2$ subblock interleaver, the second output bit from the $Y1$ subblock interleaver, the

Table 2.6: Parameters for the Subblock Interleavers

	Subblock interleaver		
Block size (bits) N_{EP}	N	m	J
48	24	3	3
72	36	4	3
96	48	4	3
144	72	5	3
192	96	5	3
216	108	5	4
240	120	6	2
288	144	6	3
360	180	6	3
384	192	6	3
432	216	6	4
480	240	7	2

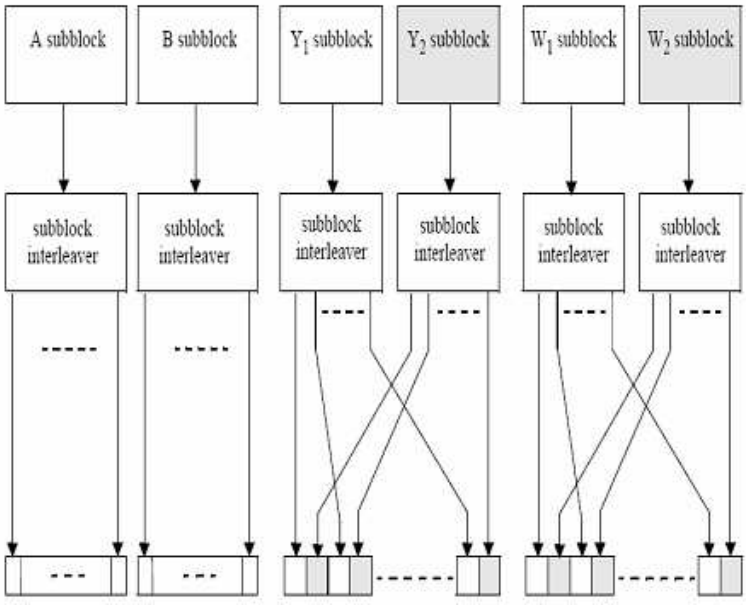


Figure 2.13: Block diagram of CTC channel interleaving scheme (from [1]).

second output bit from the $Y2$ subblock interleaver, etc. The bit-by-bit multiplexed sequence of interleaved $W1$ and $W2$ subblock sequences can consist of the first output bit from the $W1$ subblock interleaver, the first output bit from the $W2$ subblock interleaver, the second output bit from the $W1$ subblock interleaver, the second output bit from the $W2$ subblock interleaver, etc. Figure 2.13 shows the interleaving scheme. The order of bit grouping sequence is as follows:

$$A'_0,A'_1,...,A'_{N-1},B'_0,B'_1,...,B'_{N-1},$$
$$Y'_{1,0},Y'_{2,0},Y'_{1,1},Y'_{2,1},Y'_{1,2},Y'_{2,2},...,Y'_{1,N-1},Y'_{2,N-1},$$
$$W'_{1,0},W'_{2,0},W'_{1,1},W'_{2,1},W'_{1,2},W'_{2,2},...,W'_{1,N-1},W'_{2,N-1}.$$

Bit Selection

Lastly, bit selection is performed to generate the subpacket. The puncturing block is referred as bits selection in the viewpoint of subpacket generation. The mother code is transmitted with one of the subpackets. The bits in a subpacket are formed by selecting specific sequences of bits from the interleaved CTC encoder output sequence. The resulting subpacket sequence is a binary sequence of bits for the modulator. The parameters for bit selection are listed below:

- k: the subpacket index when IR HARQ is enabled.
  - When IR HARQ is not used, k=0 (for the first transmission and increases by one for the next subpacket).
  - When there are more than one FEC block in a burst, the subpacket index for each FEC block shall be the same.
- N_{EP}: the number of bits in the encoder packet (before encoding).

- N_{SCHk}: the number of concatenated slots for the subpacket, as defined in [1, Table 569] for the non-HARQ and Chase HARQ CTC schemes.

- m_k: the modulation order for the kth subpacket (m_k=2 for QPSK, 4 for 16-QAM, 6 for 64QAM).

- $SPID_k$: the subpacket ID for the kth subpacket (for the first subpacket, $SPID_{k=0}$=0).

Also, let the scrambled and selected bits be numbered from zero with the 0th bit being the first bit in the sequence. Then, the index of the ith bit for the kth subpacket shall be

$$S_{k,i} = (F_k + i) \mathrm{mod}(3 \cdot N_{EP}) \tag{2.17}$$

where $i = 0, \ldots, L_k - 1$, $L_k = 48 \cdot N_{SCHk} \cdot m_k$, and $F_k = (SPID_k \cdot L_k) \mathrm{mod}(3 \cdot N_{EP})$. The N_{EP}, N_{SCHk}, m_k , and $SPID$ values are determined by the base station (BS) and can be inferred by the subscriber station (SS) through the allocation size in the DL-MAP and UL-MAP. The above bit selection makes the following possible.

- The first transmission includes the systematic part of the mother code. Thus it can be used as the codeword for a burst where the HARQ is not applied or when Chase HARQ is applied.

- The location of the subpacket can be determined by the $SPID$ without the knowledge of previous subpacket. This is a very important property for IR HARQ retransmission.

Note that the optional IR HARQ is not considered in our research, so we bypass a detailed introduction of the IR HARQ mechanism.

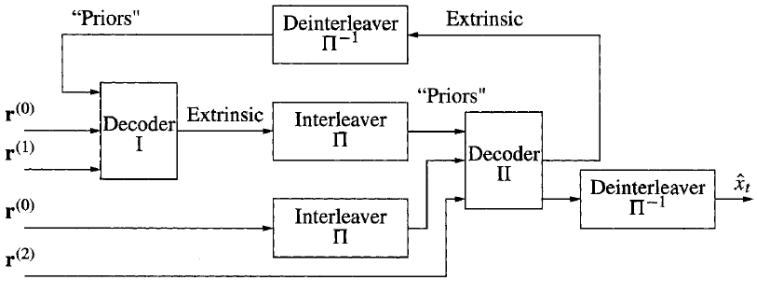


Figure 2.14: Block diagram of a turbo decoder (from [11]).

2.4 Decoding of CTC

2.4.1 The Turbo Decoding Algorithm [11]

A key in turbo codes is the iterative decoding algorithm. In iterative decoding, the decoders for the constituent encoders take turns operating on the received data.

Each decoder produces an estimate of the probabilities of the transmitted symbols; therefore, the decoders are soft output decoders. Probabilities of the symbols from one decoder, known as extrinsic probabilities, are interleaved and passed to the other decoder, where they are used as prior probabilities for the other decoder. The decoder thus passes probabilities back and forth between the decoders, with each decoder combining the evidence it receives from the incoming prior probabilities with the parity information provided by the code. After some number of iterations, hopefully the decoder converges to an estimate of the transmitted codeword. Since the output of one decoder is fed to the input of the next decoder, the decoding algorithm is called a turbo decoder, for it is reminiscent of turbo charging an automobile engine using engine-heated air at the air intake. Thus it is not really the code which is "turbo," but rather the decoding algorithm which is "turbo." The general operation of the turbo decoding algorithm is shown in Fig. 2.14.

The MAP Decoding Algorithm [11], [13]

One maximum a posteriori (MAP) decoding algorithm particularly suitable for estimating bit and/or state probabilities for a finite-state Markov system is the BCJR algorithm, named after Bahl, Cock, Jelinek, and Raviv who proposed it originally in 1974 [12]. While this algorithm has been known for some time, it was not extensively used for the decoding of convolutional codes because of the availability of a lower complexity Viterbi algorithm (for maximum-likelihood decoding of convolutional codes).

In many respects, the BCJR algorithm is similar to the Viterbi algorithm. However, the conventional Viterbi algorithm computes hard decisions by outputting a single overall decision of the entire sequence of bits (or codeword) at the end, without providing the reliability of the decoder decisions on individual bits. Furthermore, the branch metric is based upon log likelihood values; no prior information is incorporated into the decoding process. The BCJR algorithm, on the other hand, computes soft outputs in the form of posterior probabilities for each message bit. While the Viterbi algorithm produces the maximum likelihood message sequence (or codeword), the BCJR algorithm produces the a posteriori most likely sequence of message bits, where the sequence of bits may not correspond to a continuous path through the trellis. The BCJR algorithm is a soft-input soft-output decoder that can be used directly in turbo decoding whereas the conventional Viterbi algorithm cannot without some modification to yield the required soft output. The BCJR algorithm for MAP decoding of convolutional codes consists of the following steps:

- Compute branch metric γ.
- Compute forward state metric α.
- Compute backward state metric β.

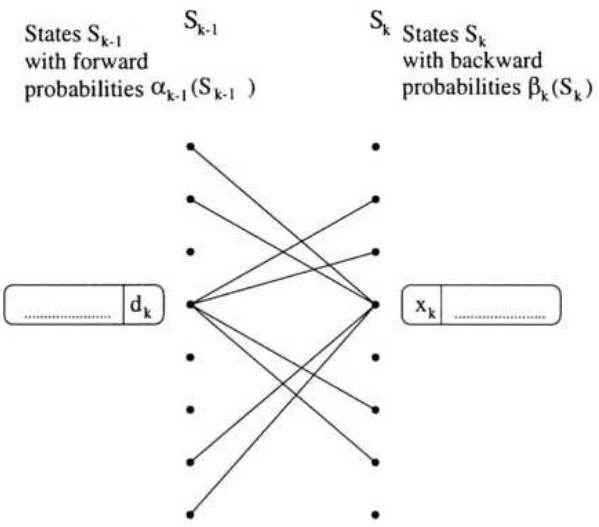


Figure 2.15: CTC trellis structure of duo-binary convolutional code with feedback encoder (from [14]).

- Compute extrinsic log likelihood ratio L_e.

A more detailed understanding can be gained from [11].

2.4.2 Decoding Rule for CRSC Codes with Non-binary Trellis [14]

The trellis of a double-binary feedback convolutional encoder has the structure shown in Fig. 2.15. The goal of the MAP algorithm is to provide us with

$$\begin{aligned} L_i(d_k) &= \ln \frac{P_r[d_k = i|\text{Observation}]}{P_r[d_k = 0|\text{Observation}]} \\ &= \ln \frac{\sum_{d_k=i}^{(S_{k-1},S_k)} p(S_{k-1}, S_k, \{y_k\})}{\sum_{d_k=0}^{(S_{k-1},S_k)} p(S_{k-1}, S_k, \{y_k\})}, \quad i = 1, 2, 3, \end{aligned} \tag{2.18}$$

where y_k is the received sample at time k. The index pair (S_{k-1}, S_k) determines the information symbol (bit couple) d_k and the coded symbol x_k from time $k-1$ to time k where d_k is in GF(2^2) with elements {0,1,2,3}. The sum of the joint probabilities $p(S_{k-1}, S_k, \{y_k\})$ in the numerator or in the denominator of (2.18) is taken over all labeled with $d_k = i$, $i = 0, 1, 2, 3$,

where we have used decimal notation for d_k instead of binary for convenience. With a memoryless transmission channel, the joint probability $p(S_{k-1}, S_k, \{y_k\})$ can be written as the product of three independent probabilities

$$\begin{aligned} p(S_{k-1}, S_k, \{y_k\}) &= p(S_{k-1}, y_{j<k}) \cdot p(S_k, y_k|S_{k-1}) \cdot p(y_{j>k}, S_k) \\ &\triangleq \alpha_{k-1}(S_{k-1}) \cdot \gamma_k(S_{k-1}, S_k) \cdot \beta_k(S_k) \end{aligned} \tag{2.19}$$

where $y_{j<k}$ denotes the sequence of received symbols y_j from the beginning of the trellis up to time $k-1$ and $y_{j>k}$ is the corresponding sequence from time $k+1$ up to the end of the trellis. The forward recursion of the MAP algorithm yields

$$\alpha_k(S_k) = \sum_{S_{k-1}} \alpha_{k-1}(S_{k-1}) \cdot \gamma_k(S_{k-1}, S_k). \tag{2.20}$$

The backward recursion yields

$$\beta_{k-1}(S_{k-1}) = \sum_{S_k} \gamma_k(S_{k-1}, S_k) \cdot \beta_k(S_k). \tag{2.21}$$

When a transition between S_{k-1} and S_k exists, the branch transition probability is given by

$$\begin{aligned} \gamma_k(S_{k-1}, S_k) &= p(S_k, y_k|S_{k-1}) \\ &= p(S_k|S_{k-1}) \cdot p(y_k|S_{k-1}, S_k) \\ &= P(d_k) \cdot p(y_k|d_k). \end{aligned} \tag{2.22}$$

Let the natural logarithm of the branch transition probability metric be

$$\Gamma_k(S_{k-1}, S_k) = \ln \gamma_k(S_{k-1}, S_k) \tag{2.23}$$

and the natural logarithms of $\alpha_k(S_k)$ and $\beta_k(S_k)$ be

$$\begin{aligned} A_k(S_k) &= \ln \alpha_k(S_k) \\ &= \ln \sum_{S_{k-1}} e^{A_{k-1}(S_{k-1}) + \Gamma_k(S_{k-1}, S_k)}, \end{aligned} \tag{2.24}$$

$$
\begin{aligned}
B_{k-1}(S_{k-1}) &= \ln \beta_{k-1}(S_{k-1}) \\
&= \ln \sum_{S_k} e^{\Gamma_k(S_{k-1},S_k)+B_k(S_k)}. \qquad (2.25)
\end{aligned}
$$

Then the log-likelihood ratios (2.18) for $i = 1, 2, 3$ are given by

$$
\begin{aligned}
L_i(d_k) &= \ln \frac{\sum_{d_k=i}^{(S_{k-1},S_k)} p(S_{k-1}, S_k, \{y_k\})}{\sum_{d_k=0}^{(S_{k-1},S_k)} p(S_{k-1}, S_k, \{y_k\})} \\
&= \ln \frac{\sum_{d_k=i}^{(S_{k-1},S_k)} \alpha_{k-1}(S_{k-1}) \cdot \gamma_k^i(S_{k-1}, S_k) \cdot \beta_k(S_k)}{\sum_{d_k=0}^{(S_{k-1},S_k)} \alpha_{k-1}(S_{k-1}) \cdot \gamma_k^0(S_{k-1}, S_k) \cdot \beta_k(S_k)} \\
&= \ln \frac{\sum_{d_k=i}^{(S_{k-1},S_k)} e^{A_{k-1}(S_{k-1})+\Gamma_k^i(S_{k-1},S_k)+B_k(S_k)}}{\sum_{d_k=0}^{(S_{k-1},S_k)} e^{A_{k-1}(S_{k-1})+\Gamma_k^0(S_{k-1},S_k)+B_k(S_k)}}. \qquad (2.26)
\end{aligned}
$$

2.4.3 Simplified Max-Log-MAP Algorithm for Double-Binary CTC [14]

Implementing (2.26) in hardware is difficult and complex. It is also relatively complicated to implement it in DSP software. We consider the suboptimal max-log-MAP algorithm for double binary convolutional turbo codes. First, from (2.22) and (2.23),

$$
\begin{aligned}
\Gamma_k(S_{k-1}, S_k) &= \ln \gamma_k(S_{k-1}, S_k) \\
&= \ln[p(y_k|d_k) \cdot P(d_k)]. \qquad (2.27)
\end{aligned}
$$

The distribution of the received symbols is given by, for i=0,1,2,3,

$$
\begin{aligned}
p(y_k|d_k = i) &= p(y_k^s|x_k^s(i)) \cdot p(y_k^p|x_k^p(i, S_{k-1}, S_k)) \\
&= \frac{1}{\pi \cdot N_0} e^{-\frac{E_s}{N_0}[(y_k^{s,I}-x_k^{s,I}(i))^2+(y_k^{s,Q}-x_k^{s,Q}(i))^2]} \\
&\cdot \frac{1}{\pi \cdot N_0} e^{-\frac{E_s}{N_0}[(y_k^{p,I}-x_k^{p,I}(i,S_{k-1},S_k))^2+(y_k^{p,Q}-x_k^{p,Q}(i,S_{k-1},S_k))^2]} \\
&= C_k \cdot e^{0.5 \cdot L_c \cdot [y_k^{s,I} \cdot x_k^{s,I}(i)+y_k^{s,Q} \cdot x_k^{s,Q}(i)+y_k^{p,I} \cdot x_k^{p,I}(i,S_{k-1},S_k)+y_k^{p,Q} \cdot x_k^{p,Q}(i,S_{k-1},S_k)]} \qquad (2.28)
\end{aligned}
$$

where y_k^s and y_k^p represent the received systematic and parity symbols, respectively, $y_k^{s,I}$, $y_k^{s,Q}$, $y_k^{p,I}$, and $y_k^{p,Q}$ represent the received bit values transmitted through the I and Q channels, respectively, $L_c = 4 \cdot (\text{fading factor}) \cdot (\text{code rate}) \cdot \frac{E_b}{N_0}$ represent the channel reliability, and

$C_k = (\frac{1}{\pi \cdot N_0})^2 e^{-\frac{E_s}{N_0}[(y_k^{s,I})^2+(x_k^{s,I}(i))^2+(y_k^{s,Q})^2+(x_k^{s,Q}(i))^2+(y_k^{p,I})^2+(x_k^{p,I}(i,S_{k-1},S_k))^2+(y_k^{p,Q})^2+(x_k^{p,Q}(i,S_{k-1},S_k))^2]}$.

Hence,

$$\begin{aligned}\Gamma_k(S_{k-1},S_k) &= \ln[p(y_k|d_k)\cdot P(d_k)] \\ &= 0.5\cdot L_c\cdot[y_k^{s,I}\cdot x_k^{s,I}(i)+y_k^{s,Q}\cdot x_k^{s,Q}(i)+y_k^{p,I}\cdot x_k^{p,I}(i,S_{k-1},S_k) \\ &+ y_k^{p,Q}\cdot x_k^{p,Q}(i,S_{k-1},S_k)]+\ln P(d_k)+K \qquad (2.29)\end{aligned}$$

where the constant K includes the constants and common terms that are cancelled in comparisons at later stages. Note that

$$\begin{aligned}A_k(S_k) &= \ln\sum_{S_{k-1}} e^{A_{k-1}(S_{k-1})+\Gamma_k(S_{k-1},S_k)} \\ &\approx \max_{S_{k-1}}[A_{k-1}(S_{k-1})+\Gamma_k(S_{k-1},S_k)] \qquad (2.30)\end{aligned}$$

$$\begin{aligned}B_{k-1}(S_{k-1}) &= \ln\sum_{S_k} e^{\Gamma_k(S_{k-1},S_k)+B_k(S_k)} \\ &\approx \max_{S_k}[\Gamma_k(S_{k-1},S_k)+B_k(S_k)] \qquad (2.31)\end{aligned}$$

The above can be derived by the Jacobian logarithm [11], i.e.,

$$\ln(e^{L1}+e^{L2}) = \max(L_1,L_2)+\ln(1+e^{-|L_1-L_2|}) \qquad (2.32)$$

If the correction term (i.e., the second RHS term) is omitted and only the max term is retained, we obtain the above max-function (max-log-MAP) approximation. For iterative decoding of circular trellis, tail-biting gives

$$A_0(S_0) = A_N(S_N) \quad \forall S_0, \qquad (2.33)$$

$$B_N(S_N) = B_0(S_0) \quad \forall S_N. \qquad (2.34)$$

As a result, the log-likelihood ratios (2.26) reduce to

$$\begin{aligned}L_i(d_k) &\approx \max_{(S_{k-1},S_k)}[A_{k-1}(S_{k-1})+\Gamma_k^i(S_{k-1},S_k)+B_k(S_k)] \\ &- \max_{(S_{k-1},S_k)}[A_{k-1}(S_{k-1})+\Gamma_k^0(S_{k-1},S_k)+B_k(S_k)]. \qquad (2.35)\end{aligned}$$

We omit the detailed mathematical derivation for separating the log-likelihood ratios into intrinsic (prior information), systematic and extrinsic information. The interested reader may refer to [14]. It turns out that the extrinsic information can be expressed as

$$\begin{aligned} L_i^e(\hat{d}_k) &= L_i(\hat{d}_k) - 0.5 \cdot [y_k^{s,I} \cdot x_k^{s,I}(i) + y_k^{s,Q} \cdot x_k^{s,Q}(i)] \\ &+ 0.5 \cdot [y_k^{s,I} \cdot x_k^{s,I}(0) + y_k^{s,Q} \cdot x_k^{s,Q}(0)] - \ln \frac{P[d_k = i]}{P[d_k = 0]}. \end{aligned} \tag{2.36}$$

The extrinsic information of the next decoder is computed from the prior information of previous decoder as

$$L_i^a(d_k) = \ln \frac{P[d_k = i]}{P[d_k = 0]} \tag{2.37}$$

where $i = 0, 1, 2, 3$. Since

$P[d_k = 01] = e^{L_1^a(d_k)} \cdot P[d_k = 00]$, $P[d_k = 10] = e^{L_2^a(d_k)} \cdot P[d_k = 00]$,

$P[d_k = 11] = e^{L_3^a(d_k)} \cdot P[d_k = 00]$, and $P[d_k = 00] + P[d_k = 01] + P[d_k = 10] + P[d_k = 11] = 1$,

we have

$P[d_k = 00] = \frac{1}{1+e^{L_1^a(d_k)}+e^{L_2^a(d_k)}+e^{L_3^a(d_k)}}$, $P[d_k = 01] = \frac{L_1^a(d_k)}{1+e^{L_1^a(d_k)}+e^{L_2^a(d_k)}+e^{L_3^a(d_k)}}$,

$P[d_k = 10] = \frac{L_2^a(d_k)}{1+e^{L_1^a(d_k)}+e^{L_2^a(d_k)}+e^{L_3^a(d_k)}}$, $P[d_k = 11] = \frac{L_3^a(d_k)}{1+e^{L_1^a(d_k)}+e^{L_2^a(d_k)}+e^{L_3^a(d_k)}}$.

Using max-function approximation yields

$\ln P[d_k = 00] = -\max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)]$,

$\ln P[d_k = 01] = L_1^a(d_k) - \max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)]$,

$\ln P[d_k = 10] = L_2^a(d_k) - \max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)]$,

$\ln P[d_k = 11] = L_3^a(d_k) - \max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)]$.

Assuming equally likely symbols initially, we have

$$A_0(S_0) = 0 \quad \forall S_0, \tag{2.38}$$

$$B_N(S_N) = 0 \quad \forall S_N, \tag{2.39}$$

$$L_i^a(d_k) = 0 \quad \forall i, d_k. \tag{2.40}$$

After sufficient decoding iterations, the decisions are made according to

$$\hat{d_k} = \begin{cases} 01, & = \text{if } L(\hat{d_k}) = L_1^a(d_k) \text{ and } L_1^a(d_k) > 0, \\ 10, & = \text{if } L(\hat{d_k}) = L_2^a(d_k) \text{ and } L_2^a(d_k) > 0, \\ 11, & = \text{if } L(\hat{d_k}) = L_3^a(d_k) \text{ and } L_3^a(d_k) > 0, \\ 00, & = \text{else}, \end{cases} \tag{2.41}$$

where $L(\hat{d_k}) = \max[L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)]$.

This above algorithm have been known as the max-log-MAP algorithm which only uses the max functions to compute log-likelihood ratios. But coming with the approximation to reducing log-likelihood ratios is some performance degradation. We will see the effect later in the simulation results.

Chapter 3

DSP Implementation Environment

In our implementation, we employ the DSP baseboard SMT395 made by the Sundance company, which have a Texas Instruments (TI) TMS320C6416T DSP chip and a Xilinx Virtex-II Pro FPGA. In this chapter, we discuss the DSP system development environment, especially the VCP (Viterbi decoder coprocessor) and its features. The TI's Code Composer Studio (CCS) EDMA and the 3L Diamond EDMA are also introduced.

3.1 The DSP Baseboard

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 TI's Code Composer Studio and the 3L Diamond real-time operating system (RTOS) to enable multi-DSP system implementation with minimum effort by the programmer.

Features of the SMT395 board include:

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

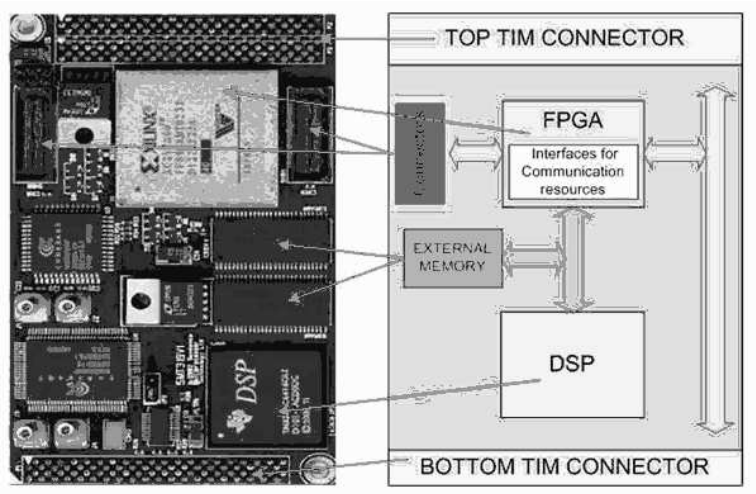


Figure 3.1: Sundance's SMT395 module (from [18]).

- 256 Mbytes of SDRAM at 133 MHz.
- Eight 2 Gbit/sec Rocket Serial Links (RSL) for inter module communication.
- 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 Viterbi-Decoder Coprocessor (VCP) [19]

The Viterbi-decoder coprocessor (VCP) is on some of the number of the TMS320C6000 DSP family, including C6416, C6418, and C6455. It has been designed to perform Viterbi decoding for IS2000 and 3GPP wireless standards. We can also use it for other convolutional decoding applications, including WiMAX.

3.2.1 Overview of VCP [19], [21], [22]

The VCP should be accessed using the EDMA (Enhanced Direct Memory Access) for mostly, but the CPU must first configure the VCP control values. There are also a number of

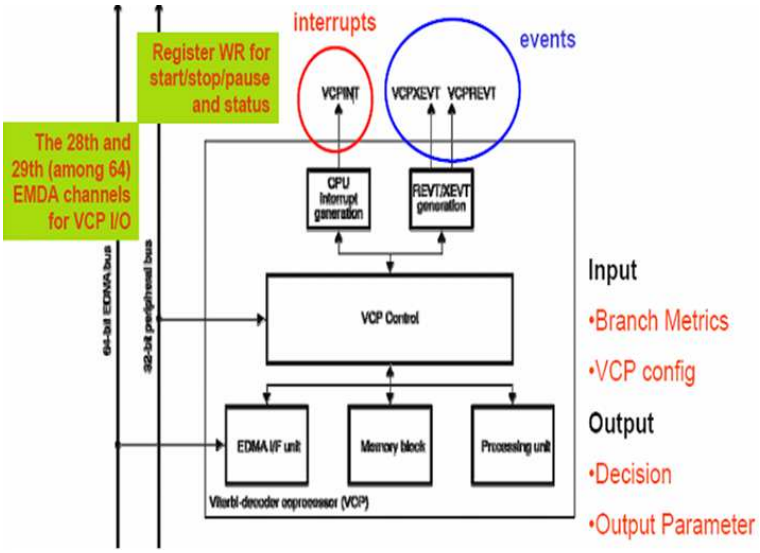


Figure 3.2: VCP block diagram (modified from [19]).

functions available to the CPU to monitor the VCP status and access decision and output parameter data.

The DSP controls the operation of the VCP using memory-mapped registers and data buffers. The DSP typically sends and receives data using synchronized EDMA transfer through the 64-bit EDMA bus. The VCP sends two synchronization events to the EDMA: a receiver event (VCPREVT) and a transmit event (VCPXEVT), as shown in Fig. 3.2. The VCP is composed of VCP Control, EDMA I/F unit, memory block, processing unit, CPU interrupt generator, and REVT/XEVT generator. Fig. 3.2 shows two VCP external communication mechanisms, in one of which DSP (CPU) accesses VCP Control through the 32-bit peripheral bus and in the other EDMA I/F unit through the 64-bit EDMA bus. In the latter case, EDMA channel 28 (RX) is for VCP transmission to DSP and EDMA channel 29 (TX) is for DSP transmission to VCP.

Fig. 3.3 and Fig. 3.4 show the DSP chip architecture and chip die photo, where the

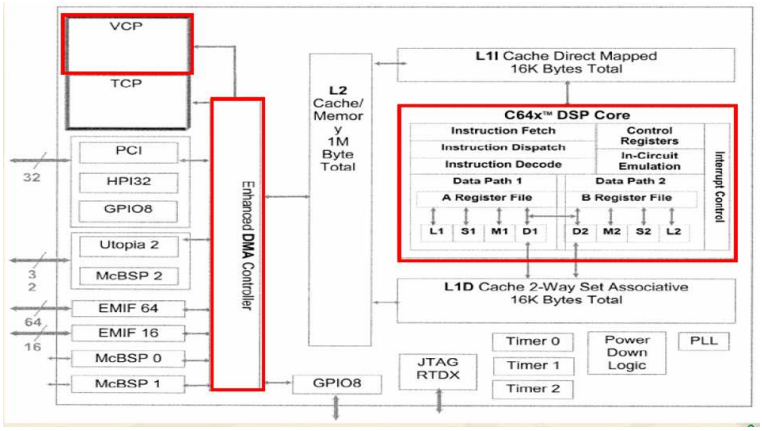


Figure 3.3: DSP chip architecture (from [20]).

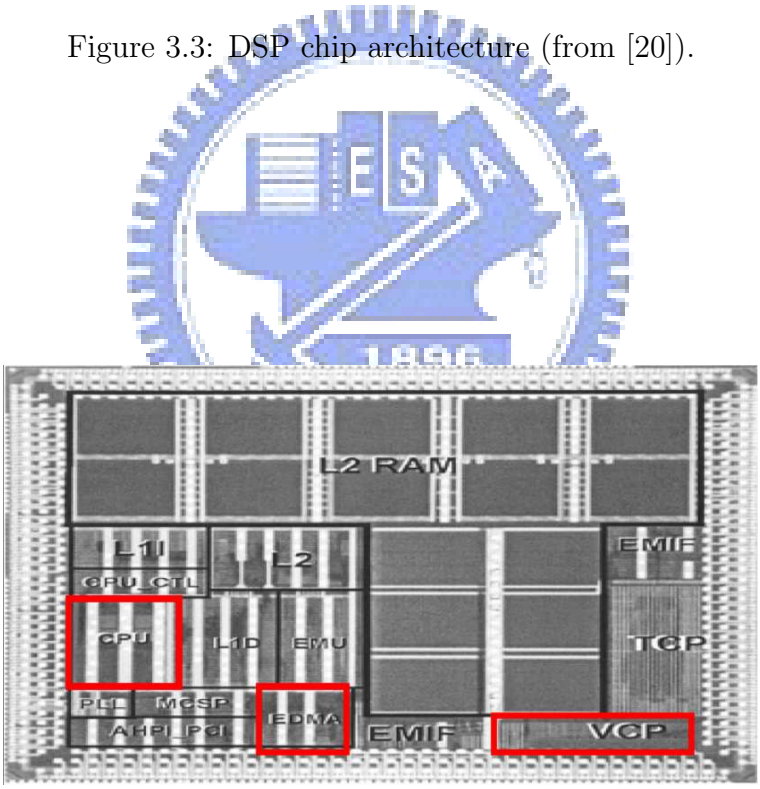


Figure 3.4: DSP chip die (from [20]).

position of the VCP is indicated. The VCP input data are the branch metrics and the output data are the hard or soft decisions. The VCP provides the following features and capabilities:

- Variable constraint length, $K = 5$, 6, 7, or 9.
- User-supplied code coefficients.
- Code rate (1/2, 1/3, or 1/4).
- Configurable trace back settings (convergence distance, frame structure).
- Branch metrics calculation and depuncturing is done in software by the DSP.
- Frees up DSP resources for other processing.
- Communication between the DSP and the VCP is performed through a high performance DMA engine.
- VCP uses its own optimized working memories.

The VCP is able to decode only a subset of the convolutional codes known as single register, nonrecursive convolutional codes (an example is shown in Fig. 3.5). Important parameters for this type of codes are:

- The constraint length K (K = the number of linear finite-state registers + 1).
- The rate R is given by $R = k/n$, where k is the number of information bits needed to produce n output bits known as the codeword.
- The generator polynomials Gn describe how the outputs are generated from the inputs.

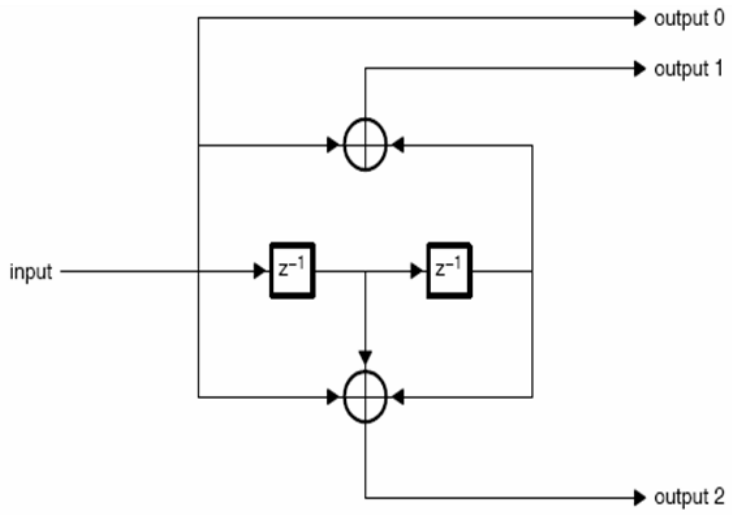


Figure 3.5: Convolutional encoder example, where $K = 3$, $R = 1/3$, $G0 = (100)_8$, $G1 = (101)_8$, $G2 = (111)_8$ (from [19]).

From the parameters, we can derive a trellis diagram providing a useful representation of the code whose complexity grows exponentially with the constraint length K. Fig. 3.6 shows the trellis diagram of the code of Fig. 3.5.

As a maximum-likelihood sequence estimation (MLSE) decoder, the Viterbi decoder identifies the code sequence with the highest probability of matching the transmitted sequence based on the received sequence. The Viterbi algorithm is composed of a metric update and a traceback routine. The metric update performs a forward recursion in the trellis over a finite number of symbol periods where probabilities are accumulated (the VCP accumulates on 12 bits) for each individual state based on the current input symbol (branch metric information). Once a path through the trellis is identified, the traceback routine performs a backward recursion in the trellis and outputs hard or soft decisions.

To facilitate the decoding process, the initial state of delay elements is all zero. In addition, by appending $(K - 1)$ zero tail bits at the end of the F-bit input sequence, it is

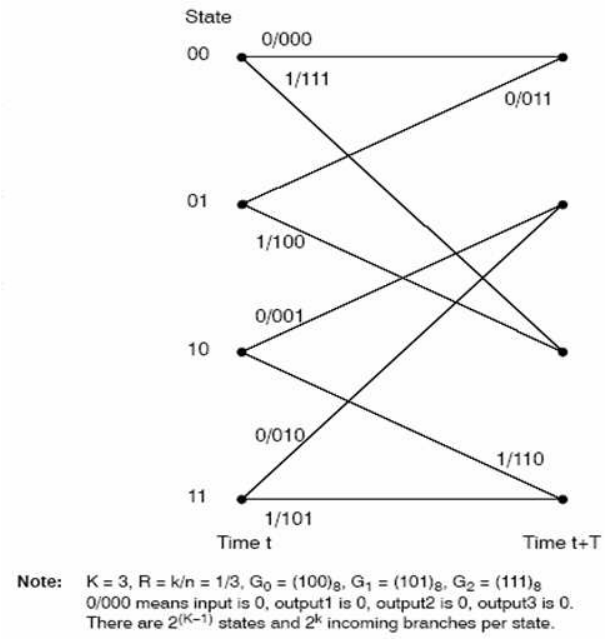


Figure 3.6: Convolutional code trellis example (from [19]).

also ensured that the final state is the all-zero state, which is called zero tail. For example, in Fig. 3.7 the decoded sequence is u_{est} = 0,1,1,1 and the last four zeros in the path are tail bits and not part of the information frame (F). As IEEE 802.16e CC adopts tail-biting, we used to modify the basic way of using VCP to handle it.

3.2.2 VCP Inputs (Brach Metrics and VCP Input Configuration) [19], [22]

BM (Branch Metrics) are calculated by the DSP and stored in the DSP memory subsystem as 7-bit signed values. For rate $1/n$ codes, a total of 2^{n-1} branch metrics need to be computed per symbol period and passed to the VCP.

Consider BPSK modulated bits ($0 \rightarrow 1, 1 \rightarrow -1$), for example. Let the rate be 1/2. Then there are 2 branch metrics per symbol period. We have $BM_0(t) = r_0(t) + r_1(t)$,

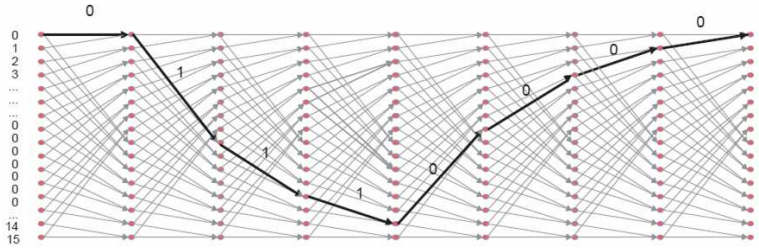


Figure 3.7: Example of survivor path and associated decoded sequence (from [21]).

Table 3.1: Branch Metrics for Rate-1/2 Code

Address (hex)	MSB			LSB
Base	$BM_1(t=T)$	$BM_0(t=T)$	$BM_1(t=0)$	$BM_0(t=0)$
Base + 4h	$BM_1(t=3T)$	$BM_0(t=3T)$	$BM_1(t=2T)$	$BM_0(t=2T)$
Base + 8h	.....			

$BM_1(t) = r_0(t) - r_1(t)$, where $r(t)$ is the received codeword at time t. Note that if we utilize the VCP to decode CC, we must note the definition of the VCP modulation. We find that it may reverse the index of the constellation coordinate for three different modulations.

The data should be sent to the VCP as described in Table 3.1 for rate-1/2 coding (the base address must be double-word aligned). For rate-1/3 and 1/4 coding, the interested reader may refer to [19] for details. The branch metrics can be saved in the DSP memory subsystem in either their native format or packed in words by the user. By default, the VCP works in the little-endian mode, but it can also work in the big-endian, whose detailed settings are discussed in [19].

VCP Input FIFO (Brach Metrics)

The FIFO is used in a double-buffering fashion as shown in Fig. 3.8. The VCP generates a VCPXEVT synchronization event each time the top half or bottom half of the buffier is

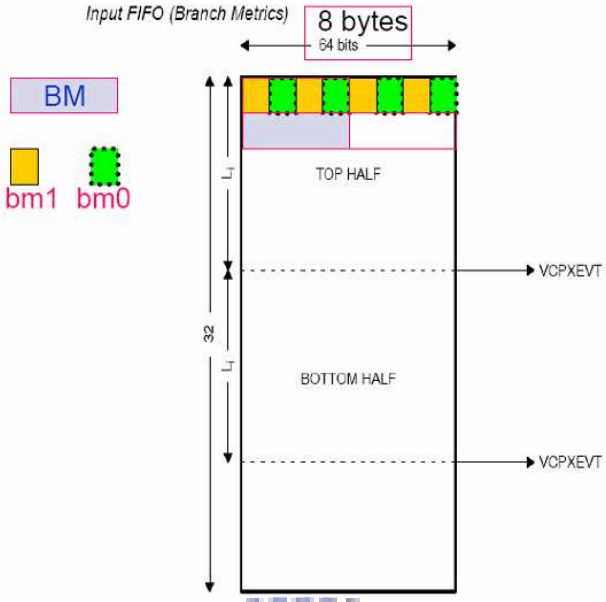


Figure 3.8: VCP input FIFO (modified from [19]).

empty. The SYMX bits are in VCPIC5 and define the buffer length as well as the VCPXEVT event rate. However the SYMX can be automatically determined by parameters such as F, K, and R.

VCP Input Configuration

The VCP contains several memory-mapped registers accessible by the CPU load and store instructions, the QDMA (quick direct memory access), and EDMA. A peripheral-bus access is faster than an EDMA-bus access for isolated accesses (typically when accessing control registers), as shown in Fig. 3.2. EDMA-bus accesses are intended to be used for EDMA transfer and are meant to provide maximum throughput to/from the VCP. The memory map is as shown in Fig. 3.9. Note that the branch metric memory contents are not accessible and the memory can be regarded as FIFOs by the DSP, meaning no need to perform any indexing on the addresses.

Start Address (hex)				
EDMA bus	Peripheral Bus	Acronym	Register Name	Section
5000 0000	01B8 0000	VCPIC0	VCP input configuration register 0	6.1
5000 0004	01B8 0004	VCPIC1	VCP input configuration register 1	6.2
5000 0008	01B8 0008	VCPIC2	VCP input configuration register 2	6.3
5000 000C	01B8 000C	VCPIC3	VCP input configuration register 3	6.4
5000 0010	01B8 0010	VCPIC4	VCP input configuration register 4	6.5
5000 0014	01B8 0014	VCPIC5	VCP input configuration register 5	6.6
5000 0048	01B8 0048	VCPOUT0	VCP output register 0	6.7
5000 004C	01B8 004C	VCPOUT1	VCP output register 1	6.8
5000 0080	–	VCPWBM	VCP branch metrics write register	–
5000 0088	–	VCPRDECS	VCP decisions read register	–
–	01B8 0018	VCPEXE	VCP execution register	6.9
–	01B8 0020	VCPEND	VCP endian mode register	6.10
–	01B8 0040	VCPSTAT0	VCP status register 0	6.11
–	01B8 0044	VCPSTAT1	VCP status register 1	6.12
–	01B8 0050	VCPERR	VCP error register	6.13

Figure 3.9: VCP registers (modified from [19]).

To utilize the VCP, we must first configure the control values, or IC (input configuration) value, which will be sent via the EDMA to program its operation. For this, we may set up the *VCP_Params* structure and pass it to *VCP_icConfig()*. Let *VCP_Params* contain all the channel characteristics required to configure the VCP. We create the object and pass it to the *VCP_genParams()* function which return the *VCP_Params* structure. The input configuration function *VCP_icConfig()* returns a pointer to the IC values which are to be sent using the EDMA. The flow chart shown in Fig. 3.10 explains the working.

3.2.3 VCP Output (Decisions) [19]

The VCP can be configured to send either hard decisions (a bit) or soft decisions (a 16-bit value, 12-bit sign-extended) to the DSP after the decoding.

The decisions buffer start address must be double-word aligned and the buffer size must contain an even number of 32-bit words. The memory map is as shown in Fig. 3.9. Note

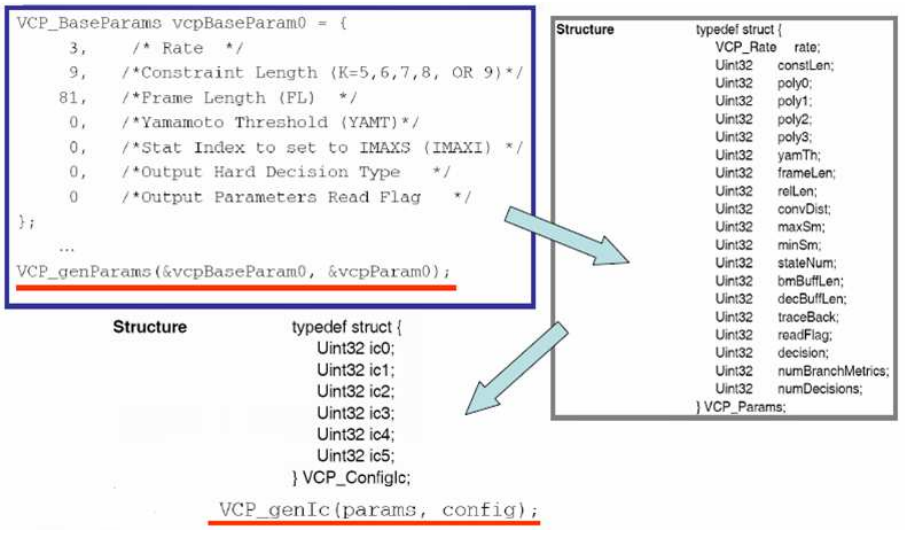


Figure 3.10: VCP configuration structure (modified from [22]).

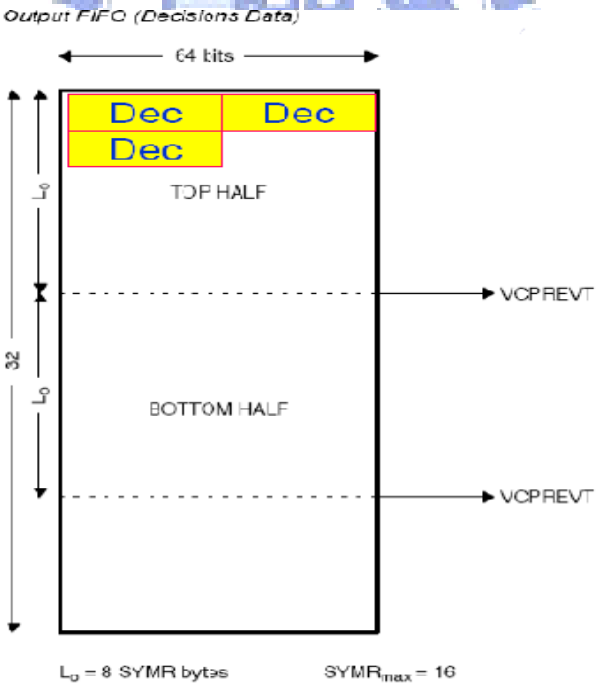


Figure 3.11: VCP output FIFO (modified from [19]).

that the decisions memory contents are not accessible and the memory can be regarded as FIFOs by the DSP, meaning no need to perform any indexing on the addresses.

The FIFO works in a double-buffering manner as depicted in Fig. 3.11, where a "Dec" represents a decisions word (32 bits) in reverse order. The VCP generates a VCPREVT synchronization event each time the top half or bottom half of the buffier is full. The SYMR bits are in VCPIC5 and define the buffer length as well as the VCPREVT event rate. However, the SYMR can be automatically determined by parameters such as F, K, and R.

3.2.4 Sliding Windows Processing [19]

The hard-decision memory can store up to 32,768 traceback bits and there are 2^{K-1} bits stored at each trellis stage. Therefore, the hard-decision memory can store decisions of $32{,}768/2^{K-1}$ symbols. The soft-decision memory can store up to 8,192 traceback soft values and, therefore, contain up to 8,192 soft decisions of $8{,}192/2^{K-1}$ symbols.

Assume a terminated frame of length F (excluding tail bits) and a constraint length K, which determine whether all decisions can be stored in the traceback memories. If all decisions do not fit, then the traceback mode should be set to mixed and the original frame segmented into sliding windows (SW); otherwise, the traceback mode can be set to tailed and no segmentation is required.

In case of a non-terminated frame or if one wants to start decoding without waiting for the end of the frame, the traceback mode should be set to convergent and the frame might have to be segmented into sliding windows depending on whether the decisions will fit in the traceback memories. We only introduce the tailed traceback mode because the our frame length can fit into the VCP memory.

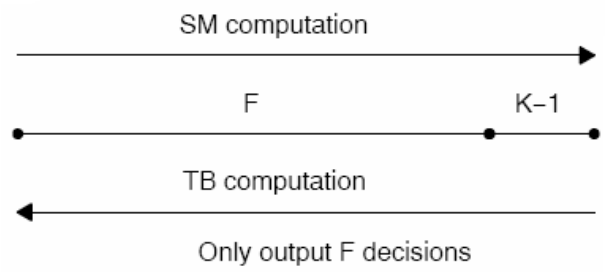


Figure 3.12: VCP tailed traceback mode (from [19]).

Tailed Traceback Mode

This mode is utilized when a full frame can reside within the coprocessor's traceback memory, as shown in Fig. 3.12. The state metrics (SM) are computed over $F+K-1$ symbols, and the traceback (TB) is initialized with the tail state and executed over $F+K-1$ symbol. Only F decisions are output, reversed order. For more information about the mixed traceback mode and convergent traceback mode, refer to [19].

Limitations on F, R, and C

Given a frame of length F (length prior to convolutional encoding with no tail bit information accounted), there are some limitations on the R and C values that one must follow. Unpredictable behavior will occur if these constraints are not observed. The limitations are summarized in Fig. 3.13. Note that we set F to 378 in tailed traceback mode with constraint length equal to 7. The reason will be explained in the next chapter.

3.3 VCP Programming [19], [21]

This section outlines steps required to decode a single frame of data using the VCP. The VCP requires setting up the following context per user channel:

	Traceback Mode					
	Hard Decisions			Soft Decisions		
	Tailed	Mixed/Convergent†		Tailed	Mixed/Convergent†	
K	Fmax	R + C	Possible C values	Fmax	R, C = 3 (K – 1) (non-punctured code)	R, C = 6 (K – 1) (punctured code)
9	120	124	3,6,9,12,15 × (K – 1)	24	R=4, C=24	not allowed
8	217	217	3,6,9,12,15,18 × (K – 1)	49	R=28, C=21	R=7, C=42
7	378	372	3,6,9,12,15,18 × (K – 1)	90	R=60, C=18	R=54, C=36
6	635	605	3,6,9,12,15,18 × (K – 1)	155	R=60, C=15	R=60, C=30
5	2044	1020	3,6,9,12,15,18 × (K – 1)	508	R=60, C=12	R=60, C=24

† Mixed mode is not allowed for frame sizes that can be handled in tailed mode

Figure 3.13: VCP frame, reliability, and convergence length limitations(modified from [19]).

Table 3.2: VCP Required EDMA Links Per User Channel

Direction	Data	Usage	Req/Opt
Transmit	IC parameters	Send the input configuration parameters	Required
Transmit	Branch metrics	Send branch metrics	Required
Receive	Decisions	Read decisions	Required
Receive	Output parameters	Read output parameters	Optional

- 3 to 4 EDMA parameters transfers (see Table 3.2).
- The input configurations parameters.

Several user channels can be programmed prior to starting the VCP. A suggested implementation is to use the EDMA interrupt generation capabilities and program the EDMA to generate an interrupt after the last VCPREVT synchronized EDMA transfer of the user channel has completed.

3.3.1 Prepare Input Configuration, Initialize Input Buffers, and Allocate Output Buffers [21]

Prepare Input Configuration

For each frame, the VCP input configuration register VCPIC0–VCPIC5 are programmed as described before [19]. The register configuration is first prepared in the DSP memory (internal or external). It is transferred to the VCP via EDMA once the VCP is started. The DSP memory address of the beginning of the prepared input configuration is denoted as &input_config[0].

Initialize Input Buffers

The user computes branch metrics and store them in DSP internal or external memory. For terminated frame with F information bits and code with constraint length K, the total number of symbols is $N = F + K - 1$. For non-terminated frame, i.e., no tail bits, the total number of input symbol is $N = F$.

For rate r and constraint length K code, there will be $N \times 2^{(1/r)-1}$ 7-bit branch metrics. The DSP memory address of the beginning of the pre-computed branch metrics array will be referred to as &bm[0]. The beginning of the branch metric array should be aligned on a 64-bit boundary.

Allocate Output Buffers

Hard decisions are transferred from the VCP in 64-bit words, stored in a bit-packed manner. Therefore, for a frame with F information bits, the size of the allocated output buffer should be $\lceil F/64 \rceil \times 8$ bytes. For soft-decision decodign, refer to [21].

If the output parameter read flag is set (OUTF = 1), two additional 64-bit words should be allocated for the output parameter word. The DSP memory address of the beginning

of the allocated buffers for VCP decisions and output parameter will be referred to as &hard_decision[0] and &output_parameter[0], respectively. All buffers should be aligned on an 64-bit boundary.

3.3.2 EDMA Resource [19]

Within the available 64 EDMA channel event sources, two are assigned to the VCP: event 28 (RX) and event 29 (TX). Event 28 is associated to the VCP receive event (VCPREVT) and is used as the synchronization event for EDMA transfers for the VCP to the DSP (receive). Event 29 is associated to the VCP transmit event (VCPXEVT) and is used as the synchronization event for EDMA transfers for the DSP to the VCP (transmit).

The EDMA parameters comprise six words as shown in Fig. 3.14. All EDMA transfers, in the context of the VCP, must be done using 32-bit word elements, must contain an even number of words, and must have sources and destination addresses double-word aligned.

The element count for the VCP EDMA transfer must be a multiple of 2. Single-word transfers that are not double-word aligned cause unexpected errors in VCP memory. For more information about EDMA input configuration parameter transfer, branch metrics transfer, and hard-decisions mode, refer to the good tutorial in [19].

3.3.3 VCP Procedure [21]

Start EDMA

The EDMA channels corresponding to VCPREVT and VCPXEVT are enabled in the EDMA Event Enable Register (EER), and these channels are also allowed to generate CPU interrupts by setting appropriate bits in the Channel Interrupt Enable Register (CIER). The EDMA control registers are described in detail in [23].

(a) EDMA Registers

	31	0	EDMA parameter
Word 0	EDMA Channel Options Parameter (OPT)		OPT
Word 1	EDMA Channel Source Address (SRC)		SRC
Word 2	Array/frame count (FRMCNT)	Element count (ELECNT)	CNT
Word 3	EDMA Channel Destination Address (DST)		DST
Word 4	Array/frame index (FRMIDX)	Element index (ELEIDX)	IDX
Word 5	Element count reload (ELERLD)	Link address (LINK)	RLD

(b) EDMA Channel Options Parameter (OPT)

31–29	28–27	26	25–24	23	22–21	20	19–16
PRI	ESIZE	2DS	SUM	2DD	DUM	TCINT	TCC

15	14–13	12	11	10–5	4	3	2	1	0
—	TCCM	ATCINT	—	ATCC	—	PDTS	PDTD	LINK	FS

Figure 3.14: VCP EDMA parameters structure (from [19]).

Start VCP

CPU writes a "START" command into the execution word register VCPEXE of the VCP. This causes the VCP to generate the first VCPXEVT expecting input control. This in turn triggers the the EDMA transfer which is programmed into the Event PaRAM location corresponding to VCPXEVT.

Service EDMA Interrupt from VCP Channel at the End of Decoding

The EDMA link associated with the last VCPREVT is configured to generate a CPU interrupt. In the CPU interrupt service routine, the output decision buffer for the completed frame can be processed and decoding of next frame can be initiated.

3.4 EDMA under the Code Composer Studio (CCS) [23]

To utilize the VCP, we must understand how to use the EDMA. Under the CCS, we utilize the CSL (Chip Support Library) functions provided by TI CCS to help use of EDMA. For convenience, we name it CCS EDMA. The following text is mainly taken from [23].

The EDMA controller handles all data transfers between the level-two (L2) cache/memory controller and the device peripherals on the C621x/C671x/C64x. These data transfers include cache servicing, non-cacheable memory accesses, user-programmed data transfers, and host accesses.

The EDMA controller comprises:

- event and interrupt processing registers,
- event encoder,
- parameter RAM, and
- address generation hardware.

A block diagram of the EDMA controller is shown in Fig. 3.15.

EDMA events are captured in the event register. An event is a synchronization signal that triggers an EDMA channel to start a transfer. If events occur simultaneously, they are resolved by way of the event encoder. The transfer parameters corresponding to this event are stored in the EDMA parameter RAM, and passed onto the address generation hardware, which address the EMIF (External Memory Interface) and/or peripherals to perform the necessary read and write transactions.

In the following subsections, the CCS EDMA is introduced in six parts: EDMA control

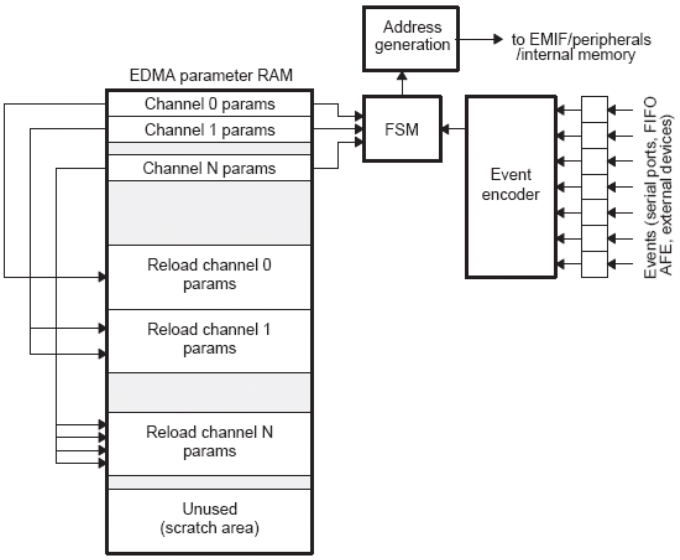


Figure 3.15: EDMA control (from [23]).

registers, parameter RAM (PaRAM), EDMA transfer parameter entry, initiating an EDMA transfer, linking EDMA transfers, and EDMA interrupt generation.

3.4.1 EDMA Control Registers [23]

Each of the 64 channels (C64x) or 16 channels (C621x/C671x) in the EDMA has a specific synchronization event associated with it. These events trigger the data transfer associated with that channel. The list of control registers that perform various processing of events is shown in Table 3.3. We introduce the most important registers for our work below.

Event Registers (ER, ERL, ERH)

All events are captured in the event register (ER), even when the events are disabled. The C621x/C671x has only one event register (ER). The C64x has two event registers, event low register (ERL) and event high register (ERH) for the 64 channels.

Table 3.3: EDMA Control Registers (Modified from [23])

Byte Address	Acronym	Register Name
01A0 FF9Ch	EPRH	Event polarity high register (C64x only)
01A0 FFA4h	CIPRH	Channel interrupt pending high register(C64x only)
01A0 FFA8h	CIERH	Channel interrupt enable high register(C64x only)
01A0 FFACh	CCERH	Channel chain enable high register (C64x only)
01A0 FFB0h	ERH	Event high register (C64x only)
01A0 FFB4h	EERH	Event enable high register (C64x only)
01A0 FFB8h	ECRH	Event clear high register (C64x only)
01A0 FFBCh	ESRH	Event set high register (C64x only)
01A0 FFC0h	PQAR0	Priority queue allocation register 0 (C64x only)
01A0 FFC4h	PQAR1	Priority queue allocation register 1 (C64x only)
01A0 FFC8h	PQAR2	Priority queue allocation register 2 (C64x only)
01A0 FFCCh	PQAR3	Priority queue allocation register 3 (C64x only)
01A0 FFDCh	EPRL	Event polarity low register (C64x only)
01A0 FFE0h	PQSR	Priority queue status register
01A0 FFE4h	CIPR	Channel interrupt pending register (C621x/C671x)
	CIPRL	Channel interrupt pending low register(C64x)
01A0 FFE8h	CIER	Channel interrupt enable register (C621x/C671x)
	CIERL	Channel interrupt enable low register (C64x)
01A0 FFECh	CCER	Channel chain enable register (C621x/C671x)
	CCERL	Channel chain enable low register (C64x)
01A0 FFF0h	ER	Event register (C621x/C671x)
	ERL	Event low register (C64x)
01A0 FFF4h	EER	Event enable register (C621x/C671x)
	EERL	Event enable low register (C64x)
01A0 FFF8h	ECR	Event clear register (C621x/C671x)
	ECRL	Event clear low register (C64x)
01A0 FFFCh	ESR	Event set register (C621x/C671x)
	ESRL	Event set low register (C64x)

Event Enable Registers (EER, EERL, EERH)

In addition to the event register, the EDMA controller also provides the user the option of enabling/disabling events. Any of the event bits in the event enable register can be set to "1" to enable that event. The C621x/C671x has only one event enable register (EER). The C64x has two event enable registers, event enable low register (EERL) and event enable high register (EERH) for the 64 channels.

All events that are captured by the EDMA are latched in the ER even if that event is disabled. This is analogous to an interrupt enable and interrupt-pending register for interrupt processing. This ensures that no events are dropped by the EDMA. Thus, re-enabling an event with a pending event signaled in the ER forces the EDMA controller to process that event according to its priority. Writing a "0" to the corresponding bit in the EER disables an event.

Event Clear Registers (ECR, ECRL, ECRH)

Once an event has been posted in the ER, the event can be cleared in two ways. If the event is enabled in the event enable register (EER), the corresponding event bit in the ER is cleared as soon as the EDMA submits a transfer request for that event. Alternatively, if the event is disabled in the EER, the CPU can clear the event by way of the event clear register (ECR). This feature allows the CPU to release a lock-up or error condition. Therefore, once an event bit is set in the ER, it remains set until the EDMA submits a transfer request for that event or the CPU clears the event by setting the relevant bit in the ECR.

Event Set Registers (ESR, ESRL, ESRH)

The CPU can also set events by way of the event set register (ESR). Writing a "1" to one of the event bits causes the corresponding bit to be set in the event register. The event does

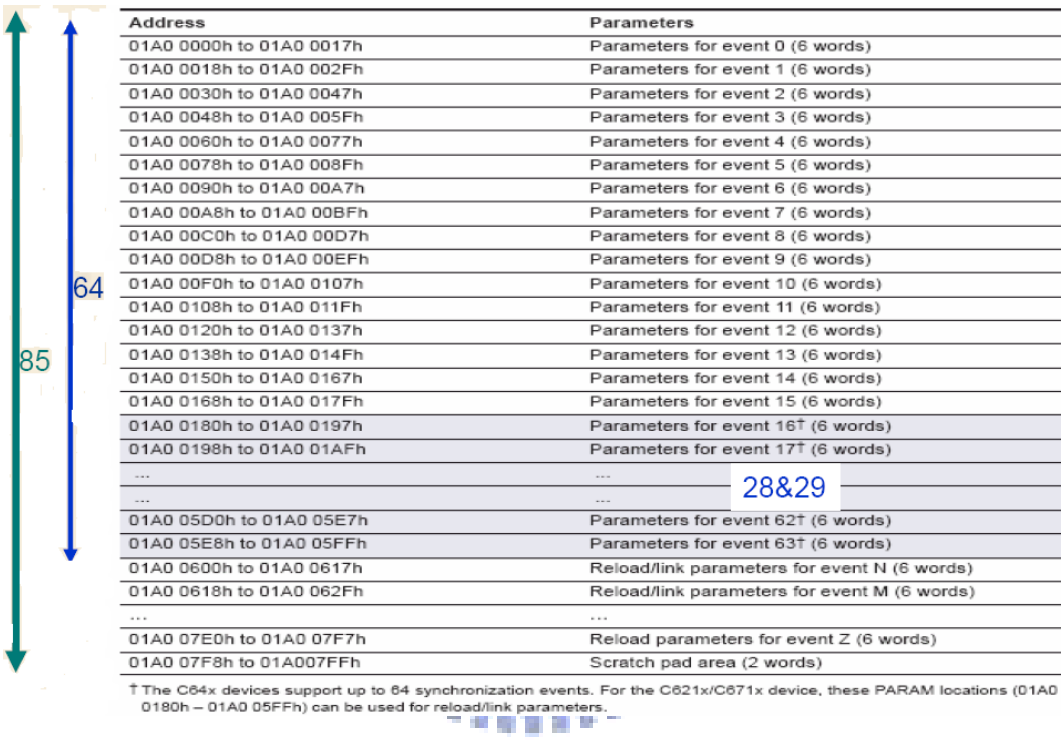
not have to be enabled in this case. This provides a good debugging tool and also allows the CPU to submit EDMA requests in the system. Note that such CPU-initiated EDMA transfers are basically unsynchronized transfers. In other words, an EDMA transfer occurs when the relevant ESR bit is set and is not triggered by the associated event.

3.4.2 Parameter RAM (PaRAM) [23]

Unlike the C620x/C670x DMA controller, which is a register-based architecture, the EDMA controller is a RAM-based architecture. EDMA channels are configured in a parameter table. The table is a 2-Kbyte block of internal parameter RAM (PaRAM) located within the EDMA. The table consists of six-word parameter sets (entries), for a total of 85 entries. The contents of the 2-Kbyte PaRAM, shown in Fig. 3.16, comprises:

- For C621x/C671x there are 16 transfer parameter entries for the 16 EDMA events. For C64x, there are 64 transfer parameter entries for the 64 EDMA events. Each entry is six words or 24 bytes.
- Remaining transfer parameter sets are used for linking transfers. Each set or entry is 24 bytes.
- 8 bytes of unused RAM can be used as scratch pad area. Note that a part or entire EDMA RAM can be used as a scratch pad RAM provided the event(s) this area corresponds to is/are disabled. It is the user's responsibility to provide the transfer parameters when the event is eventually enabled.

Once an event is captured, its parameters are read from one of the top 64 entries (C64x) or 16 entries (C621x/C671x) in the PaRAM. These parameters are then sent to the address generation hardware.



Address	Parameters
01A0 0000h to 01A0 0017h	Parameters for event 0 (6 words)
01A0 0018h to 01A0 002Fh	Parameters for event 1 (6 words)
01A0 0030h to 01A0 0047h	Parameters for event 2 (6 words)
01A0 0048h to 01A0 005Fh	Parameters for event 3 (6 words)
01A0 0060h to 01A0 0077h	Parameters for event 4 (6 words)
01A0 0078h to 01A0 008Fh	Parameters for event 5 (6 words)
01A0 0090h to 01A0 00A7h	Parameters for event 6 (6 words)
01A0 00A8h to 01A0 00BFh	Parameters for event 7 (6 words)
01A0 00C0h to 01A0 00D7h	Parameters for event 8 (6 words)
01A0 00D8h to 01A0 00EFh	Parameters for event 9 (6 words)
01A0 00F0h to 01A0 0107h	Parameters for event 10 (6 words)
01A0 0108h to 01A0 011Fh	Parameters for event 11 (6 words)
01A0 0120h to 01A0 0137h	Parameters for event 12 (6 words)
01A0 0138h to 01A0 014Fh	Parameters for event 13 (6 words)
01A0 0150h to 01A0 0167h	Parameters for event 14 (6 words)
01A0 0168h to 01A0 017Fh	Parameters for event 15 (6 words)
01A0 0180h to 01A0 0197h	Parameters for event 16† (6 words)
01A0 0198h to 01A0 01AFh	Parameters for event 17† (6 words)
...	...
...	...
01A0 05D0h to 01A0 05E7h	Parameters for event 62† (6 words)
01A0 05E8h to 01A0 05FFh	Parameters for event 63† (6 words)
01A0 0600h to 01A0 0617h	Reload/link parameters for event N (6 words)
01A0 0618h to 01A0 062Fh	Reload/link parameters for event M (6 words)
...	...
01A0 07E0h to 01A0 07F7h	Reload parameters for event Z (6 words)
01A0 07F8h to 01A007FFh	Scratch pad area (2 words)

† The C64x devices support up to 64 synchronization events. For the C621x/C671x device, these PARAM locations (01A0 0180h – 01A0 05FFh) can be used for reload/link parameters.

Figure 3.16: EDMA parameter RAM contents (modified from [23]).

Offset Address (bytes)	Parameter	As defined for...	
		1-D transfer	2-D transfer
0	Options	Transfer configuration options.	
4	Source address	The address from which data is transferred.	
8	Element count	The number of elements per frame.	The number of elements per array.
10	Frame count (1D), Array count (2D)	The number of frames per block minus one.	The number of arrays per frame minus one.
12	Destination address	The address to which data is transferred.	
16	Element index	The address offset of elements within a frame.	—
18	Frame index (1D), Array index (2D)	The address offset of frames within a block.	The address offset of arrays within a frame.
20	Link address	The PaRAM address containing the parameter set to be linked.	
22	Element count reload	The count value to be loaded at the end of each frame.†	—

Figure 3.17: EDMA channel parameters (from [23]).

3.4.3 EDMA Transfer Parameter Entry [23]

Each parameter entry of an EDMA event is organized in six 32-bit words or 24 bytes as shown in Fig. 3.14. Access to the EDMA parameter RAM is provided only via the peripheral bus. These parameters are shown in Fig. 3.17. For more information, see [23].

3.4.4 Initiating an EDMA Transfer [23]

There are two ways to initiate data transfer using the EDMA. One is CPU-initiated EDMA and the other is an event-triggered EDMA. The latter is a more typical usage of the EDMA. This allows the submission of transfer requests to occur automatically based on system events, without any intervention by the CPU. CPU-initiated transfer is included in the design for added control and robustness. Each EDMA channel can be started independently. The CPU can also disable an EDMA channel by disabling the event associated with that channel.

- CPU-initiated EDMA or unsynchronized EDMA: The CPU can write to the event set register, ESR, in order to start an EDMA transfer. Writing a "1" to the corresponding event in the ESR triggers an EDMA event. Just as with a normal event, the transfer

parameters in the EDMA parameter RAM corresponding to this event are passed to the address generation hardware, which performs the requested access of the EMIF, L2 memory or peripherals, as appropriate. CPU-initiated EDMA transfers are unsynchronized data transfers. The events enable bit does not have to be set in the EER for CPU-initiated EDMA transfers. This is because a CPU write to the ESR is treated as a real-time event.

- Event-triggered EDMA: An event that is latched in the event register, ER, via the event encoder causes its transfer parameters to be passed on to the address generation hardware, which performs the requested accesses. Although the event causes this transfer, it is very important that the event itself be enabled by the CPU. Writing a"1" to the corresponding bit in EER enables an event. Alternatively, an event is still latched in the ER even if its corresponding enable bit in EER is "0" (disabled). The EDMA transfer related to this event occurs as soon as it is enabled in EER. In addition to event enable via EER, the completion of a transfer can also trigger another EDMA transfer through chaining and the CCER.

For more information about synchronization of EDMA transfers, see [23].

3.4.5 Linking EDMA Transfers [23]

The EDMA controller provides linking, a feature especially useful for complex sorting, circular buffering type of applications. If LINK = 1, upon completion of a transfer, the EDMA link feature reloads the current transfer parameters with the parameter pointed to by the 16-bit link address. The entire EDMA parameter RAM is located in the 01A0 xxxxh area. Therefore the 16-bit link address, which corresponds to the lower 16-bit physical address, is sufficient to specify the location of the next transfer entry. The link address must be aligned on a 24-byte boundary. An example of a linked EDMA transfer is shown in Fig. 3.18. Note

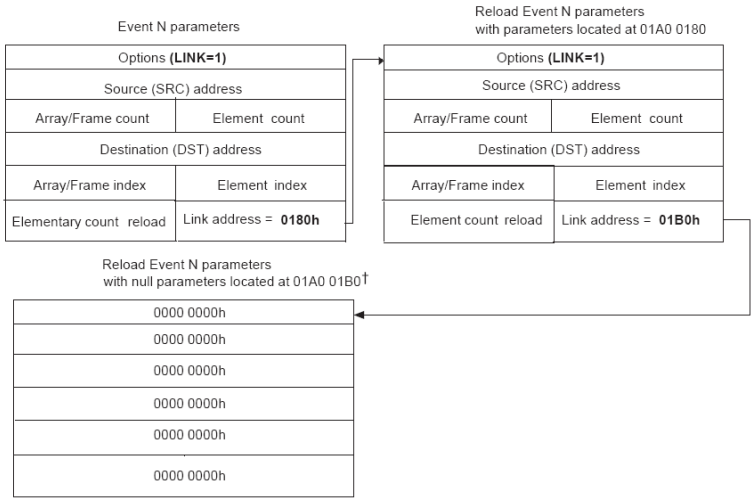


Figure 3.18: Example of linked EDMA transfers (from [23]).

that the last transfer parameter entry should have its LINK = 0 so that the linked transfer stops after the last transfer. That is the last entry should be linked to a NULL parameter set.

3.4.6 EDMA Interrupt Generation [23], [24]

The EDMA control is resposible for generating transfer-completion interrupts to the CPU. The EDMA generates a single interrupt (EDMA_INT) to the CPU on behalf of all 16 channel (C621x/C671x) or 64 channel (C64x). The various control and bit fields facilitate EDMA interrupt generation.

When TCINT bit options entry is set to "1" for an EDMA channel and a specific transfer complete code is provided, the EDMA controller sets a bit in the channel interrupt pending register (CIPR). Lastly, the important action is to generate the EDMA_INT to the CPU. To do this, the corresponding interrupt enable bit should be set in the channel interrupt enable register (CIER). To configure the EDMA for any channel (or QDMA request) to interrupt the CPU:

- Set CIEn to “1” in the CIER.
- Set TCINT to “1” in channel options.
- Set Transfer Complete Code to n in channel options.

Note that if the CIER bit is disabled, the channel completion event is still registered in the CIPR if its TCINT = 1. Once the CIER bit is enabled, the corresponding channel interrupt is sent to the CPU. If the CPU interrupt (defaults to CPU_INT8) is enabled, its ISR (Interrupt Service Routines) is executed. More than one QDMA/EDMA channel can use the same TCC value, and the TCC value is not required to be equal to the channel number [24].

In the C621x/C671x, the transfer complete code is specified in the TCC field, with values between 0000b to 1111b. In the C64x, which has a total of 64 channels, the transfer complete code is expanded to a 6-bit value that accomodates the 64 channels. The 6-bit transfer complete code of the C64x is comprised of the new TCCM bits (most significant bits of the transfer complete code), in addition to the TCC field in the options parameter.

EDMA Interrupt Servicing by the CPU

Since the EDMA controller is aware of when the EDMA channel transfer is complete, it sets the appropriate bit in the CIPR as per the transfer complete code specified by the user. The CPU ISR should read the CIPR and determine what, if any events/channels have completed and perform the operations necessary. The ISR should clear the bit in CIPR upon servicing the interrupt, therefore enabling recognition of further interrupts. Writing a “1” to the relevant bit can clear CIPR bits, writing a “0” has no effect. By the time one interrupt is serviced, many others could have occurred and relevant bits set in CIPR. Each of these bits in CIPR would probably need different types of service. The ISR should check for all

pending interrupts and continue until all the posted interupts are serviced.

3.5 EDMA under the 3L Diamond Real-Time Operating System

In this section, we describe the operation of EDMA under the 3L Diamond because our VCP implementation is part of an implementation that uses one or more DSPs running on 3L Diamond. For convenience, we call it 3L EDMA to distinguish it from CCS EDMA. Notice that 3L EDMA is functionally equivalent to CCS EDMA. They are different only in called libraries and header files.

3.5.1 Introduction to 3L Diamond

Diamond is 3L company's system for multiprocessor software design and implementation. Diamond uses the communicating sequential processes (CSP) model to give a simple but powerful way of developing applications that make use of one or more processors [25]. The 3L Ltd has been working closely with Sundance, aiming to provide simple-to-use, reliable and flexible development environment for the Sundance hardware.

The way to build and run applications using Diamond differs substantially from the more traditional techniques used in other environments, particularly the CCS. The CCS has been designed to produce applications for single processor systems; multiprocessor systems are seen as several separate applications that happen to be executed at the same time. Diamond takes the opposite view and considers multiprocessor systems as an integrated whole [26].

3.5.2 SC6xEDMA [26]

The Diamond kernel manages the available EDMA channels and dynamically allocates them to concurrently active inter-processor <chan.h> and <link.h> calls. User code that wants

to make direct use of the channels must claim that from the kernel, complete the DMA operation, and return the channels to the kernel. Holding on to EDMA channels can seriously affect the performance of other transfers, in particular, link operations.

There is an example about using EDMA to copy "Frames" (blocks of 8 32-bit words) from a device FIFO to memory at "Buffer" in [26]. The example is not complete, too simple, and cannot be directly used, but it can help one to understand 3L EDMA better. The interested reader is referred to [26].

There are several things to notice about 3L EDMA (modified from [26]):

- <edma.h> declares the kernel functions used in the rest of the code and creates a reference, _kernel, to kernel data structures. It also contains a typedef for a structure type, EDMA_REG, which can be used to access the EDMA transfer parameters, plus macros for accessing the various fields within the EDMA registers. EDMA_CTRL is also defined to be a pointer to the hardwares block of EDMA control registers. They may be used as follows:
  #include <edma.h>, and
  struct EDmaControl *C = EDMA_CTRL; // EDMA control registers.
- EdmaI = $SC6xKernel\_LocateInterface$(_kernel, SIID_SC6xEDMA);
  - It would not call $SC6xKernel\_LocateInterface$ for every transfer.
  - It would initiate the interface pointers once on program startup.
- dma = $SC6xEDMA\_Claim$(EdmaI, 4, &channel);
  - It returns a pointer to the corresponding EDMA transfer parameters, or NULL if the requested EDMA engine cannot be allocated (because it is already claimed by another thread or by the kernel for an inter-processor link communication).

- If $SC6xEDMA\_Claim$ succeeds, it returns an $SC6xEDMAChannel$ pointer via its final argument. This pointer refers to a software structure in the kernel that describes the allocated EDMA channel.

- $SC6xEDMAChannel\_StartWait$(channel); // do the transfer
  - $SC6xEDMAChannel\_StartWait$ is one of the functions that can be applied to such an EDMA channel pointer. It sets up the various EDMA control registers needed to control the transfer and then suspends the calling thread until the EDMA channel interrupts at the end of the block.
  - While the thread is suspended and the EDMA operation is executing, other threads can continue to execute on the CPU. The kernel will catch the EDMA completion interrupt, resume the suspended thread and return control to the caller.

- $SC6xEDMAChannel\_Release$(channel);
  - $SC6xEDMAChannel\_Release$ informs the kernel that a previously claimed EDMA channel is no longer required and can be returned to the kernels pool of free channels.

There is no obligation to use $SC6xEDMAChannel\_StartWait$, which is provided to make handling EDMA interrupts easier, but one is free to wait for EDMA completion either by polling (not recommended) or waiting for the interrupt with $SC6xEDMAChannel\_AwaitInterrupt$. The only mandatory step is to claim the EDMA channel before attempting to touch the corresponding hardware [26].

3.5.3 EDMA Channel Availability [26]

Different C6x processors provide different numbers of EDMA channels: the C64 has 64 while other processors have 16. As it is highly unlikely that many applications will require large numbers of EDMA channels, Diamond usually arranges for the first 16 to be made available. This minimizes the amount of memory needed to support EDMA and has proved to be adequate for the kernel and the most users. However, if one does need more than 16 channels, one can request 32, 48, or the full 64. This is done by defining a new processor type and using the "MAP=" qualifier to identify the appropriate EDMA handler module. For example, to create a variant of an existing processor type "MyProc" with 64 EDMA channels one could define a new processor type as follows:

Proctype MyProc64 MyProc MAP = DMA:EDMA64

The definition should be defined in the configuration file (xxx.cfg).

3.5.4 SC6xEDMAChannel Functions [26]

These functions all operate on one of the SC6xEDMAChannel pointers returned by the "claim" functions described above. Functions dealing with external devices do not set the various device enables that are necessary to allow EDMA synchronization or CPU interrupts. One needs to refer to the C6000 modules hardware documentation for a description of enabling events and interrupts for particular devices. EDMA termination interrupts are automatically managed. The SC6xEDMAChannel functions include the following:

- $SC6xEDMAChannel\_Release$.
- $SC6xEDMAChannel\_ResetEvent$.

- *SC6xEDMAChannel_AwaitInterrupt*: Each SC6xDMAChannel has an EVENT synchronisation object associated with it. The kernel catches interrupts from the underlying hardware EDMA channel (*EDMA_INT*) and arranges for the appropriate event to be signalled. This function suspends the calling thread until that event is signalled. One should clear the event *SC6xEDMAChannel_ResetEvent* before setting up the transfer and waiting for the interrupt.

- *SC6xEDMAChannel_Start*: This function starts an EDMA transfer by setting the appropriate bit in ESR.

- *SC6xEDMAChannel_StartWait*: This function assumes that the actual transfer will be initiated by the synchronisation event associated with the EDMA channel being used. One should call *SC6xEDMAChannel_KickWait* when one wants the transfer to start immediately. It encapsulates the following sequence:

  – *SC6xEDMAChannel_ResetEvent*(channel);

  – Set the bits in CIER and EER corresponding to the given channel;

  – *SC6xEDMAChannel_AwaitInterrupt*(channel);

  – Clear the bits in CIER and EER corresponding to the given channel.

- *SC6xEDMAChannel_KickWait*: This function is provided for the common case where the EDMA channel does not need to wait for a synchronisation signal before initiating a transfer. It encapsulates the sequence:

  – *SC6xEDMAChannel_ResetEvent*(channel);

  – Set the bits in CIER and EER corresponding to the given channel;

  – Set the bit in ESR corresponding to the given channel to start the transfer;

  – *SC6xEDMAChannel_AwaitInterrupt*(channel);

– Clear the bits in CIER and EER corresponding to the given channel.

Note that $SC6xEDMAChannel\_KickWait$ is different from $SC6xEDMAChannel\_StartWait$ by setting of ESR. For more information, see [26].

Chapter 4

DSP Implementation of Convolutional Encoder and Decoder

In this chapter, we consider DSP implementation of the convolutional encoder and decoder, especially that employing the VCP. The simulation results provide information concerning proper choices of certain design parameters, such as F (frame) and the amount of circular shift in the tail-biting CC decoder. We discuss how to use the VCP, and we compare them with the fixed-point C program computation results with and without using the VCP.

For the purpose of overall transmission system integration, we also consider running the VCP under the 3L RTOS. We present the BER performance obtained using TI's Code Composer Studio (CCS) tool set and the data rate results under CCS and under 3L. Fig. 4.1 shows the overall encoder and decoder structure with CC decoding executed on VCP. Our implementation is based on modification of the code of Wu [3] for IEEE 802.16e OFDMA convolutional coding and decoding.

4.1 VCP Parameter Setting

In this section, we introduce how to set the VCP's important parameter for WiMAX CC with tail-biting. Since the VCP control and data transmission must be done through EDMA,

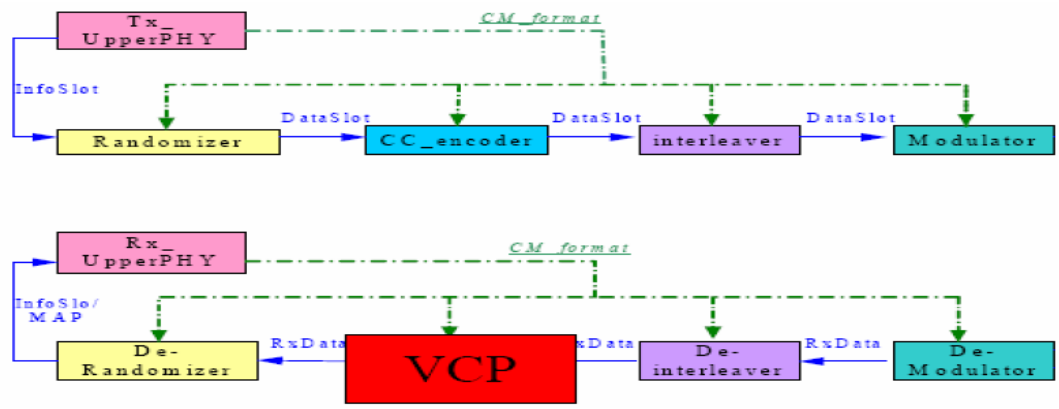


Figure 4.1: CC encoding and decoding with VCP.

we have introduced the EDMA in chapter 3.

4.1.1 Generator Polynomials

The VCP has been designed for IS2000 and 3GPP wireless applications. We has find that the generator polynomials $(171_{OCT}, 133_{OCT})$ for the CC in IEEE 802.16e to be the same as in the 3GPP standard, IEEE 802.11, and DVB standard.

In VCP, the generator polynomial (Gn) can be set by specifying the constraint length (K) and code rate (R). For IEEE 802.16e, we can set $K = 7$ and $R = 1/2$ for its CC. Note that same codes may define the two generator polynomials in reverse order relative to that of the 3GPP. The user has to pay attention to this situation. But in IEEE 802.16e, the order is the same.

4.1.2 EDMA Setting

Fig. 4.2 shows the EDMA transfer parameters for VCP.

The third row in the table gives the address in the PaRAM. Link 0 of both VCPXEVT and VCPREVT have to be set to fixed locations in the PaRAM denoted as ADDR_VCPXEVT

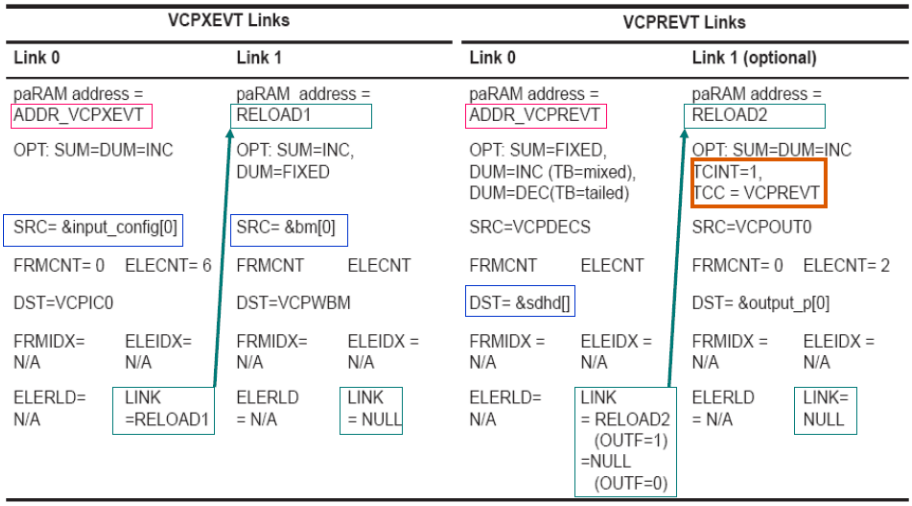


Figure 4.2: VCP parameter setting (modified from [21]).

and ADDR_VCPREVT, respectively. Other links can point to anywhere in the PaRAM. These additional locations in the PaRAM are denoted RELOAD1, RELOAD2. The LINK entry in each parameter set gives the PaRAM address of the next linked transfer. Setting LINK = NULL indicates that the next transfer, that is, the EDMA transferred is terminated [21].

4.1.3 Tail-Biting

Because the CC in IEEE 802.16e is a tail-biting one whereas the VCP does tailed traceback, we need to modify the basic Viterbi decoding flow of the the VCP to accommodate this situation. As a result of the VCP frame (F) limitations with K and the tail-biting relationship, we choose the frame maximum value 378 ($= 288 + 90$) as F, as shown in Fig. 3.13. Base on [2], [3], we shift some bits at the end of the decoded sequence to replace the bits at the beginning of the sequence which are more prone to error due to tail-biting. The suitable member of such circularly shifted bits (CSB) has been determined by experiments as 51 to 58. A sketch of how the above works is shown in Fig. 4.3.

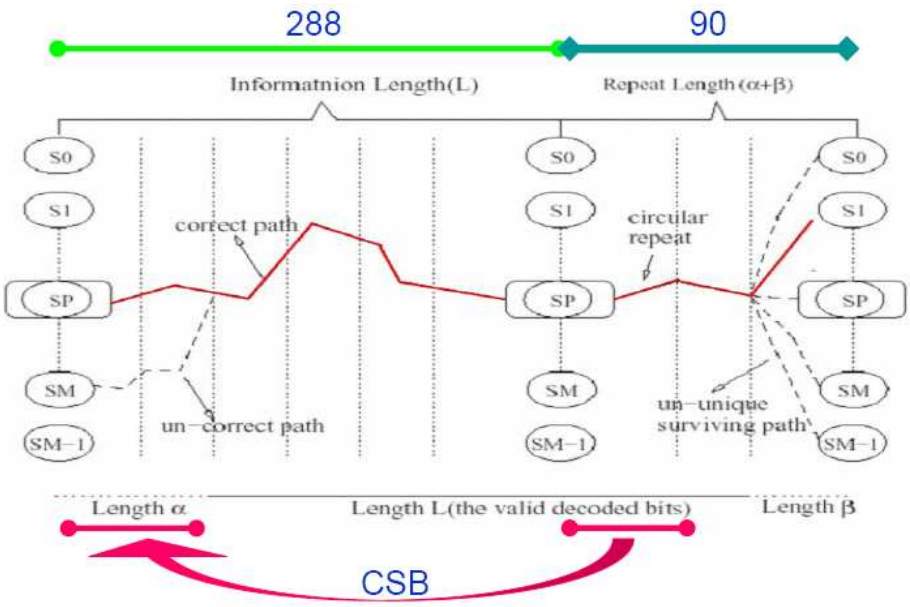


Figure 4.3: Tail-biting CC decoding employing (modified from [3]).

4.2 Coding Gain Analysis [3]

The contents of this section have been taken to a large extent from [3]. In this section, we analyze the convolutional coding gains to obtain a 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

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

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 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 the 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 and R_c is the code rate. As a result, the lower bound on E_b/N_0 is given by

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

The upper-bound coding gain 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 bounds of the seven coding-modulation schemes in IEEE 802.16e in Table 4.1.

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

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

Table 4.1: Coding Gain Upper Bounds 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

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 to 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.

4.3 Comparison of Performance in AWGN of VCP and Wu's Viterbi Decoder

In this section, we present the simulated performance of convolutional decoding performance in AWGN based on the system structure shown in Fig. 4.1 that uses VCP. We also compare Wu's fixed-point Viterbi decoder without using the VCP [3].

Table 4.2: Approximate Coding Gains 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

In order to implement the VCP, we need to have the input data in the fixed-point format. Since the BM are calculated by the DSP and stored in the DSP memory subsystem as 7-bit signed values, we must do necessary rounding or truncation of the decoder input to make them 7-bit signed values.

Our simulations considered different placements of the binary point in the 7 bit BM values. We also simulate Wu's design without using the VCP, which quantizes decoder input to 16 bits. In the case of Wu's decoder, two placements of the binary points are considered, namely, S9.6 and S11.4, where Sa.b means there are a integer bits and b fractional bits, plus a sign bit.

An interesting thing is that our BM array must be declared "unsigned char" but not "char" for the VCP to operate correctly. From Figs. 4.4,– 4.6, we see that VCP with S3.4 or S2.5 as the BM input format can achieve a performance close to Wu's decoder with S9.6 input in all cases except 64QAM with rate-3/4 coding. For 64QAM with rate-3/4 coding, S3.4 under VCP becomes worse. We need to use S2.5 or S1.6 to express BM input to achieve a performance closely to Wu with S9.6 or S7.8 input in this case. Therefore, we use the S2.5

Table 4.3: Comparison of Soft-Decision Decoding Performance, in AWGN at BER = 10^{-6}

Modulation	CC Code Rate	Theoretic Soft-Decision CC Coding Gain (dB)	CC Coding Gain from Simulation Employing Wu's Fixed-Point Computation (dB) (S9.6)	CC Coding Gain from Simulation Employing VCP (dB) (S2.5, CSB=51)
QPSK	1/2	6.99	5.62	5.12
QPSK	3/4	5.74	4.72	4.11
16QAM	1/2	6.99	6.62	6.73
16QAM	3/4	5.74	4.23	3.11 (S1.6)
64QAM	1/2	6.99	6.62	6.91
64QAM	2/3	6.02	5.91	5.02
64QAM	3/4	5.74	4.55	3.45 (CSB=58)

as the BM data format in the DSP implementation.

Table 4.3 compares the fixed-point coding gain obtained by Wu [3], the coding gain obtained by employing VCP, and the theoretic coding gain obtained previously. We see that the convolutional coding gain employing the VCP is lower than Wu's decoder by about 0.5 to 1.1 dB. It is less than theoretic value about 0.1 to 2.6 dB.

The CSB Effect

In Figs. 4.7,– 4.9, we can see the performance is almost close, when choosing the CSB as 45 to 68 in rate-1/2 QPSK and 16QAM. Besides, we can find that the performance of CSB = 45 ,51, 55, or 58 is better than CSB = 59, 62, 65, or 68 in rate-3/4 QPSK and 16QAM. But in 64QAM, we can see that performance of CSB = 45 is very worse, especially in rate-3/4. Furthermore, see that the performance of CSB = 58 is better than CSB = 51 or 58 by approximately 1 dB or more in rate-3/4 64QAM. In fact, the CSB can vary from 51 to 58 without affecting the BER performance much in all conditions. In the DSP implementation,

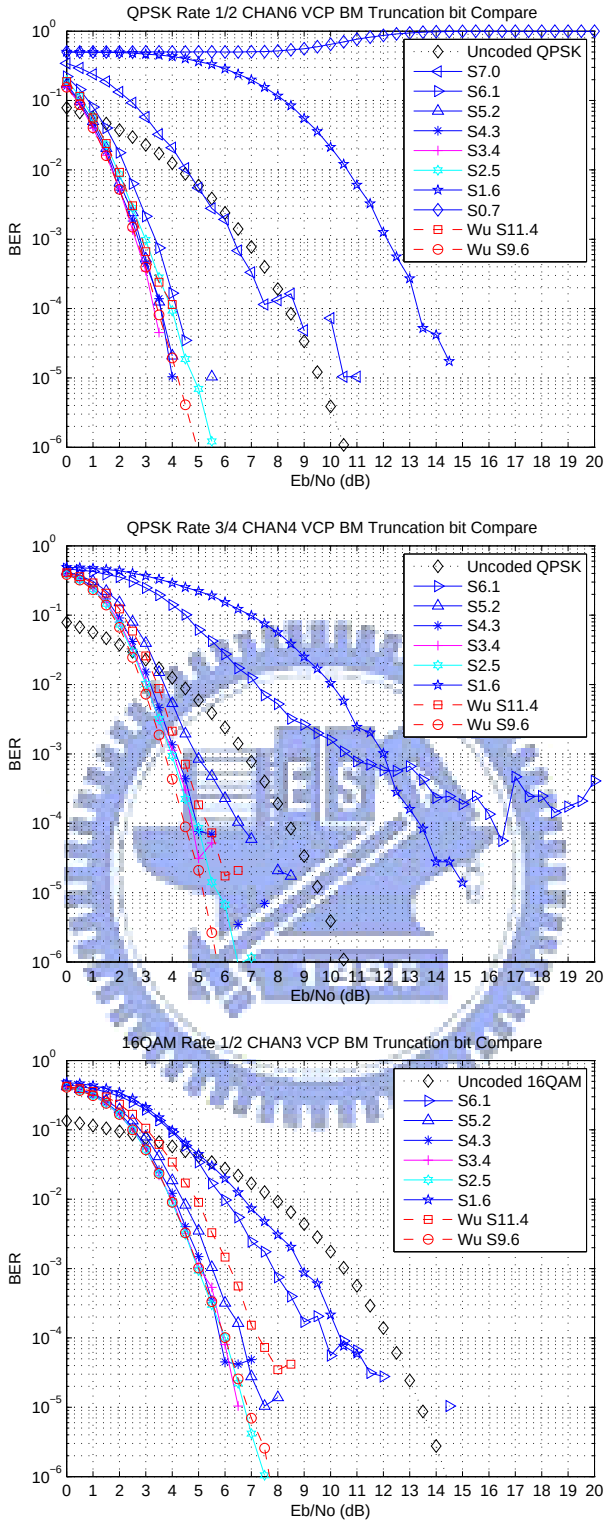


Figure 4.4: VCP decoding performance in AWGN with different BM truncation precisions (1/3).

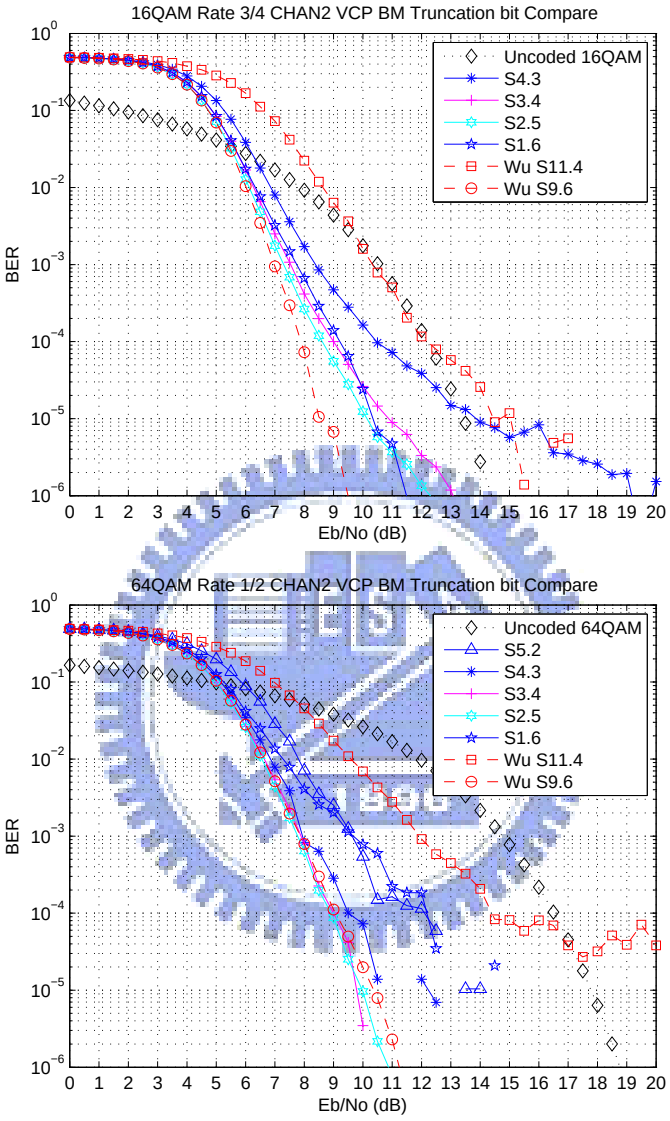


Figure 4.5: VCP decoding performance in AWGN with different BM truncation precisions (2/3).

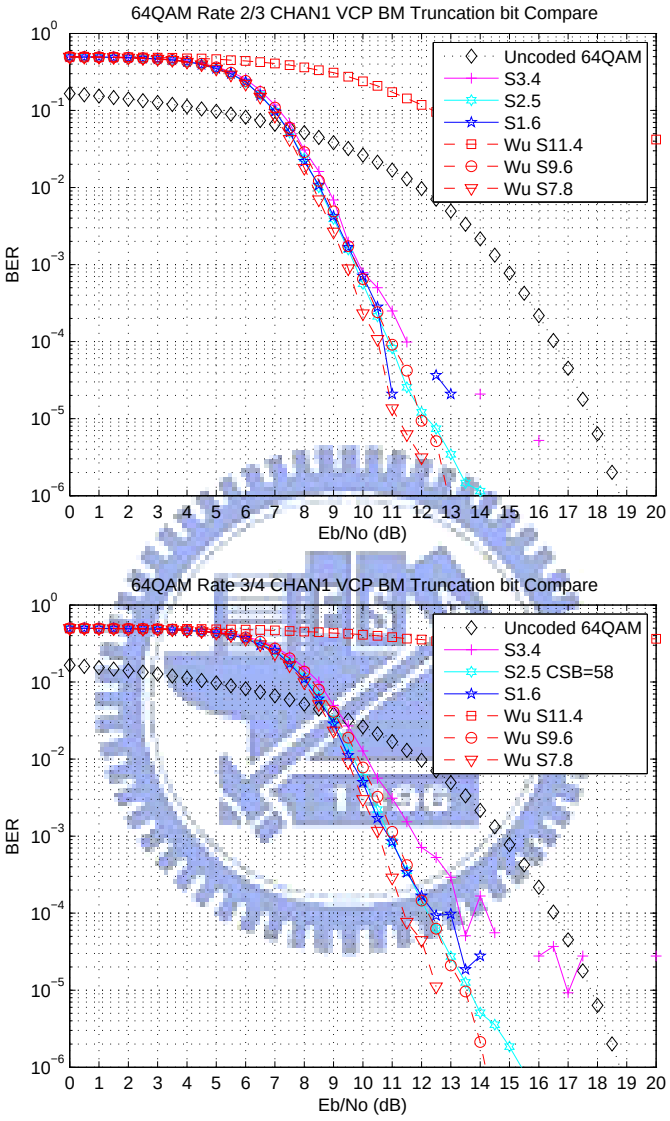


Figure 4.6: VCP decoding performance in AWGN with different BM truncation precisions (3/3).

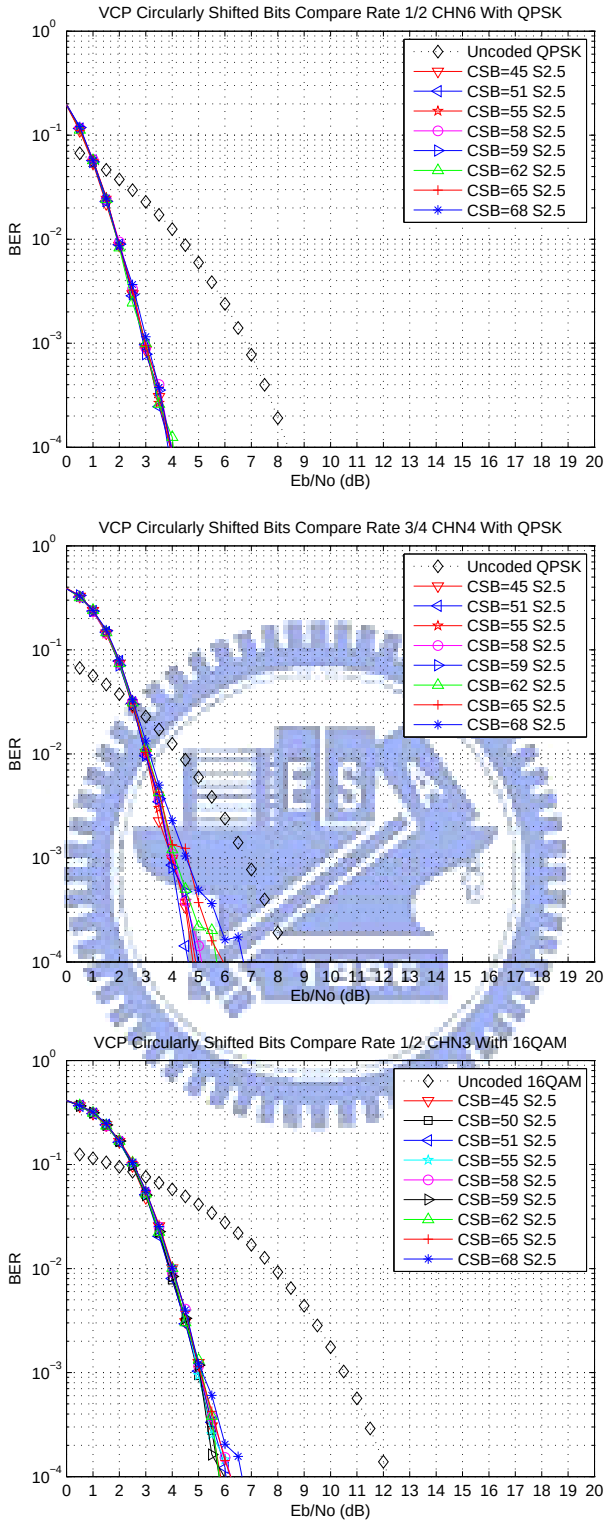


Figure 4.7: Effect of CSB values in VCP-based decoding in AWGN at different coding-modulation settings (1/3).

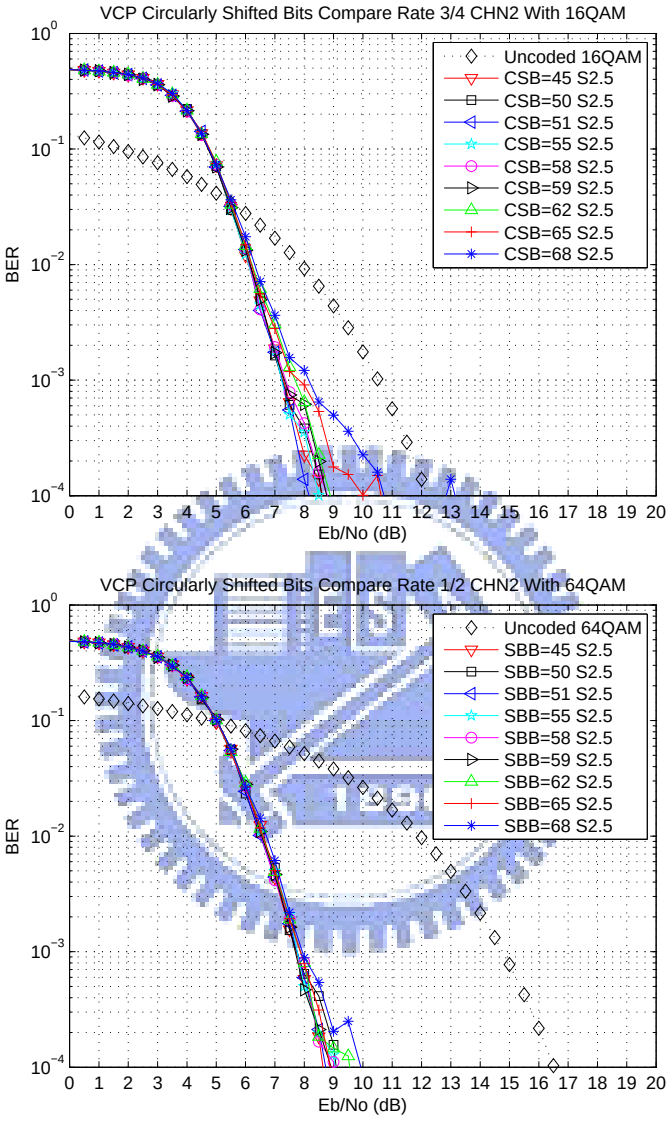


Figure 4.8: Effect of CSB values in VCP-based decoding in AWGN at different coding-modulation settings (2/3).

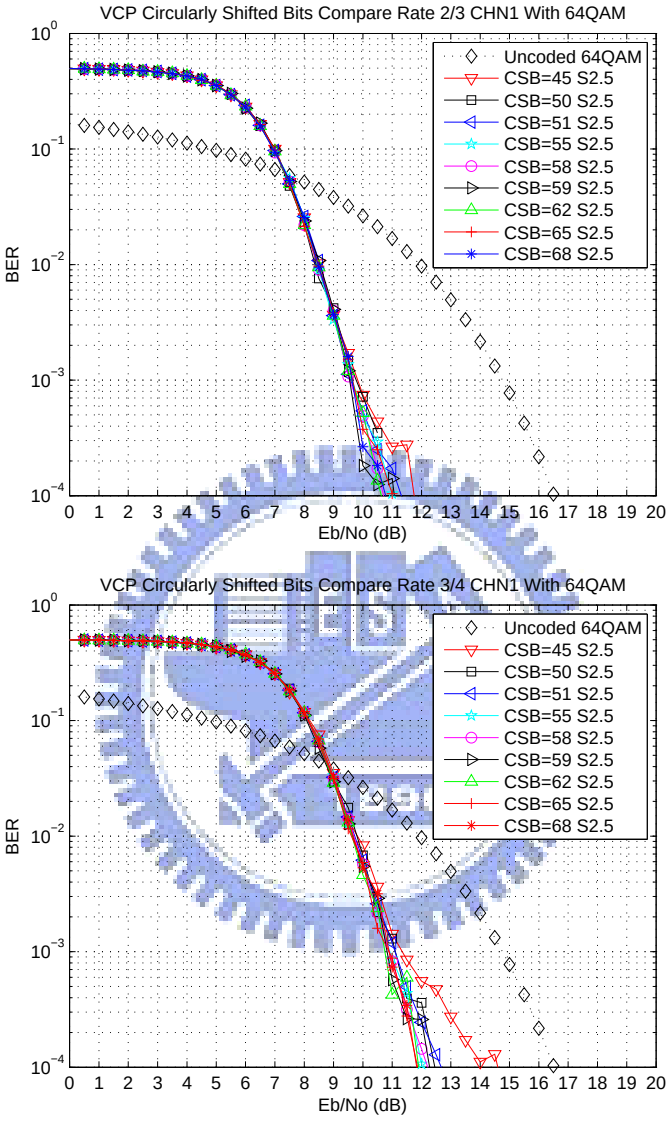


Figure 4.9: Effect of CSB values in VCP-based decoding in AWGN at different coding-modulation settings (3/3).

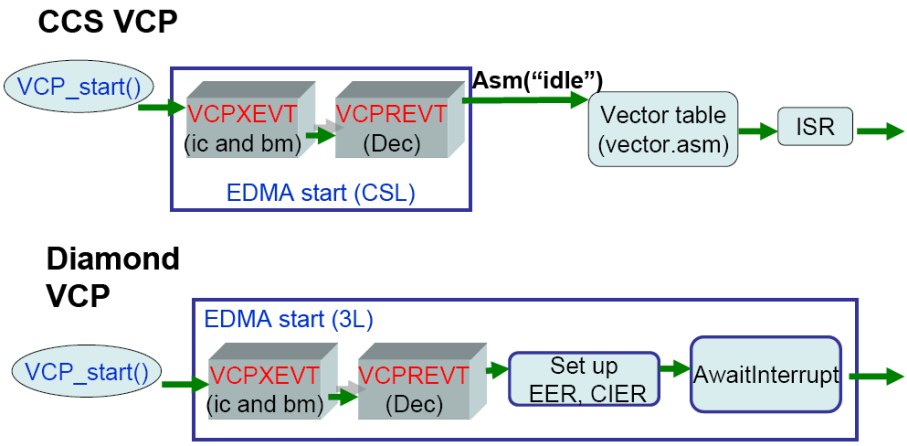


Figure 4.10: How VCP operates under 3L Diamond and TI CCS.

we chose the CSB = 58.

4.4 VCP Operation Under 3L Diamond

In this section, we consider use of VCP in a multiple DSPs environment which uses the 3L Diamond RTOS.

In chapter 3, we have described how EDMA function calls differ under 3L Diamond than TI CCS. The 3L RTOS helps the user in performing interrupts and, as a result, we do not have to use its vector table and the ISR (Interrupt Service Routines) in xxx.asm or xxx.c files. Fig. 4.10 illustrates how VCP operates under 3L and CCS.

For the 3L, we must prepare two EDMA channel parameter settings for VCPXEVT and VCPREVT. Then we set up EER (Event Enable Register) and CIER (Channel Interrupt Enable Register) and call the *AwaitInterrupt*() function. When *VCP_start*() is executed, it enables the ic_config and bm transfer parameters of VCPXEVT and the dec transfer parameter of VCPREVT. Then it wait for a interrupt of the finished work by *AwaitInterrupt*(). Finally, it finishes the operation of the VCP. In Fig. 4.10, we can also see that the in EDMA

Table 4.4: Speed of Overall Decoder from 3L-Measured Execution Time

Speed	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
Wu Executive Time (ms)	0.4631	0.4349	0.4606	0.4321	0.4613	0.3222	0.3454
Wu Information Data Rate (Kbps)	622	662	625	667	624	596	625
VCP Executive Time (ms)	0.1102	0.0839	0.1075	0.0816	0.1062	0.068	0.0678
VCP Channel Data Rate (Kbps)	**5226**	4577	5358	**4705**	5424	4236	4248
VCP Information Data Rate (Kbps)	**2616**	3433	2679	**3529**	2712	2824	3186
VCP Speed up (times over Wu)	**4.2**	5.2	4.3	**5.3**	4.3	4.7	5.1

start operation is different under 3L and CCS.

We now present the execution speed of the CC encoder and decoder under different conditions, including Wu's decoder without using VCP and VCP-based decodin. When operating under 3L, the speed data are obtained using the 3L's timer, and when operating under CCS, they are obtained using the profiling functionality of the CCS.

Table 4.4 shows the execution speed of the overall decoder consisting of demodulator, deinterleaver, tail-biting CC decoder, and derandomizer. We see that there is almost 4 to 5 times improvement in execution time by using the VCP. Moreover, if we do not consider the peripheral functions including demodulator, deinterleaver, and derandomizer then the VCP is 8 to 10 times higher in speed than the Viterbi decoder of Wu without using VCP.

We also utilize CCS's profiling functionality to estimate the executive cycles of different functions blocks. The results are shown in Table 4.5. Similarly, we utilize the 3L's timer to measure the executive times for different function blocks, with the result show in Table 4.6.

Table 4.7 shows the information data processing rate of different CC coding and modulation modes calculated from the CCS profile. The encoder, we can approach data rates

Table 4.5: CCS Profile of CC Coding and Decoding with VCP (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	4358	4358	4358	4358	4358	2918	3278
Encoder	1617	4070	1617	4070	1617	2914	3062
Interleaver	787	531	3493	2340	37574	18500	18500
Modulator	7451	4979	925	637	837	453	453
TX total	**14213**	**13938**	**10393**	**11405**	**44386**	**24767**	**25293**
De-modulator	676	460	745	4358	4358	422	422
De-interleaver	2327	1559	3484	2332	5228	2636	2636
VCP	**24860**	**24987**	**24860**	**24987**	**24860**	**24908**	**24891**
De-randomizer	4358	4358	4358	4358	4358	2918	3278
RX total	**32221**	**31364**	**33447**	**36035**	**38804**	**30884**	**31227**

between 7.8 and 20.7 Mbps whereas the decoder between 6.2 and 9.2 Mbps, with VCP. This may be computed with the decoding processing rates between 732 and 835 Kbps without using the VCP in [3].

Table 4.8 shows the corresponding estimates of processing rates calculated from the 3L-measured execution times. We can approach data rates between 3.4 and 4.1 Mbps for the encoder and between 2.6 and 3.5 Mbps for the decoder with the VCP. In comparison, Wu's decoding data rates without using VCP are between 596 and 667 Kbps in Table 4.4.

One peculiar point exists between Tables 4.8 and 4.7, that is, the encoder processing rates measured by CCS and 3L are highly incompatible. It is very strange that the encoder processing rate under 3L is just about 3 Mbps, which is on the same order of magnitude with the decoder processing rate. One explanation for this is that the 3L may have much overhead in some aspects, such as IO processing and others. Future work will hopefully clarify this situation better.

Table 4.6: 3L-Measured Execution Time of CC Coding and Decoding with VCP (ms)

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	0.0042	0.0055	0.0038	0.0048	0.0043	0.0028	0.0029
Encoder	0.0186	0.0260	0.0165	0.0280	0.0150	0.0120	0.0210
Interleaver	0.0600	0.0400	0.0590	0.0400	0.0590	0.0029	0.0290
Modulator	0.0086	0.0050	0.0026	0.0014	0.0027	0.0020	0.0025
TX total	**0.0914**	**0.0765**	**0.0819**	**0.0742**	**0.0810**	**0.0458**	**0.0554**
De-modulator	0.0045	0.0034	0.0032	0.0019	0.0022	0.0011	0.0012
De-interleaver	0.0690	0.0460	0.0680	0.0477	0.0714	0.0340	0.0341
VCP	**0.0314**	**0.0300**	**0.0315**	**0.0274**	**0.0325**	**0.0300**	**0.0290**
De-randomizer	0.0044	0.0055	0.0045	0.0057	0.0054	0.0030	0.0028
RX total	**0.1093**	**0.0849**	**0.1072**	**0.0827**	**0.1115**	**0.0681**	**0.0671**

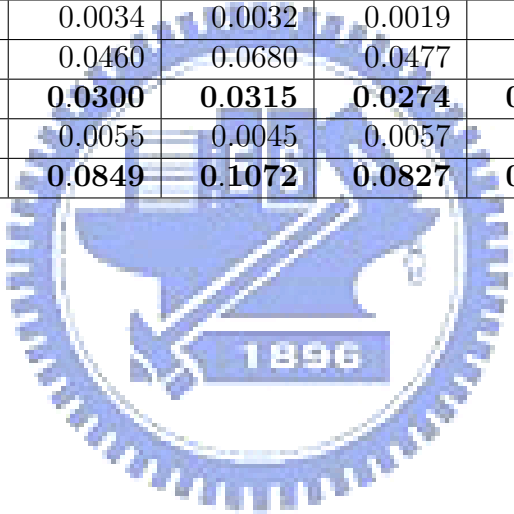


Table 4.7: Information Data Processing Rate Calculated from CCS Profile of CC with VCP

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	20,263	20,663	27,711	25,252	6,488	7,752	8,540
Decoder	8,938	9,183	8,611	7,992	7,422	6,217	6,917

Table 4.8: Information Data Processing Rate Calculated from 3L-Measure Execution Time of CC with VCP

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	3,412	3,840	3,547	4,000	3,429	4,042	4,075
Decoder	2,616	3,433	2,679	3,529	2,712	2,824	3,186

However, under CCS and 3L, as the measurements indicate the decoder processing rates are improved significantly by about 9.8 and 4.7 times, respectively, with use of VCP. Nevertheless there is still a very big gap between the processing rates in encoding and decoding under the CCS.

The programs will require multiple DSPs to run in parallel to handle the data rate under a 10 MHz transmission bandwidth. Acknowledgedly, further optimization of the programs may be possible. For our VCP implementation, we utilize to idle the DSP when the VCP is operated. For example, if we can parallelize the execution of the peripheral functions and the VCP, we may get approach an information data rate of about 6 Mbps in decoding under 3L.

Chapter 5

Simulation and DSP Implementation of CTC Encoder and Decoder

In this chapter, we present some simulation results for the CTC in IEEE 802.16e. We only implement rate-1/3 CTC encoder and decoder, which do not contain subpacket generation. This chapter considers floating-point and fixed-point simulations and DSP implementation.

5.1 Performance in AWGN Channel with Floating-Point Processing

The iteration number is a most important factor in the turbo decoding algorithm. This number affects the decoding accuracy and system complexity. A larger iteration number usually leads to better performance, but the complexity and latency are increased. The most frequently used numbers are 4 to 8 iterations.

Fig. 5.1 compares the performance for iteration numbers between 1 and 10 for max-log-MAP decoding at 480 information bits under three different modulations. We can see that if the iteration number is more than 2, the BER curves are very close. To limit the decoding complexity and maintain a reasonable performance, therefore, we choose 4 to be the iteration number in other simulations and DSP implementation.

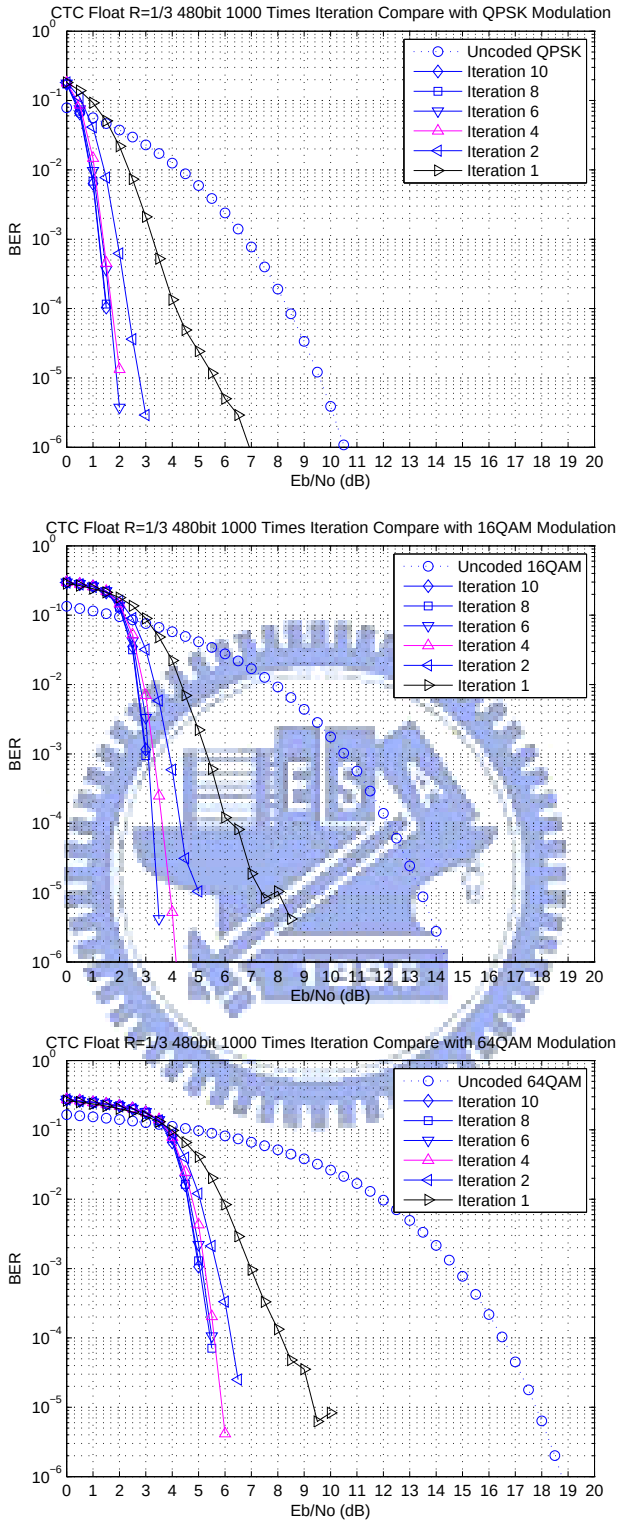


Figure 5.1: Performance of CTC at different iteration counts under different modulations.

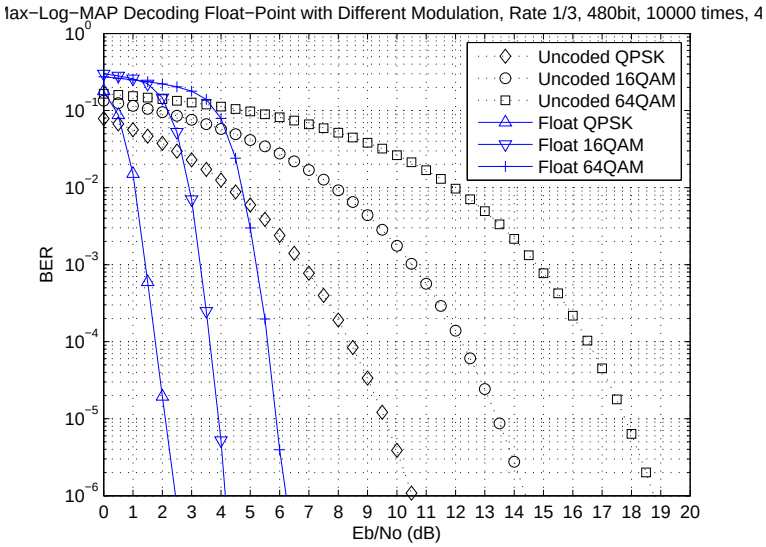


Figure 5.2: CTC decoding performance with different modulations employing floating-point computation at 4 iterations.

Fig. 5.2 compares the performance of the same three modulations at 4 decoding iterations, where the data are the same as those shown in Fig. 5.1. The coding gains of QPSK, 16QAM, and 64QAM at BER = 10^{-6} are 8.01, 10.15, and 12.55 dB, respectively.

In Table 5.1, we compare the coding gains of CTC and convolutional codes. Note that CC with tail-biting at rate 1/2 and CTC at rate 1/3 cannot be compared directly since they are different in code rate. So the comparison must be treated with caution. CTC is known to be better and close to the Shannon limit [10].

5.2 Performance in AWGN Channel with Fixed-Point Processing

We convert the floating-point value to the fixed-point value by multiplying the original floating-point value by 1000 and truncating the result to integer. Note that we only change the number of bits in the decoder input, *Extrinsics* loop and $GAMMA$ function. The aim

Table 5.1: Comparison of Coding Gains of CTC and Tail-biting CC in AWGN at BER = 10^{-6}

Modulation Type	Rate-1/2 Tail-biting CC With the VCP	Rate-1/3 CTC
QPSK	5.12	8.01
16QAM	6.73	10.15
64QAM	6.91	12.55

of truncation in the *Extrinsics* loop and $GAMMA$ function is to avoid the overflows at high SNR.

In Fig. 5.3 and 5.4, we list that the truncation parameters, which consist of "ChaReliab," "Scal," "Scal_E," and "Scal_g," standing for truncation of bits in the channel reliability, decoder input, *Extrinsics* loop and $GAMMA$ function, respectively. We also show how these parameters are used in the functions of our C program.

In Fig. 5.5, we compare the performance when the number of fractional bits in decoder is between 0 to 9 (S15.0 to S6.9) for max-log-MAP decoding at rate-1/3, 480 information bits and three different modulations. When we use S12.3 to S6.9., the BER curves are almost the same for QPSK, 16QAM and 64QAM. The BER curve for QPSK is in our acceptable limit when we use S12.3. But for 16QAM and 64QAM, S11.4 is the limit that we can accept. We can see that S10.5 to S6.9 cause have the overflows at high SNR. Hence, we employ "Scal_E" and "Scal_g" to avoid the overflows.

In Fig. 5.6, we show the performance with "Scal_E" and "Scal_g." We see that the overflow at high SNR disappears, but the performance is degraded at low SNR. Fortunately, no overflow occurs at high SNR for QPSK with S12.3, 16QAM with S11.4, and 64 QAM with S11.4. Consequently, we employ S12.3 and S11.4 for DSP implementation for the three

CTC Rate 1/3 CTC_FecDec.c Fixed Point Scale (S 11.4)
Last Update: 20080520
Author : Uefang-Smith

Prototype declarations	`#define ChaReliab 1`
	`/*float 10-Scal*/`
	`#define Scal 6`
	`#define Scal_E 1`
	`#define Scal_g 0`

Decoder input	`for (j=0; j<CTC_Input_Block; j++){`
	`*(sys_array+j) = (*(sys_array+j)*(L_c>>1))>>(Scal+ChaReliab);`
	`*(pinfo_array+j) = (*(pinfo_array+j)*(L_c>>1))>>(Scal+ChaReliab);`
	`*(parity1_array+j) = (*(parity1_array+j)*(L_c>>1))>>(Scal+ChaReliab);`
	`*(parity2_array+j) = (*(parity2_array+j)*(L_c>>1))>>(Scal+ChaReliab);}`

Channel Reliability	`a=1; //fading factor`
	`L_c=(a<<2)*CodeRate*pow(10,eb_ovr_n0/10)*(1<<ChaReliab);`

Extrinsics (MAP)	`Extrinsics[i]=`
	`(Log_likelihood_ratio[i]-(A[i]+(-1)*B[i])+(A[i]+B[i])-apriori[i])>>(Scal_E); //AB=01`
	`Extrinsics[i+N]=`
	`(Log_likelihood_ratio[i+N]-((-1)*A[i]+B[i])+(A[i]+B[i])-apriori[i+N])>>(Scal_E); //AB=10`
	`Extrinsics[i+(N<<1)]=(Log_likelihood_ratio[i+(N<<1)]-`
	`((-1)*A[i]+(-1)*B[i])+(A[i]+B[i])-apriori[i+(N<<1)])>>(Scal_E); //AB=11`

Branch Metric (Gamma)	`Case 0:gamma[i*4*State+j]=((A[i]*OUT[0]+B[i]*OUT[1]+Y[i]*OUT[2]+W[i]*OUT[3])+(p[0][i]))`
	`>>(Scal_g);break;//AB=00`
	`Case 1:gamma[i*4*State+j]=((A[i]*OUT[0]+B[i]*OUT[1]+Y[i]*OUT[2]+W[i]*OUT[3])+(p[1][i]))`
	`>>(Scal_g);break;//AB=01`
	`Case 2:gamma[i*4*State+j]=((A[i]*OUT[0]+B[i]*OUT[1]+Y[i]*OUT[2]+W[i]*OUT[3])+(p[2][i]))`
	`>>(Scal_g);break;//AB=10`
	`Case 3:gamma[i*4*State+j]=((A[i]*OUT[0]+B[i]*OUT[1]+Y[i]*OUT[2]+W[i]*OUT[3])+(p[3][i]))`
	`>>(Scal_g);break;//AB=11`

Figure 5.3: CTC fixed-point truncation parameters.

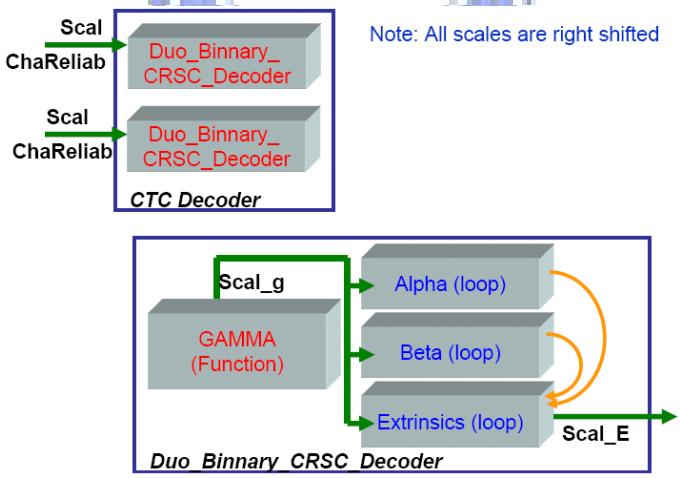
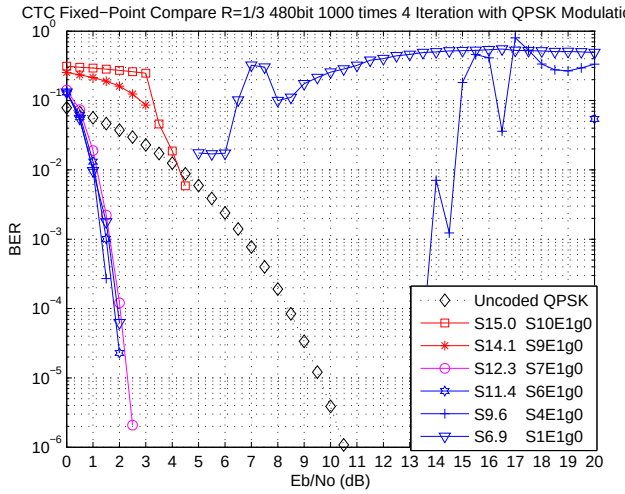
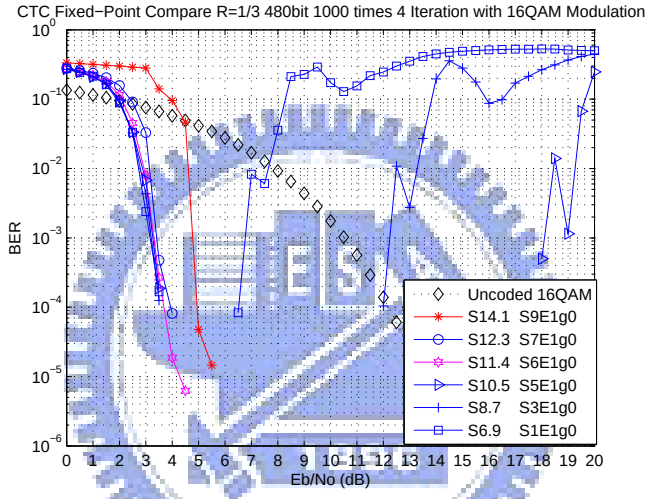


Figure 5.4: CTC fixed-point truncation parameters flow chart.





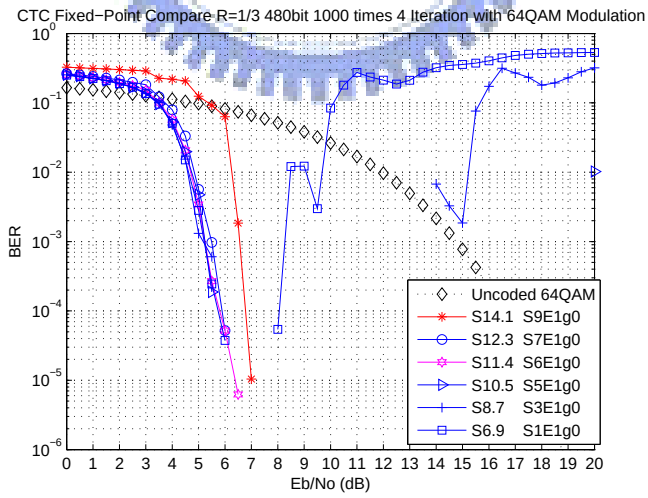


Figure 5.5: CTC at different bit numbers with different modulations.

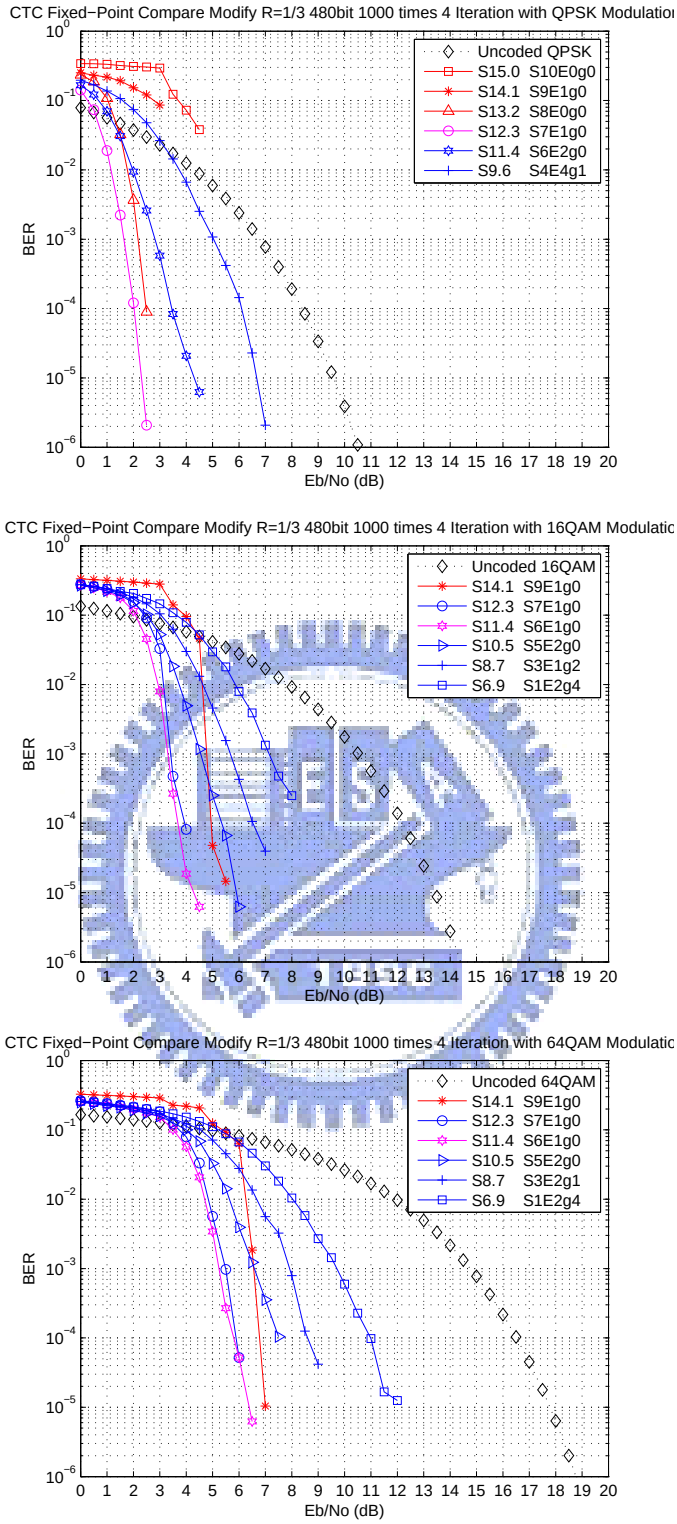


Figure 5.6: Performance with scaling of various quantities in CTC decoding to avoid overflow at high SNR.

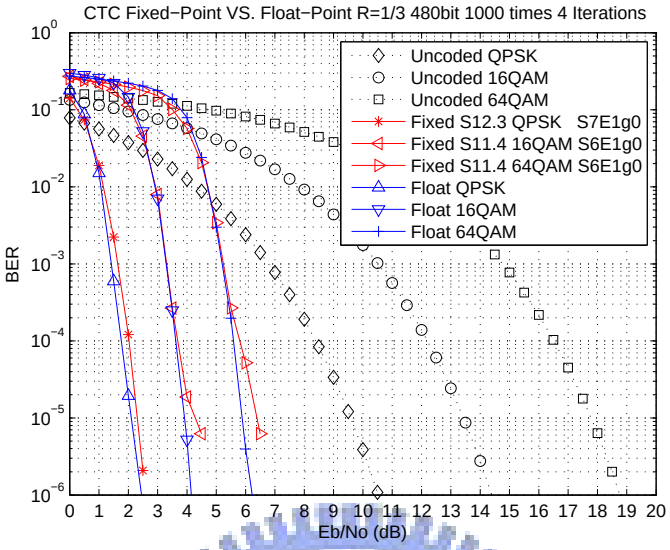


Figure 5.7: BER performance of CTC decoding with fixed-point computation vs. floating-point computation.

Table 5.2: Coding Gain Performance of Rate-1/3 CTC in AWGN at BER = 10^{-5} with Floating-Point and Fixed-Point Computation

Modulation	Floating-Point Coding Gain (dB)	Fixed-Point Coding Gain (dB)
QPSK	7.54	7.34
16QAM	9.41	9.08
64QAM	11.92	11.42

different modulations.

Table 5.2 shows the coding gains obtained with floating-point computation and that with fixed-point computation. And Fig. 5.7 depicts the BER results.

5.2.1 Speed Performance of the DSP Code

In this section, we show the CCS profile of our DSP code that includes the 1/3 CTC encoder, modulator, demodulator, and decoder. We also measure the speed in 3L.

Compiler Optimization Options

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 the 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.

Rate-1/3 CTC Encoder

First of all, we optimize our code and obtain the profile using CCS. We also utilize 3L's timer to measure the executive times and data rates of CTC encoder with three different

Table 5.3: CTC Rate-1/3 Encoder Execution Times Measured under 3L and the Corresponding Data Rate with 480-Bit Information Data Blocks

Modulation	Execution Time (ms)	Information Data Rate (Kbps)	Channel Data Rate (Kbps)
QPSK	0.062	7,742	23,226
16QAM	0.053	9,057	27,170
64QAM	0.061	7,869	23,607

Table 5.4: Profile of *CTC_Encoder* with QPSK Modulation for One Data Block

Function	Times Called	CPU Cycles	Percentage (%)
DBCRSCC_Encoder	2	9,906	38
CTC_Interleaver	1	15,969	62

modulations, whose results are shown in Table 5.3. The information data rates are about 7.7 to 9.1 Mbps.

In Table 5.4, we see that 38% and 62% of the execution time are spent in *CTC_Interleaver* and *DBCRSCC_Encoder*, respectively. The *DBCRSCC_Encoder* is called twice for producing the parity bits, i.e., $Y1$, $W1$, $Y2$, and $W2$. But the two permutations performed by *CTC_Interleaver* requires more computational cycles.

Rate-1/3 CTC Decoder

Table 5.5 lists execution times measured over ten iterations for the three different modulations. We find that the three modulations are not significantly different in execution times. In addition, the executive time averages to about 4 ms per iteration.

Table 5.6 shows the equivalent processing rates of our CTC decoder on DSP for two iter-

Table 5.5: CTC Rate-1/3 Decoder Executive Times for 480 Information Bits Measured under 3L

	Interation Number									
Modulation	10	9	8	7	6	5	4	3	2	1
QPSK (ms)	39.02	35.13	31.23	27.37	23.48	19.56	15.68	11.80	7.92	4.03
16QAM (ms)	39.01	35.16	31.23	27.35	23.47	19.59	15.67	11.83	7.92	4.03
64QAM (ms)	39.00	35.11	31.24	27.34	23.46	19.56	15.68	11.84	7.93	4.04

ations and four iterations. Note that although the processing data rates with two iterations are two times that with four iterations, its performance is degraded by about 1 dB.

Table 5.7 dissects a *CTC_Decoder* into constituent functions and the corresponding complexity. The results show that about 90% execution time is spent on the *Duo_Binnary_CRSC_Decoder* function, which is used for the double binary max-log-MAP decoding algorithm. Two points are worth making about Table 5.7. The first is that the *Duo_Binnary_CRSC_Decoder* function is called 2 times for decoding two constituent codings in one iteration. The second point is that the $max4$ function is called 2537 times for estimating the decoding output bits and it takes 9% of the CPU cycles.

A more detailed understanding of the *Duo_Binnary_CRSC_Decoder* function relationship can be gained from Table 5.8, where we show that it consists of $GAMMA$ function, *Alpha* loop, *Beta* loop, *Extrinsics* loop, and other parts. As the $max8$ and $max4$ functions are called by these functions and loops many times in the last two rows.

In *Duo_Binnary_CRSC_Decoder* function, the $max4$ function is called 8 times to do the comparisons each of the *Alpha* (forward) and *Beta* (backward) computation in each symbol. Since the number of symbols N = 240, the $max4$ function is called $(8 + 8) \times 240 = 3840$ times.

In the same way, as the $max8$ function is called 4, 1, and 1 times in *Extrinsic*, *Alpha*, and

Table 5.6: Corresponding Processing Rates of CTC Rate-1/3 Decoder Based on 3L-Measured Execution Times for One Information Data Block of 480 Bits

Number of Iterations	Modulation	Execution Time (ms)	Channel Data Rate (Kbps)	Information Data Rate (Kbps)
2	QPSK	7.92	181.81	60.61
	16QAM	7.92	181.81	60.61
	64QAM	7.93	181.59	60.53
4	QPSK	15.68	91.84	30.61
	16QAM	15.67	91.90	30.63
	64QAM	15.68	91.84	30.61

Table 5.7: Profile of *CTC_Decoder* with QPSK Modulation for One Data Block in One Iteration

Function	Times Called	CPU Cycles	Percentage (%)
De_multiplex	1	9571	0.18
Duo_Binnary_CRSC_Decoder	2	4749690	89.99
Permutation	1	10028	0.19
MAP_Interleaver_Decoder	1	10168	0.19
MAP_Interleaver_Inverse	2	13136	0.25
max4	2537	487104	9.2

Table 5.8: Profile of *Duo_Binnary_CRSC_Decoder*

Function/Loop	Times Called	CPU Cycles	Percentage (%)	Cycles/Access
GAMMA (F)	1	645148	26.9	645148
Alpha (L)	240	619442	25.9	2581
Beta (L)	240	647011	27.1	2696
Extrinsics (L)	240	478205	20.1	1993
max4 (F)	3840	737280	*N/A*	192
max8 (F)	1440	624960	*N/A*	434

Table 5.9: Rate-1/3 CTC Processing Rate with 4 iterations in Decoding

Processing Rate (Kbps)	Modulation type		
	QPSK	16QAM	64QAM
Encoder with Modulation	7742	9057	7869
Decoder with Demodulation	30.61	30.63	30.61

Beta computations, respectively, in each symbol. Hence it is called $(4+1+1) \times 240 = 1440$ times.

Table 5.9 shows the processing rate in different modulation modes for encoder and decoder. As the decoder requires a large amount of computation in operations like *Alpha*, *Beta*, and $GAMMA$ computations, we can only achieve about 30 Kbps of decoding speed.

5.2.2 Improving CTC Decoding Speed

As the CTC decoding data rate is low, we consider some methods to improve the decoding speed as following:

- Use the software pipeline information from the DSP's C compiler output to see the degree of parallelism of our assembly code.
- Use the DSP intrinsic function $\_max2()$, which is a special function that maps directly to inline C64x instructions, to replace our $max4$ and $max8$ functions.
- Use the DSP intrinsic $\_sadd2()$ to perform parallel addition operations.
- Do loop unrolling to improve the parallelizability of our C program.
- Avoid using the malloc() function to allocate memory arrays.

Table 5.10: Profile of Improved *Duo_Binnary_CRSC_Decoder*

Function/Loop	Original (Cycles)	Improved (Cycles)	Reduction in Complexity (%)
GAMMA (F)	645148	17830	97.23
Alpha (L)	619442	9084	98.53
Beta (L)	647011	8128	98.74
Extrinsics (L)	478205	13440	97.19

- Avoid using the switch-case programming. We can modify it to a table lookup. This is because if a loop contains conditional break, it is not software pipelined.

- Utilizing shift operations as much possible as to replace multiplications and divisions.

We take the *GAMMA* function as an example to show its assembly code and software pipeline information. For space reason, we omit the assembly code of the other functions. For additional information on software pipelining, we refer to [27]. The improved code of *GAMMA* function is shown in Figs. 5.8, 5.9, and 5.10. Note that we use the same loop unrolling technique in *GAMMA* array, yielding a total of 16 loops for 32 branch states, as shown in Figs. 5.9 and Fig. 5.10.

In Figs. 5.11,– 5.16, we show the assembly code that computes the branch metrics from received systematic and parity bits. We can see from the software pipeline information in Fig. 5.14 and 5.16 that it achieved a certain degree of parallelism. Note that we have only shown a small part of the assembly code for the *GAMMA* function.

Table 5.10 shows the improvement in speed of *GAMMA* function, *Alpha* loop, *Beta* loop, and *Extrinsics* loop. They account for 36.8, 18.7, 16.8 and 27.7%, respectively, of the complexity of the improved *Duo_Binnary_CRSC_Decoder*.

```
void GAMMA(short *gamma,short *aprior,short *A,short *B,short *Y,short *W,int length)
{
#ifdef _no_cycle
int j;
int OUT[4];
#endif
int i,k;
#ifdef _Malloc_Memory_Bug
short **p,*pData;  //p[4][block_length]
#endif

#ifdef _Normal_Memory
short p[4][960];
#endif
#ifdef _cycle
/*Output lookup table for TX Trellis (0-->1 1--->-1):BPSK hardware*/
int OutTable[32][4]={
                                  //ABYW
        /*state 0*/  { 1, 1, 1, 1},//0000
                     { 1,-1,-1,-1},//0111
                     {-1, 1,-1,-1},//1011
                     {-1,-1, 1, 1},//1100
        /*state 1*/  { 1, 1, 1, 1},//0000
                     { 1,-1,-1,-1},//0111
                     {-1, 1,-1,-1},//1011
                     {-1,-1, 1, 1},//1100
        /*state 2*/  { 1, 1,-1, 1},//0010
                     { 1,-1, 1,-1},//0101
                     {-1, 1, 1,-1},//1001
                     {-1,-1,-1, 1},//1110
        /*state 3*/  { 1, 1,-1, 1},//0010
                     { 1,-1, 1,-1},//0101
                     {-1, 1, 1,-1},//1001
                     {-1,-1,-1, 1},//1110
        /*state 4*/  { 1, 1,-1,-1},//0011
                     { 1,-1, 1, 1},//0100
                     {-1, 1, 1, 1},//1000
                     {-1,-1,-1,-1},//1111
        /*state 5*/  { 1, 1,-1,-1},//0011
                     { 1,-1, 1, 1},//0100
                     {-1, 1, 1, 1},//1000
                     {-1,-1,-1,-1},//1111
        /*state 6*/  { 1, 1, 1,-1},//0001
                     { 1,-1,-1, 1},//0110
                     {-1, 1,-1, 1},//1010
                     {-1,-1, 1,-1},//1101
        /*state 7*/  { 1, 1, 1,-1},//0001
                     { 1,-1,-1, 1},//0110
                     {-1, 1,-1, 1},//1010
                     {-1,-1, 1,-1} //1101
                     };
#endif
#ifdef _Malloc_Memory_Bug
/*Avoid memory fragment for malloc */
p=(short **)malloc(4*sizeof(short *)+4*length*sizeof(short));
for(i=0,pData=(short *)(p+4); i<4; i++, pData+=length)
  p[i]=pData;
#endif
```

Figure 5.8: Function *Gamma*() (1/3).

```
#ifdef _cycle
for(i=0;i<length;i++)
  {
    /*apprior  00 01 10 11*/ //gamma 1068407  1068394
    p[0][i]=-_max2(_max2(0,aprior[1+(i<<2)]),_max2(aprior[2+(i<<2)],aprior[3+(i<<2)]));
    p[1][i]=aprior[1+(i<<2)]-_max2(_max2(0,aprior[1+(i<<2)]),_max2(aprior[2+(i<<2)],aprior[3+(i<<2)]));
    p[2][i]=aprior[2+(i<<2)]-_max2(_max2(0,aprior[1+(i<<2)]),_max2(aprior[2+(i<<2)],aprior[3+(i<<2)]));
    p[3][i]=aprior[3+(i<<2)]-_max2(_max2(0,aprior[1+(i<<2)]),_max2(aprior[2+(i<<2)],aprior[3+(i<<2)]));
  }

/*Loop Unrolling*/
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+0]=((A[i]*OutTable[0][0]+B[i]*OutTable[0][1]+Y[i]*OutTable[0][2]+W[i]*OutTable[0][3])+(p[0][i]))>>(Scal_g);//AB=00
gamma[k+1]=((A[i]*OutTable[1][0]+B[i]*OutTable[1][1]+Y[i]*OutTable[1][2]+W[i]*OutTable[1][3])+(p[1][i]))>>(Scal_g);//AB=01
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+2]=((A[i]*OutTable[2][0]+B[i]*OutTable[2][1]+Y[i]*OutTable[2][2]+W[i]*OutTable[2][3])+(p[2][i]))>>(Scal_g);//AB=10
gamma[k+3]=((A[i]*OutTable[3][0]+B[i]*OutTable[3][1]+Y[i]*OutTable[3][2]+W[i]*OutTable[3][3])+(p[3][i]))>>(Scal_g);//AB=11
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+4]=((A[i]*OutTable[4][0]+B[i]*OutTable[4][1]+Y[i]*OutTable[4][2]+W[i]*OutTable[4][3])+(p[0][i]))>>(Scal_g);//AB=00
gamma[k+5]=((A[i]*OutTable[5][0]+B[i]*OutTable[5][1]+Y[i]*OutTable[5][2]+W[i]*OutTable[5][3])+(p[1][i]))>>(Scal_g);//AB=01
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+6]=((A[i]*OutTable[6][0]+B[i]*OutTable[6][1]+Y[i]*OutTable[6][2]+W[i]*OutTable[6][3])+(p[2][i]))>>(Scal_g);//AB=10
gamma[k+7]=((A[i]*OutTable[7][0]+B[i]*OutTable[7][1]+Y[i]*OutTable[7][2]+W[i]*OutTable[7][3])+(p[3][i]))>>(Scal_g);//AB=11
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+8]=((A[i]*OutTable[8][0]+B[i]*OutTable[8][1]+Y[i]*OutTable[8][2]+W[i]*OutTable[8][3])+(p[0][i]))>>(Scal_g);//AB=00
gamma[k+9]=((A[i]*OutTable[9][0]+B[i]*OutTable[9][1]+Y[i]*OutTable[9][2]+W[i]*OutTable[9][3])+(p[1][i]))>>(Scal_g);//AB=01
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+10]=((A[i]*OutTable[10][0]+B[i]*OutTable[10][1]+Y[i]*OutTable[10][2]+W[i]*OutTable[10][3])+(p[2][i]))>>(Scal_g);//AB=10
gamma[k+11]=((A[i]*OutTable[11][0]+B[i]*OutTable[11][1]+Y[i]*OutTable[11][2]+W[i]*OutTable[11][3])+(p[3][i]))>>(Scal_g);//AB=11
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+12]=((A[i]*OutTable[12][0]+B[i]*OutTable[12][1]+Y[i]*OutTable[12][2]+W[i]*OutTable[12][3])+(p[0][i]))>>(Scal_g);//AB=00
gamma[k+13]=((A[i]*OutTable[13][0]+B[i]*OutTable[13][1]+Y[i]*OutTable[13][2]+W[i]*OutTable[13][3])+(p[1][i]))>>(Scal_g);//AB=01
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+14]=((A[i]*OutTable[14][0]+B[i]*OutTable[14][1]+Y[i]*OutTable[14][2]+W[i]*OutTable[14][3])+(p[2][i]))>>(Scal_g);//AB=10
gamma[k+15]=((A[i]*OutTable[15][0]+B[i]*OutTable[15][1]+Y[i]*OutTable[15][2]+W[i]*OutTable[15][3])+(p[3][i]))>>(Scal_g);//AB=11
}
```

Figure 5.9: Function *Gamma*() (2/3).

```
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+16]=((A[i]*OutTable[16][0]+B[i]*OutTable[16][1]+Y[i]*OutTable[16][2]+W[i]*OutTable[16][3])+(p[0][i]))>>(Scal_g);//AB=00
gamma[k+17]=((A[i]*OutTable[17][0]+B[i]*OutTable[17][1]+Y[i]*OutTable[17][2]+W[i]*OutTable[17][3])+(p[1][i]))>>(Scal_g);//AB=01
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+18]=((A[i]*OutTable[18][0]+B[i]*OutTable[18][1]+Y[i]*OutTable[18][2]+W[i]*OutTable[18][3])+(p[2][i]))>>(Scal_g);//AB=10
gamma[k+19]=((A[i]*OutTable[19][0]+B[i]*OutTable[19][1]+Y[i]*OutTable[19][2]+W[i]*OutTable[19][3])+(p[3][i]))>>(Scal_g);//AB=11
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+20]=((A[i]*OutTable[20][0]+B[i]*OutTable[20][1]+Y[i]*OutTable[20][2]+W[i]*OutTable[20][3])+(p[0][i]))>>(Scal_g);//AB=00
gamma[k+21]=((A[i]*OutTable[21][0]+B[i]*OutTable[21][1]+Y[i]*OutTable[21][2]+W[i]*OutTable[21][3])+(p[1][i]))>>(Scal_g);//AB=01
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+22]=((A[i]*OutTable[22][0]+B[i]*OutTable[22][1]+Y[i]*OutTable[22][2]+W[i]*OutTable[22][3])+(p[2][i]))>>(Scal_g);//AB=10
gamma[k+23]=((A[i]*OutTable[23][0]+B[i]*OutTable[23][1]+Y[i]*OutTable[23][2]+W[i]*OutTable[23][3])+(p[3][i]))>>(Scal_g);//AB=11
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+24]=((A[i]*OutTable[24][0]+B[i]*OutTable[24][1]+Y[i]*OutTable[24][2]+W[i]*OutTable[24][3])+(p[0][i]))>>(Scal_g);//AB=00
gamma[k+25]=((A[i]*OutTable[25][0]+B[i]*OutTable[25][1]+Y[i]*OutTable[25][2]+W[i]*OutTable[25][3])+(p[1][i]))>>(Scal_g);//AB=01
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+26]=((A[i]*OutTable[26][0]+B[i]*OutTable[26][1]+Y[i]*OutTable[26][2]+W[i]*OutTable[26][3])+(p[2][i]))>>(Scal_g);//AB=10
gamma[k+27]=((A[i]*OutTable[27][0]+B[i]*OutTable[27][1]+Y[i]*OutTable[27][2]+W[i]*OutTable[27][3])+(p[3][i]))>>(Scal_g);//AB=11
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+28]=((A[i]*OutTable[28][0]+B[i]*OutTable[28][1]+Y[i]*OutTable[28][2]+W[i]*OutTable[28][3])+(p[0][i]))>>(Scal_g);//AB=00
gamma[k+29]=((A[i]*OutTable[29][0]+B[i]*OutTable[29][1]+Y[i]*OutTable[29][2]+W[i]*OutTable[29][3])+(p[1][i]))>>(Scal_g);//AB=01
}
for(i=0;i<length;i++)
{
k=i<<5;
gamma[k+30]=((A[i]*OutTable[30][0]+B[i]*OutTable[30][1]+Y[i]*OutTable[30][2]+W[i]*OutTable[30][3])+(p[2][i]))>>(Scal_g);//AB=10
gamma[k+31]=((A[i]*OutTable[31][0]+B[i]*OutTable[31][1]+Y[i]*OutTable[31][2]+W[i]*OutTable[31][3])+(p[3][i]))>>(Scal_g);//AB=11
}
#endif
```

Figure 5.10: Function *Gamma*() (3/3).

Table 5.11: Speed up in Decoding of One Data Block with QPSK Modulation for One Iteration

Function	Cycles	Reduction in Complexity (%)
Duo_Binnary_CRSC_Decoder (Original)	4749690	*N/A*
Duo_Binnary_CRSC_Decoder (Improved)	109816	97.68
CTC_Decoder (Original)	4932457	*N/A*
CTC_Decoder (Improved)	147116	97.01
DeModulation	1468	*N/A*

```
void GAMMA(short *gamma,short *aprior,short *A,short *B,short *Y,short *W,int length)
{
00001578          GAMMA:
00001578 07EFF053            ADDK.S2       -8224,SP
0000157C 0FBC1FD8  ||        OR.L1X        0,SP,A31
00001580 058808FE            STW.D2T2      B11,*+SP[0x808]
00001584 050807FE            STW.D2T2      B10,*+SP[0x807]
00001588 018806FE            STW.D2T2      B3,*+SP[0x806]
0000158C 067C4144            STDW.D1T1     A13:A12,*-A31[0x2]
00001590 057C6144            STDW.D1T1     A11:A10,*-A31[0x3]
00001594 06181FD8            OR.L1X        0,B6,A12
00001598 05101FDB            OR.L2X        0,A4,B10
0000159C 05201FD9  ||        OR.L1X        0,B8,A10
000015A0 05A006A1  ||        OR.S1         0,A8,A11
000015A4 069808F1  ||        OR.D1         0,A6,A13
000015A8 059006A2  ||        OR.S2         0,B4,B11
#ifdef _no_cycle
int j;
int OUT[4];
#endif
int i,k;
#ifdef _Malloc_Memory_Bug
short **p,*pData;  //p[4][block_length]
#endif

#ifdef _Normal_Memory
short p[4][960];
#endif
#ifdef _cycle
/*Output lookup table for TX Trellis (0-->1 1--->-1);BPSK hardware*/
int OutTable[32][4]={
000015AC 001BF810            B.S1          memcpy
000015B0 018D8428            MVK.S1        0x1b08,A3
000015B4 018001E8            MVKH.S1       0x30000,A3
000015B8 020F0428            MVK.S1        0x1e08,A4
000015BC 018B0162            ADDKPC.S2     RL26,B3,0
000015C0 020C1FDB            OR.L2X        0,A3,B4
000015C4 023C9079  ||        ADD.L1X       A4,SP,A4
000015C8 03010028  ||        MVK.S1        0x0200,A6
000015CC          RL26:
000015CC 0BFE1FAB            MVK.S2        0xfffffc3f,B23
000015D0 00000001  ||        NOP
000015D4 00000001  ||        NOP
000015D8 00000000  ||        NOP
000015DC 0207852B            MVK.S2        0x0f0a,B4
000015E0 0B7E2028  ||        MVK.S1        0xfffffc40,A22
000015E4 0B01DFAB            MVK.S2        0x03bf,B22
000015E8 023C807B  ||        ADD.L2        B4,SP,B4
000015EC 02AC4943  ||        ADD.D2        B11,0x2,B5
```

Figure 5.11: The assembly code of *Gamma*() function (1/6).

```
000015F0 0A81E028  ||        MVK.S1        0x03c0,A21
                                        //ABY%
        /*state 0*/  { 1, 1, 1, 1},//0000
                     { 1,-1,-1,-1},//0111
                     {-1, 1,-1,-1},//1011
                     {-1,-1, 1, 1},//1100
        /*state 1*/  { 1, 1, 1, 1},//0000
                     { 1,-1,-1,-1},//0111
                     {-1, 1,-1,-1},//1011
                     {-1,-1, 1, 1},//1100
        /*state 2*/  { 1, 1,-1, 1},//0010
                     { 1,-1, 1,-1},//0101
                     {-1, 1, 1,-1},//1001
                     {-1,-1,-1, 1},//1110
        /*state 3*/  { 1, 1,-1, 1},//0010
                     { 1,-1, 1,-1},//0101
                     {-1, 1, 1,-1},//1001
                     {-1,-1,-1, 1},//1110
        /*state 4*/  { 1, 1,-1,-1},//0011
                     { 1,-1, 1, 1},//0100
                     {-1, 1, 1, 1},//1000
                     {-1,-1,-1,-1},//1111
        /*state 5*/  { 1, 1,-1,-1},//0011
                     { 1,-1, 1, 1},//0100
                     {-1, 1, 1, 1},//1000
                     {-1,-1,-1,-1},//1111
        /*state 6*/  { 1, 1, 1,-1},//0001
                     { 1,-1,-1, 1},//0110
                     {-1, 1,-1, 1},//1010
                     {-1,-1, 1,-1},//1101
        /*state 7*/  { 1, 1, 1,-1},//0001
                     { 1,-1,-1, 1},//0110
                     {-1, 1,-1, 1},//1010
                     {-1,-1, 1,-1} //1101
                     };
#endif
#ifdef _Malloc_Memory_Bug
/*Avoid memory fragment for malloc */
p=(short **)malloc(4*sizeof(short *)+4*length*sizeof(short));
for(i=0,pData=(short *)(p+4); i<4; i++, pData+=length)
  p[i]=pData;
#endif

#ifdef _cycle
for(i=0;i<length;i++)
000015F4 0C8403E2            MVC.S2        CSR,B25
000015F8 00003BA9            MVK.S1        0x0077,A0
000015FC 09939059  ||        SUB.L1X       B4,4,A19
```

Figure 5.12: The assembly code of *Gamma*() function (2/6).

```
00001600 0367CF5B  ||         AND.L2        -2,B25,B6
00001604 00800040  ||         MVK.D1        0,A1
00001608 0213805B             SUB.L2        B4,4,B4
0000160C 009803A3  ||         MVC.S2        B6,CSR
00001610 0A141FD9  ||         OR.L1X        0,B5,A20
00001614 008000E9  ||         MVKH.S1       0x10000,A1
00001618 0ABC8943  ||         ADD.D2        SP,0x4,B21
0000161C 09000040  ||         MVK.D1        0,A18
00001620           L69:
00001620 9314C0C7      [!A1]  LDH.D2T2      *-B5[0x6],B6
00001624 02DE4859  ||         MAX2.L1       A18,A23,A5
00001628 94D04045  || [!A1]   LDH.D1T1      *-A20[0x2],A9
0000162C 031410DB  ||         NEG.L2X       A5,B6
00001630 00000001  ||         NOP
00001634 00000001  ||         NOP
00001638 00000001  ||         NOP
0000163C 00000000  ||         NOP
00001640 981500C7      [!A1]  LDH.D2T2      *-B5[0x8],B16
00001644 92504045  || [!A1]   LDH.D1T1      *-A20[0x2],A4
00001648 00000001  ||         NOP
0000164C 00000000  ||         NOP
00001650 91D08045      [!A1]  LDH.D1T1      *-A20[0x4],A3
00001654 03208859  ||         MAX2.L1       A4,A8,A6
00001658 935452D6  || [!A1]   STH.D2T2      B6,*++B21[0x2]
0000165C 9814C0C7      [!A1]  LDH.D2T2      *-B5[0x6],B16
00001660 93D08044  || [!A1]   LDH.D1T1      *-A20[0x4],A7
00001664 9394E0C7      [!A1]  LDH.D2T2      *-B5[0x7],B7
00001668 03C2485B  ||         MAX2.L2       B18,B16,B7
0000166C 93D06044  || [!A1]   LDH.D1T1      *-A20[0x3],A7
00001670 038CF85B             MAX2.L2X      B7,A3,B7
00001674 08502247  ||         LDH.D1T2      *+A20[0x1],B16
00001678 039516C4  ||         LDH.D2T1      *B5++[0x8],A7
0000167C 081A1859             MAX2.L1X      A16,B6,A16
00001680 08CA185B  ||         MAX2.L2X      B16,A18,B17
00001684 041460C7  ||         LDH.D2T2      *-B5[0x3],B8
00001688 02D04244  ||         LDH.D1T1      *+A20[0x2],A5
0000168C 021CF5E1             SUB.S1X       A7,B7,A4
00001690 0C11385B  ||         MAX2.L2X      B9,A4,B24
00001694 018E4859  ||         MAX2.L1       A18,A3,A3
00001698 0914E0C7  ||         LDH.D2T2      *-B5[0x7],B18
0000169C 03511644  ||         LDH.D1T1      *A20++[0x8],A6
000016A0 0942785B             MAX2.L2X      B19,A16,B18
000016A4 081E4859  ||         MAX2.L1       A18,A7,A16
000016A8 0814C0C7  ||         LDH.D2T2      *-B5[0x6],B16
000016AC 0BD08044  ||         LDH.D1T1      *-A20[0x4],A23
000016B0 0948C5E3             SUB.S2        B6,B18,B18
000016B4 0340E85B  ||         MAX2.L2       B7,B16,B6
000016B8 08C4E859  ||         MAX2.L1       A7,A17,A17
```

Figure 5.13: The assembly code of *Gamma*() function (3/6).

```
;
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 1028
;*      Loop opening brace source line   : 1029
;*      Loop closing brace source line   : 1035
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 120
;*      Known Maximum Trip Count         : 120
;*      Known Max Trip Count Factor      : 120
;*      Loop Carried Dependency Bound(^) : 5
;*      Unpartitioned Resource Bound     : 16
;*      Partitioned Resource Bound(*)    : 16
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                    12       12
;*      .S units                     1        0
;*      .D units                    16*      16*
;*      .M units                     0        0
;*      .X cross paths               3        9
;*      .T address paths            16*      16*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        2     (.L or .S unit)
;*      Addition ops (.LSD)          3        4     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             7        7
;*      Bound(.L .S .D .LS .LSD)    11       12
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 16 Schedule found with 2 iterations in parallel
;*      Done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
: v5.1.0 Wed Jun 04 10:24:09 2008

)5 Texas Instruments Incorporated
                                        PAGE   78

;*      Collapsed prolog stages     : 1
;*      Minimum required memory pad : 0 bytes
;*
;*      For further improvement on this loop, try option -mh16
;*
;*      Minimum safe trip count     : 1 (after unrolling)
;*----------------------------------------------------------------------------*
L69:    ; PIPED LOOP PROLOG
;** ---------------------------------------------------------------------------*
L70:    ; PIPED LOOP KERNEL
```

Figure 5.14: The assembly code of *Gamma*() function (4/6).

```
000016BC 039480C7  ||         LDH.D2T2      *-B5[0x4],B7
000016C0 08D04044  ||         LDH.D1T1      *-A20[0x2],A17
000016C4 09A5185B             MAX2.L2X      B8,A9,B19
000016C8 04C4A859  ||         MAX2.L1       A5,A17,A9
000016CC C07B1021  || [ A0]   BDEC.S1       L69,A0
000016D0 039500C7  ||         LDH.D2T2      *-B5[0x8],B7
000016D4 04504044  ||         LDH.D1T1      *-A20[0x2],A8
000016D8 09CE885B             MAX2.L2       B20,B19,B19
000016DC 9912CAD7  || [!A1]   STH.D2T2      B18,*+B4[B22]
000016E0 04A6E5E1  ||         SUB.S1        A23,A9,A9
000016E4 01986859  ||         MAX2.L1       A3,A6,A3
000016E8 02506044  ||         LDH.D1T1      *-A20[0x3],A4
000016EC 031A285B             MAX2.L2       B17,B6,B6
000016F0 08CC05A3  ||         NEG.S2        B19,B17
000016F4 9212EAD5  || [!A1]   STH.D2T1      A4,*+B4[B23]
000016F8 018D05E1  ||         SUB.S1        A8,A3,A3
000016FC 04D06047  ||         LDH.D1T2      *-A20[0x3],B9
00001700 0240B858  ||         MAX2.L1X      A5,B16,A4
00001704 08C3185B             MAX2.L2X      B24,A16,B17
00001708 0318E5E3  ||         SUB.S2        B7,B6,B6
0000170C 91CEAA55  || [!A1]   STH.D1T1      A3,*+A19[A21]
00001710 98D422D7  || [!A1]   STH.D2T2      B17,*+B21[0x1]
00001714 019A4858  ||         MAX2.L1       A18,A6,A3
00001718 80844F01     [ A1]   MPYSU.M1      2,A1,A1
0000171C 034525E3  ||         SUB.S2        B9,B17,B6
00001720 94CECA55  || [!A1]   STH.D1T1      A9,*+A19[A22]
00001724 931020D7  || [!A1]   STH.D2T2      B6,*-B4[0x1]
00001728 0A48F85B  ||         MAX2.L2X      B7,A18,B20
0000172C 02906858  ||         MAX2.L1       A3,A4,A5
00001730 09CC81A1             ADD.S1        4,A19,A19
00001734 934C0257  || [!A1]   STH.D1T2      B6,*+A19[0x0]
00001738 021081A3  ||         ADD.S2        4,B4,B4
0000173C 0814E0C5  ||         LDH.D2T1      *-B5[0x7],A16
00001740 09C8F85B  ||         MAX2.L2X      B7,A18,B19
00001744 019E4858  ||         MAX2.L1       A18,A7,A3
00001748           DW$L$_GAMMA$4$E:
00001748 0F802129             MVK.S1        0x0042,A31
0000174C 02DE4859  ||         MAX2.L1       A18,A23,A5
00001750 04D04045  ||         LDH.D1T1      *-A20[0x2],A9
00001754 031410DB  ||         NEG.L2X       A5,B6
00001758 0314C0C6  ||         LDH.D2T2      *-B5[0x6],B6
0000175C 0E01E0A9             MVK.S1        0x03c1,A28
00001760 02504045  ||         LDH.D1T1      *-A20[0x2],A4
00001764 081500C6  ||         LDH.D2T2      *-B5[0x8],B16
00001768 0E81E029             MVK.S1        0x03c0,A29
0000176C 03208859  ||         MAX2.L1       A4,A8,A6
00001770 035452D7  ||         STH.D2T2      B6,*++B21[0x2]
00001774 01D08044  ||         LDH.D1T1      *-A20[0x4],A3
```

Figure 5.15: The assembly code of *Gamma*() function (5/6).

```
          DW$L$_GAMMA$4$E:
          ;*----------------------------------------------------------------------
          ;*   SOFTWARE PIPELINE INFORMATION
          ;*
          ;*      Loop source line                 : 1038
          ;*      Loop opening brace source line   : 1039
          ;*      Loop closing brace source line   : 1043
          ;*      Loop Unroll Multiple             : 2x
          ;*      Known Minimum Trip Count         : 120
          ;*      Known Maximum Trip Count         : 120
          ;*      Known Max Trip Count Factor      : 120
          ;*      Loop Carried Dependency Bound(^) : 2
          ;*      Unpartitioned Resource Bound     : 8
          ;*      Partitioned Resource Bound(*)    : 8
sembler PC v5.1.0 Wed Jun 04 10:24:09 2008

) 1996-2005 Texas Instruments Incorporated
                                                PAGE   80

          ;*      Resource Partition:
          ;*                                A-side   B-side
          ;*      .L units                    0        0
          ;*      .S units                    2        2
          ;*      .D units                    7        5
          ;*      .M units                    8*       8*
          ;*      .X cross paths              4        7
          ;*      .T address paths            6        6
          ;*      Long read paths             0        0
          ;*      Long write paths            0        0
          ;*      Logical  ops (.LS)          0        0     (.L or .S unit)
          ;*      Addition ops (.LSD)         8        9     (.L or .S or .D unit)
          ;*      Bound(.L .S .LS)            1        1
          ;*      Bound(.L .S .D .LS .LSD)    6        6
          ;*
          ;*      Searching for software pipeline schedule at ...
          ;*         ii = 8  Schedule found with 3 iterations in parallel
          ;*      Done
          ;*
          ;*      Epilog not removed
          ;*      Collapsed epilog stages     : 0
          ;*
          ;*      Prolog not entirely removed
          ;*      Collapsed prolog stages     : 1
          ;*
          ;*      Minimum required memory pad : 0 bytes
          ;*
          ;*      For further improvement on this loop, try option -mh8
          ;*
          ;*      Minimum safe trip count     : 2 (after unrolling)
          ;*----------------------------------------------------------------------
          L71:    ; PIPED LOOP EPILOG AND PROLOG

OF802129            MVK     .S1     66,A31
```

Figure 5.16: The assembly code of *Gamma*() function (6/6).

Table 5.12: Improved CTC Decoding Speed Base on 3L-Measured Execution Times for One Information Data Block of 480 Bits

Number of Iterations	Modulation	Execution Time (ms)	Channel Data Rate (Kbps)	Information Data Rate (Kbps)
2	QPSK	0.84	1714.29	571.43
	16QAM	0.85	1694.13	564.71
	64QAM	0.84	1714.29	571.43
4	QPSK	1.6	900	300
	16QAM	1.59	905.66	301.89
	64QAM	1.59	905.66	301.89

Table 5.13: CTC Code Sizes

Operation	Original Code (bytes)	Improved Code (bytes)	Percentage Increase (%)
Encoder with Modulation	3104	3104	0
Decoder with Demodulation	20032	29024	44.89

Table 5.11 compares the original cycles to the improved cycles for *Duo_Binnary_CR SC_Decoder* function and *CTC_Decoder* function. Note that *CTC_Decoder* is used to implement turbo decoding algorithm and *Duo_Binnary_CRSC_Decoder* function is used to implement BCJR algorithm.

Table 5.12 shows the decoding speed of our CTC decoder on chip for two iterations and four iterations. Compared with Table 5.6, there is approximately a 10 times speed up in decoding rate.

In Table 5.13, we show the code sizes of the original and the improved codes. As Seen, we have improved the speed performance at the expense of an increased code size.

To further improve the speed, one possible way is to examine every function and loop to improve its software pipelinability.

5.3 Comparison of Speed of Current Codes

The major purpose of the section is to investigate processing rate in CTC and CC. Base on the comparison of processing rate between CC and CTC, taken as the important reference for improving our CTC decoding processing rate in the future. Beside, we also compare the number of adders and multipliers between CC and CTC, and compare the decoder's processing rates for tail-biting CC, CTC, and LDPC on DSP.

5.3.1 The Views of Block Decoder for Processing Rate

Above all, we can get the executive times of rate 1/2 CC decoder and our rate 1/3 CTC decoder are 0.3811 ms and 1.5914 ms in QPSK modulation, respectively. Their decoding code length are 288 information bits and 480 information bits. Therefore, we can get their decoding information processing rates which are 756 Kbps and 302 Kbps, respectively.

It is worth noting that the CC decoder is pure decoder without the VCP, and its is not be included the external functions, like as de-randomizer, de-interleaver and de-modulator. Besides, as to the CC decoder which we employ can be referred in [3]. Similarly, our CTC decoder is not included de-modulator. However, the above-mentioned executive times which are measured by the 3L timer.

Second, due to there are 4 iterations for our CTC decoder, the decoding processing rate with one iteration is $302 \cdot 4 = 1208$ Kbps. If the two constituent decoders of our CTC decoder are independent and sequential operation, we can ideal suppose that the processing rate of one constituent decoder is $1208 \div 2 = 604$ Kbps.

Third, if the CC decoder is not considered their tail-biting mechanism, we can ideal suppose its decoding processing rate is 756·(288+[48·2])/288 = 1008 Kbps. To compare with our supposed a CTC constituent decoder is better 1.67 times for decoding processing rate.

Finally, if our CTC constituent decoder can be improve, its processing rate may possibly be achieved the same 1008 Kbps as the CC decoder. Therefore, the processing rate of the CTC decoder can be achieved (1008·2)/4 = 504 Kbps. We do not deny the limitations of this roughly inference, but this may possibly be refer to improve our CTC decoder in the future.

5.3.2 Comparison of Tail-Biting CC and CTC for Adders and Multipliers

We can evaluate roughly the number of adders and multipliers for CC in [3, Fig.4.8] and CTC in Chapter 2 (equations 2.29, 2.30, 2.31, 2.35, 2.36), respectively.

For the parts of CC branch operation, we know the total adders are considered about 1 adder, 2 branches, 64 states, and 384 (288+96) loop bits, which are calculated 1·2·64·384 = 49152. Besides, we know the total multipliers are considered about 2 multipliers, 2 branches, 64 states, and 384 loop bits, which are calculated 2·2·64·384 = 98304.

For the parts of CTC branch operation, we see that the total adders are considered about 4 adders, 4 branches, 8 states, and 240 loop bits, as can calculated 4·4·8·240 = 30720. Besides, we see that the total multipliers are considered about 5 multipliers, 4 branches, 8 states, and 240 loop bits, which are calculated 5·4·8·240 = 38400. Note that we also use one shifted operation for multiplying 0.5 in (2.29).

For the parts of CC decoder, we know the total adders are considered about 2 adders, 2 branches, 64 states, and 384 loop bits, which are calculated 2·2·64·384 = 98304. Besides, we know the total multipliers are considered about 2 multipliers, 2 branches, 64 states, and 384

Table 5.14: CC and CTC for Adder and Multiplier (Numbers)

Function	Adders	Multipliers
CC Branch	49,152	98,304
CTC Branch	30,720	38,400
CC Decoder	98,304	98,304
CTC Constituent Decoder(A)	65,760	41,280
CTC Decoder 4 Iterations (B)	526,080	330,240
CTC (A)/CC Complexity	0.67	0.42
CTC (B)/CC Complexity	5.35	3.36

loop bits, which are calculated $2{\cdot}2{\cdot}64{\cdot}384 = 98304$.

For the parts of CTC constituent decoder, we see that the adders of (2.29) are considered about 4 adders, 4 branches, 8 states, and 240 loop bits. The adders of (2.30), and (2.31) are considered about 2 adders, 4 branches, 8 states, and 240 loop bits. The adders of (2.35) are considered about 2 adders, 4 branches, 8 states, 3 subtractions, and 240 loop bits. The adders of (2.36) are considered about 3 adders, 2 subtractions, 3 branches, and 240 loop bits. Hence, we estimate total adders as $(4{\cdot}4{\cdot}8{\cdot}240)+(2{\cdot}4{\cdot}8{\cdot}240)+240{\cdot}(2{\cdot}4{\cdot}8+3)+(5{\cdot}3{\cdot}240) = 30720 + 15360 + 16080 + 3600 = 65760$.

Besides, we see that the multipliers of (2.29) are considered about 5 multipliers, 4 branches, 8 states, and 240 loop bits. The multipliers of (2.36) are considered about 4 multipliers, 3 branches, and 240 loop bits. Note that we also use two shifted operations for multiplying 0.5. Hence, we estimate total multipliers as $(5 \cdot 4 \cdot 8 \cdot 240) + (4 \cdot 3 \cdot 240) = 38400 + 2880 = 41280$.

However, for the sake of providing a visual picture of the numbers, consider the graphic representation in Table 5.14.

Table 5.15: Information Data Processing Rate Calculated from CCS for One Information Data Block of 480 Bits

Number of Iterations	QPSK Demodulation Cycles	CTC_Decoder Cycles	Overall Decoder Cycles	Information Data Rate (Kbps)
2	1,468	294,232	295,700	1,623
4	1,468	588,664	590,132	813

Table 5.16: Comparison of Decoder Speed for Tail-Biting CC, CTC, and LDPC Calculated from CCS

CC Information Data Rate Without using VCP for Rate-1/2 QPSK (Kbps) [3]	CC Information Data Rate With VCP for Rate-1/2 QPSK (Kbps)	CTC Information Data Rate for Rate-1/3 QPSK with 4 Iterations (Kbps)	LDPC Information Data Rate for rate-1/2 QPSK (Kbps) [3]
832	8,938	813	7.6

5.3.3 Comparison of Decoder Speed for Tail-Biting CC, CTC, and LDPC

We can get information data processing rates in decoding for tail-biting CC and LDPC code from [3]. Since we can use CCS's profile to estimate the decoding processing rate for CTC, comparing its decoding processing rate with LDPC's.

Table 5.15 shows we use the cycles of Table 5.11 to estimate information data processing rate. In Table 5.16, we show the decoder's processing rates for rate-1/2 tail-biting CC without using the VCP, rate-1/2 tail-biting CC with VCP, rate-1/3 CTC with 4 iterations and rate-1/2 LDPC. Such processing rates underscore the importance of using DSP hardware's

acceleration.

Chapter 6

Conclusion and Future Work

This primary research questions that contained two issues of IEEE 802.16e FEC in this thesis were (a) the research in convolution code with tail-biting and implementation on the VCP of TI's C6416 for the WiMAX applications, and (b) the max-log-MAP decoding research of the CTC and implementation on DSP.

In the first issue, we first analyzed and studied TI EDMA to employ the VCP based on convolution code with tail-biting in AWGN. In our implementation, the convolution coding gain in AWGN was less than theoretic value by 0.1 to 2.6 dB. When we converted the fixed-point to the VCP application, the performance was almost the same and we could just use S2.5 for BM truncation to implement the decoder. Finally, in our decoder with the VCP, we can approach data rates between 6.2 and 9.2 Mbps for CCS profile. However, we can also utilize the 3L timer to measure, approaching data rates about between 2.6 and 3.5 Mbps. Therefore, under CCS and 3L, as the measurements indicate the decoder processing rates are improved significantly by about 9.8 and 4.7 times, respectively, with use of VCP.

In the second issue, we first evaluated the performance of CTC and compared the results with the numerical results. The coding gain of CTC was much better than convolution code. Then we focused on max-log-MAP decoding algorithm. Then we converted the floating-point

to fixed-point, and we could use S12.3 and S11.4 to implement the decoder for QPSK and 16QAM (64QAM), respectively. In conclusion, in the encoder, we can approach data rates between 7.7 to 9.1 Mbps and in our decoder with 4 iterations, we can approach data rate about 300K bps.

In the future work, further optimization of the programs may be possible. For example, if we can parallelize the execution of the peripheral functions and the VCP, we may get approach an information data rate of about 6 Mbps in decoding under 3L. However the interested readers can refer to "Continuous Decoding" mechanism in [21] to study.

In CTC, there are three possible methods to enhance our DSP implementation. First, we may rewrite our code in our CTC, there are too much dependence to execute for *Alpha* and *Beta* loop. These execute too many cycles and cause software pipelined worse. However, one possible way is to examine every function and loop to improve its software pipelinability. Second, if we need further reducing complexity by max-log-MAP decoding algorithms, [28] is one of the references. Third, we can examine the TI's C6416 TCP (Turbo -decoder coprocessor) [15]. The TCP is a programmable peripheral for decoding IS2000/3GPP turbo code, integrated in into C6416 DSP. The coprocessor operates two modes, standalone processing mode and share processing mode, which are detailed discussed in [15]. It may be of interest for using TCP to be helpful in raising the decoding speed, but how to process the IO relationship for double-binary circular recursive systematic convolutional code on the TCP application is a tough problem.

Bibliography

[1] IEEE Std 802.16e, *Draft Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Broadband Wireless Access Systems.* New York: IEEE, Oct. 2007.

[2] Yu-Ping Ho, "Study on OFDM signal description and channel coding in the IEEE 802.16a TDD OFDMA wireless communication standard," M.S. thesis, Dept. of Electronics Eng., National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2003.

[3] Po-Sheng Wu, "Research in and DSP implementation of channel techniques for IEEE 802.16e OFDMA," M.S. thesis, Dept. of Electronics Eng., National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2007.

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

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

[6] 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.

[7] 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.

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

[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] B. Baumgartner, M. Reinhardt, G. Richter, and M. Bossert, "Performance of forward error correction for IEEE 802.16e," *10th International OFDM Workshop,* Hamburg, Germany, Aug. 2005.

[11] Todd K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms.* Wiley, 2005.

[12] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Info. Theory*, vol. 20, pp. 284–287, Mar. 1974.

[13] Texas Instruments, *Implementing a MAP Decoder for cdma2000 Turbo Codes on a TMS320C62x DSP Device.* Lit. no. SPRA629, May 2000.

[14] M. R. Soleymani, Y. Gao, and Y. Vilaipornsawai, *Turbo Coding for Satellite and Wireless Communications*, Dordrechtthe, Netherlands: Kluwer Academic, 2002.

[15] Texas Instruments, *Using TMS320C6416 Coprocessors:Turbo Coprocessor (TCP).* Lit. no. SPRA749b, Aug. 2006.

[16] M. C. Valenti, S. Cheng, and R. Iyer Seshadri, *Turbo Code Applications: A Journey from a Paper to Realization.* Springer, 2005.

[17] C. Berrou, M. Jezequel, C. Douillard, and S.Kerouedan, "The advantages of non-binary turbo codes," *Proc. IEEE Information Theory Workshop*, pp. 61–63, Sept. 2001

[18] Sundance, SMT6400 Help.chm.

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

[20] S. Raiagopal *et al.*, "A 600 MHz VLIW DSP," in Digest of Technical Papers, *IEEE International Solid-State Circuits Conference,* vol. 1, 2002, pp. 56–444.

[21] Texas Instruments, *Using TMS320C6416 Coprocessors: Viterbi Coprocessor (VCP).* Lit. no. SPRU750D, Sep. 2003.

[22] Texas Instruments, *TMS320C6000 Chip Support Library API Reference Guide.* Lit. no. SPRU401J, Aug. 2004.

[23] Texas Instruments, *TMS320C6000 Peripherals Reference Guide.* Lit. no. SPRU190D, Feb. 2001.

[24] Texas Instruments, *Applications Using the TMS320C6000 Enhanced DMA*. Lit. no. SPRA636, Oct. 2001.

[25] 3L Diamond Company Homepage: http://www.3l.com/Diamond/Diamond.htm

[26] 3L Diamond, *Diamond User Guide: Sundance Edition V3.0*. Jan. 2005.

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

[28] Texas Instruments, *Implementing a MAP Decoder for cdma2000 Turbo Codes on TMS320C62x DSP Device.* Lit. no. SPRA629, May 2000.

作者簡歷

姓名：陳佳楓 (Jia-Fong Chen)

出生地：台灣省新竹縣竹北市

學歷：國立新竹中學

海洋大學電機工程系學士

交通大學電子研究所碩士(2006.9~2008.6)

研究領域：通訊系統、通道編碼及數位訊號處理

論文題目：WiMAX 通道編碼技術

與數位訊號處理器實現之探討

(Study in WiMAX Channel Coding Techniques and

Associated Digital Signal Processor Implementation)