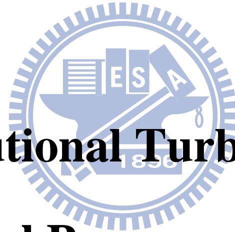
國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

WiMAX 迴旋渦輪碼技術與

數位訊號處理器實現

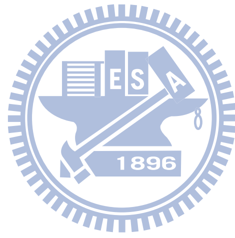


WiMAX Convolutional Turbo Code Technology and Digital Signal Processor Implementation

研 究 生：曾劭學

指導教授：林大衛 博士

中 華 民 國 九 十 八 年 十 月



WiMAX 迴旋渦輪碼技術與

數位訊號處理器實現

WiMAX Convolutional Turbo Code Technology and Digital Signal Processor Implementation

研究生:曾劭學　　　Student: Shao-Hsueh Tseng

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

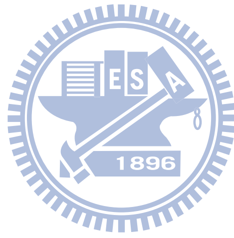
國立交通大學

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
in
Electronics Engineering
October 2009
Hsinchu, Taiwan, Republic of China

中華民國九十八年十月



WiMAX 迴旋渦輪技術
與數位訊號處理器實現

研究生:曾劭學　　　　　　　　　　指導教授:林大衛 博士

國立交通大學

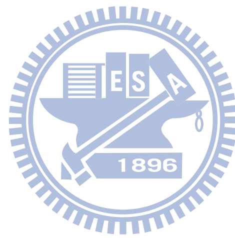
電子工程學系 電子研究所碩士班

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更正編碼的機制，藉此降低通訊頻道中雜訊失真的影響。通道編碼為本論文的重點。

本篇論文在於研究 IEEE 802.16e OFDMA 所訂定的迴旋渦輪碼（CTC）系統並實現在數位訊號處理器(DSP)。闡明迴旋渦輪碼的雙二位元循環遞迴系統迴旋（duo-binary CRSC）編碼以及利用最大對數事後機率(max-log-MAP)進行 BCJR (Bahl `Cock `Jelinek 和 Raviv 四位研究者做為命名)解碼演算法。我們利用 C 語言驗證系統演算法上的正確性，以及補償 max-log-MAP 導致的效能損失，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，我們改變格子圖順序，以及利用 DSP 內建函式達到平行運算，並且有效改善解碼器的運算速度。原始解碼器的部分僅可達到約每秒 800K 位元的處理速度，改善後解碼器的速度增進約 2 倍，進而可以達到每秒 1500K 位元的處理速度。



WiMAX Convolutional Turbo Code Technology and Digital Signal Processor Implementation

Student: Shao-Hsueh Tseng Advisor: Dr. David W. Lin

Department of Electronics Engineering

& Institute of Electronics

National Chiao Tung University

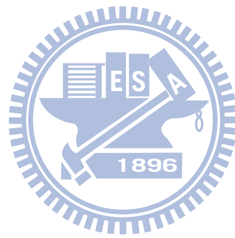
Abstract

In the IEEE 802.16e wireless communication standard, a forward error correction (FEC) mechanism is presented are the transmitter side to reduce the noisy channel effect. The focus is on the channel coding.

The focus of this thesis 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 use BCJR decoding algorithm by max-log-MAP. We employ the C program to insure the correctness of our algorithm and compensate the performance loss by max-log-MAP, furthermore, simulate the CTC for different modulations in AWGN.

Then, we implement on TI C6416 DSP, changing trellis order and using intrinsic function to achieve parallel operation. Therefore, we improve decoder operation speed efficiently. For original decoder just can achieved a processing rate of 800 Kbps . For improved decoder , which is approximately 2 times speed up in decoding rate. Therefore, the decoder

can achieve a further data processing rate of 1500 Kbps.



誌謝

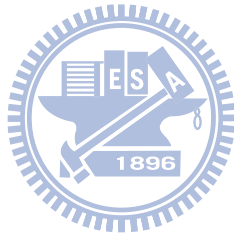
本篇論文的完成，首先要特別感謝我的指導教授林大衛博士，在我進入交大電子所開始，不論是課業或是研究上的困難，老師總能細心的給予適時方向去解決問題，使我學到了分析以及解決問題的能力。此外老師對於學生的認真以及親切樂觀的態度更是深遠影響了我，使我在研究所的這幾年得到了學術以外更重要的智慧。

此外，由衷的感謝通訊電子與訊號處理實驗室所有的成員，包含各位師長、同學、學長姐以及學弟妹們。特別感謝吳俊榮學長、林鴻志學長、王海薇學姐和盧世榮學弟對我在學術上的不吝指導與建議，謝謝你們幫我解決了許多通訊方面的疑問。感謝 95 級佳楓學長的指導，96 級凱暐、志偉、豐進、清德、辰彥等實驗室成員，以及陳紹基老師的學生嘉洵跟靖順，平日和我一起念書、討論、玩耍，讓我的研究生涯擁有美好的回憶。期待各位夥伴們畢業後都有不錯的發展。

最後，必須感謝我的家人，我父母親給予我的支持，使我在外地讀書時能無後顧之憂，感謝他們的支持，也謝謝所有幫助過我、陪我走過這段歲月的人

曾劭學

民國 98 年 11 月　於新竹



Contents

1 Introduction **1**

1.1 Scope of the Work 1

1.2 Organization of this Thesis 2

2 Overview of CTC in IEEE 802.16e OFDMA **3**

2.1 Convolution Turbo Code Specification [1] 3

2.1.1 Randomizer [1] 4

2.1.2 CTC Encoder In IEEE 802.16e OFDMA [1] 6

2.1.3 1/3 CTC Encoder [1] 8

2.1.4 CTC Interleaver [1] 10

2.1.5 CTC Tail-Biting [1], [4] 11

2.1.6 Subpacket Generation (Channel Interleaver or Interleaver and Puncturing) [1] 14

2.1.7 Modulation [1] 19

2.1.8 Demodulation for Bit-Interleaved Coded Modulation [3] 19

2.2 Decoding of CTC 23

2.2.1 The Turbo Decoding Algorithm [5] 23
2.2.2 Decoding Rule for CRSC Codes with Non-binary Trellis [8] 25
2.2.3 Simplified Max-Log-MAP Algorithm for Double-Binary CTC [8] . . . 28
3 DSP Implementation Environment **33**
3.1 The DSP Board [12] 33
3.2 The DSP Chip [12] 34
3.2.1 Central Processing Unit [12] 36
3.2.2 Memory [14] 39
3.3 TI's Code Development Environment [15] 40
3.4 Code Development Flow [17] 42
3.5 Code Optimization on TI DSP Platform 42
3.5.1 Compiler Optimization Options [17] 44
3.5.2 Software Pipelining [18] 46
3.5.3 Macros and Intrinsic Functions [17] 47
4 Fixed-Point Implementation of CTC Encoder and Decoder **48**
4.1 Performance in AWGN Channel with Floating-Point Processing 48
4.2 Performance in AWGN Channel with Fixed-Point Processing 49
4.2.1 Scaling Method [22] 53
4.2.2 Clipping Method [19], [20] 60
5 Speeding Up of DSP Implementation **67**

5.1 Speed of DSP [17] 67
5.2 Original State Order [22] 71
5.3 Arrange State Order to Achieve Parallelism 73
5.4 Comparison of Speed 78
5.4.1 Comparison of Original and Improved Codes in Additions, Multiplications and Intrinsic Functions 78
5.4.2 Processing Rate of CTC Decoder 86

6 Conclusion and Future Work **89**
6.1 Conclusion 89
6.2 Future Work 90

Bibliography **91**

List of Figures

2.1 Use of CTC in transmitter and receiver of IEEE 802.16e OFDMA (from [1]). 4

2.2 PRBS for data randomization (from [1], Fig. 337). 6

2.3 Structure of CTC in transmitter ans decoding in receiver (based on [1]). . . . 7

2.4 CTC encoder (base on [1]). 9

2.5 CTC rate 1/3 encoder flow chart [22]. 9

2.6 CTC encoding slot concatenation for different rates [1]. 11

2.7 CTC channel coding per modulation (modified from [1]). 12

2.8 CTC interleaver in two steps (modified from [1]). 13

2.9 Block diagram of CTC channel interleaving scheme (from [1]). 17

2.10 QPSK, 16-QAM, and 64-QAM constellations (from [1]). 20

2.11 Metric partitions of the 16-QAM constellation (from [3]). 23

2.12 Block diagram of a turbo decoder (from [5]). 24

2.13 CTC trellis structure of duo-binary convolutional code with feedback encoder (from [8]). 26

3.1 Sundance's SMT395 module (from [11]). 34

3.2 Functional block and CPU (DSP core) diagram [13]. 38

3.3 C64x cache memory architecture [14]. 40

3.4 Code development flow for C6000 [17]. 43

3.5 Software-pipelined loop [17]. 46

4.1 Performance of CTC at different ρ values under three modulations with 288 information bits. 50

4.2 Performance of CTC at different ρ values under three different modulations with 432 information bits. 51

4.3 Performance of CTC at 288-bit and $\rho = 0.75$ with different modulations employing floating-point computation at 4 iterations. 52

4.4 Performance of CTC at 432-bit and $\rho = 0.75$ with different modulations employing floating-point computation at 4 iterations. 52

4.5 Hypothetical reference CTC decoder implementation with marking of fixed-point data format at various place. 53

4.6 CTC fixed-point truncation parameters (modified from [22]). 54

4.7 Illustration of fixed-point data formats with the scaling method, where Q11.4 may be replaced by other setting (such as Q9.6 or Q14.1) depending on code rate and operating condition. 55

4.8 CTC decoding at different bit numbers with different modulations. 56

4.9 Performance with scaling of various quantities in CTC decoding to avoid overflow at high SNR. 57

4.10 Performance of CTC with different scale factors under three modulations with 288 information bits. 58

4.11 Performance of CTC with different scale factors under three modulations with 432 information bits. 59

4.12 Fixed-point data format with the clipping method. 61

4.13 Performance of CTC at different clipping ranges under three modulations with 288 information bits. 63

4.14 Performance of CTC at different clipping ranges under three modulations with 432 information bits. 64

4.15 Performance of rate 1/2 CTC with 288 information bits with floating-point decoding vs. fixed-point under clipping method. 65

4.16 Performance of rate 3/4 CTC with 432 information bits with floating-point decoding vs. fixed-point under clipping method. 66

5.1 Graphical representation of the $_amem4()$ and the $_max2()$ intrinsics [17]. . . 69

5.2 Graphical representation of the $_dotp2()$ intrinsic [17]. 69

5.3 Graphical representation of $_packXX2()$ intrinsics[17]. 70

5.4 Overall encoder and decoder architecture. 71

5.5 Trellis diagram, every branch in the trellis connecting at time $k-1$ to a state at time k. 74

5.6 Arrangement of trellis order for forward and backward metrics. 75

5.7 Use of the $_packXX2()$ intrinsics for forward metric 75

5.8 Improved C code for the alpha() function. 78

5.9 Assembly code of the alpha() function (1/5). 79

5.10 Assembly code of the alpha() function (2/5). 80

5.11 Assembly code of the alpha() function (3/5). 81

5.12 Assembly code of the alpha() function (4/5). 82

5.13 Assembly code of the alpha() function (5/5). 83

5.14 Software pipeline information of the alpha() function. 84

List of Tables

2.1 CTC Channel Coding Schemes for Each Modulation Method 5

2.2 Circulation State Look-Up Table (S_{C1} and S_{C2}) [1, Table 573] 14

2.3 Parameters for the Subblock Interleavers 16

2.4 Bit Metric for Method-ML and Method-LLR 22

2.5 Amount of Additions, Multiplications and Max Operations for Soft-Output Decoding of One Component Code Once, Where Number of Information Bits = 480 32

3.1 Functional Units and Operations Performed [12] 37

4.1 Coding Gain Performance of Rate-1/2 CTC in AWGN at BER = 10^{-5} with Floating-Point and Fixed-Point with Scaling Method Computation 60

4.2 Coding Gain Performance of Rate-3/4 CTC in AWGN at BER = 10^{-5} with Floating-Point and Fixed-Point with Scaling Method Computation 60

4.3 Coding Gain at Rate 1/2 with 288 Information Bits CTC in AWGN at BER = 10^{-4} with Floating-Point Computation and Fixed-Point Computations with Scaling Method and Clipping Method 62

4.4 Coding Gain at Rate 3/4 with 432 Information Bits CTC in AWGN at BER = 10^{-4} with Floating-Point Computation and Fixed-Point Computations with Scaling Method and Clipping Method Computation 62

5.1 TMS320C64X Compiler Intrinsics [17]. 68

5.2 Overall Encoder and Decoder Block Cycles 72

5.3 Speed Up in Channel Interleaver 72

5.4 Profile of *Duo_Binnary_CRSC_decoder* with QPSK Modulation for 480 Information Bits, Rate 1/2 Coding in One Iteration 73

5.5 Profile of Improve *Duo_Binnary_CRSC_Decoder* with QPSK Modulation for 480 Information Bits, Rate 1/2 Coding in One Iteration 77

5.6 Speed Up in Decoding of One Data Block with QPSK Modulation for One Iteration 77

5.7 Numbers of Intrinsic calls and arithmetic operations in Original Code for CTC Decdoding 85

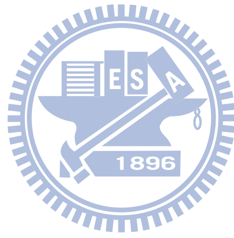
5.8 Numbers of Intrinsic Calls and Arithmetic Operation in Improved Code . . . 86

5.9 Information Data Processing Rate Calculated from CCS for Original Code for 480 Information Bits, Rate 1/2 Coding 87

5.10 Information Data Processing Rate Calculated from CCS for Improved Code for 480 Information Bits, Rate 1/2 Coding 87

5.11 Comparison of Decoder Speeds for Tail-Biting CC, CTC, and LDPC Calculated from CCS 88

1896



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 802.16 standard [1] 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 802.16.

CTC (convolutional turbo codes) comprise the mandatory channel coding schemes in

Mobile WiMAX. In addition, the puncture and M-ary modulation are used after encoder. A number of studies have been conducted using BCJR algorithm [6] as the turbo decoding. There have been numerous studies in the literature dealing with different decoding algorithm. However we need to reduce the complexity for actual digital signal processor (DSP) implementation. For convolutional turbo codes, we arrange trellis order to achieved parallel operation.

In this thesis, we focuss on the study of the simulation and the DSP implementation of the CTC in the IEEE 802.16 standard, We first study the encoding and decoding techniques. Then we perform computer simulation to investigate the coding performance. Finally, we optimize CTC on the DSP with fixed-point computation.

1.2 Organization of this Thesis

This thesis is organized as follows.

- Chapter 2 introduces the CTC (convolutional turbo codes) of IEEE 802.16e specifications.
- Chapter 3 describes the DSP implementation environment.
- Chapter 4 discusses simulation and DSP implementation of the CTC.
- Chapter 5 discusses the optimization of CTC decoder on DSP.
- Chapter 6 contain the conclusion and future work.

Chapter 2

Overview of CTC in IEEE 802.16e OFDMA

The convolutional turbo code (CTC) is one mandatory channel coding scheme in Mobile WIMAX. In this chapter, We introduce the encoding and the decoding methods for the CTC in IEEE 802.16e OFDMA.

ES A 1896

2.1 Convolution Turbo Code Specification [1]

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 turbo encoder. The block-by-block coding makes the convolutional turbo code effectively a block code. However, we do not implement the repetition block, which can be used to further increase the signal-to-noise-ratio (SNR) margin over the modulation and FEC mechanisms, for the channel coding procedures in IEEE 802.16e. Repetition block can be applied only to QPSK modulation. Reader interested in the repetition block can refer to relevant material in [1].

To make the system more flexibly adaptable to the channel condition, 32 coding-modulation schemes are defined in IEEE 802.16e, as shown in Table 2.1. The different coding rates are

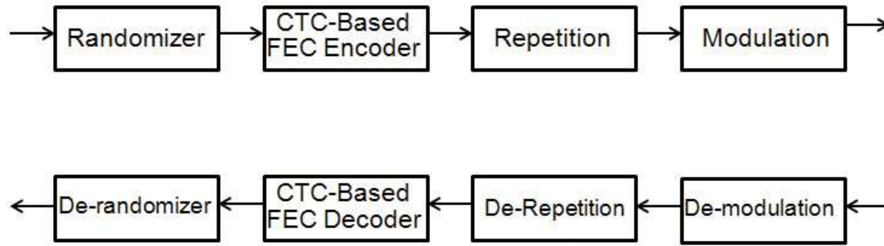


Figure 2.1: Use of CTC in transmitter and receiver of IEEE 802.16e OFDMA (from [1]).

made by puncturing of the native convolutional turbo code. The puncturing mechanism in convolutional turbo coding can provide variable code rates through one convolutional turbo 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 downlink (DL) and uplink (UL), except the frame control header (FCH). The randomization is initialized on each FEC block.

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 the 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

Table 2.1: CTC 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	48	1/2	96	8
QPSK	54	1/2	108	9
QPSK	60	1/2	120	10
QPSK	9	3/4	12	1
QPSK	18	3/4	24	2
QPSK	27	3/4	36	3
QPSK	36	3/4	48	4
QPSK	45	3/4	60	5
QPSK	54	3/4	72	6
16QAM	12	1/2	24	1
16QAM	24	1/2	48	2
16QAM	36	1/2	72	3
16QAM	48	1/2	96	4
16QAM	60	1/2	120	5
16QAM	18	3/4	24	1
16QAM	36	3/4	48	2
16QAM	54	3/4	72	3
64QAM	18	1/2	36	1
64QAM	36	1/2	72	2
64QAM	54	1/2	108	3
64QAM	24	2/3	36	1
64QAM	48	2/3	72	2
64QAM	27	3/4	36	1
64QAM	54	3/4	72	2
64QAM	30	5/6	36	1
64QAM	60	5/6	72	2

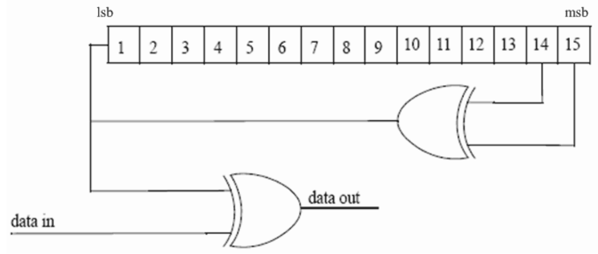


Figure 2.2: PRBS for data randomization (from [1], Fig. 337).

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).

We do not implement the hybrid automatic repeat request (HARQ) mechanism. In HARQ the randomizer can be initialized with different vector, so the detail are given in [1] for HARQ required, which can refer to [1] in detail.

2.1.2 CTC Encoder In IEEE 802.16e OFDMA [1]

The convolutional turbo code (CTC) defined in IEEE 802.16e OFDMA is shown in Fig. 2.3. The input data are first encoded by the CTC encoder. Then, they are interleaved by the interleaving block and followed by puncturing. 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.

Turbo code was first proposed 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

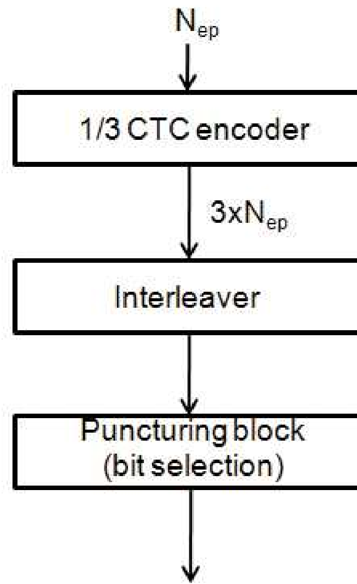


Figure 2.3: Structure of CTC in transmitter ans decoding in receiver (based on [1]).

standards, including DVB-RCS (Digital Video Broadcasting—Return Channel via Satellite) and WiMAX (IEEE 802.16e).

IEEE 802.16e employs the double-binary code, whose advantages over a binary code include [10]:

- Better convergence.
- 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 [9, 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 [9, Chapter 12].

A more detailed understanding of this relationship can be gained form [10].

2.1.3 1/3 CTC Encoder [1]

The CTC encoder, including its constituent encoder, is shown in Figure 2.4. 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 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 $1 + D^3$.

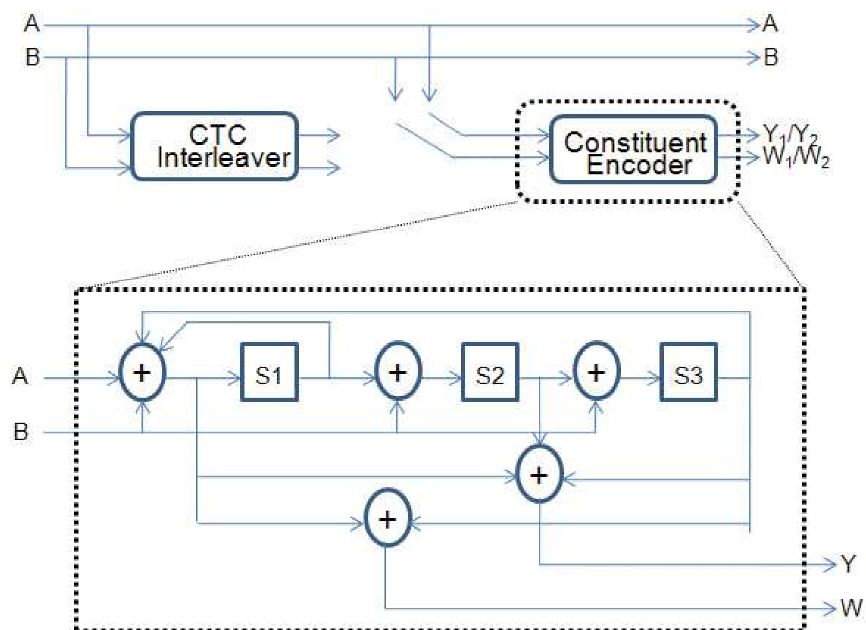


Figure 2.4: CTC encoder (base on [1]).

1896

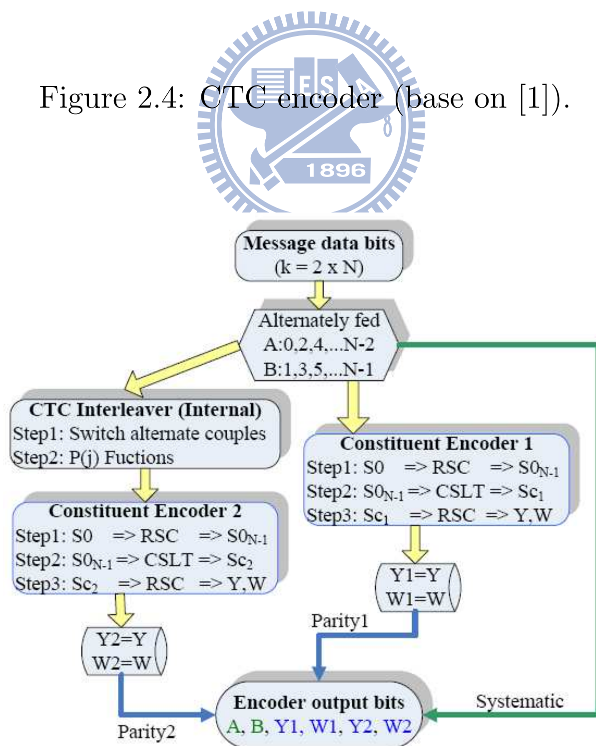


Figure 2.5: CTC rate 1/3 encoder flow chart [22].

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 $j = 0, \ldots, N-1$, 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 =

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

We can represent the above rule with the flow chart shown as Fig. 2.5. Note that "CSLT" stand for *c*irculation *s*tate *l*ook-up *t*able, as shown in Table 2.2.

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. 2.6. The concatenation rule shall not be used when using (incremental redundancy) IR HARQ.

2.1.4 CTC Interleaver [1]

The interleaver requires the parameters P_0, P_1, P_2, and P_3 shown in Fig. 2.7, 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. 2.8, where two possible errors in the standard is indicated.

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

Figure 2.6: CTC encoding slot concatenation for different rates [1].

2.1.5 CTC Tail-Biting [1], [4]

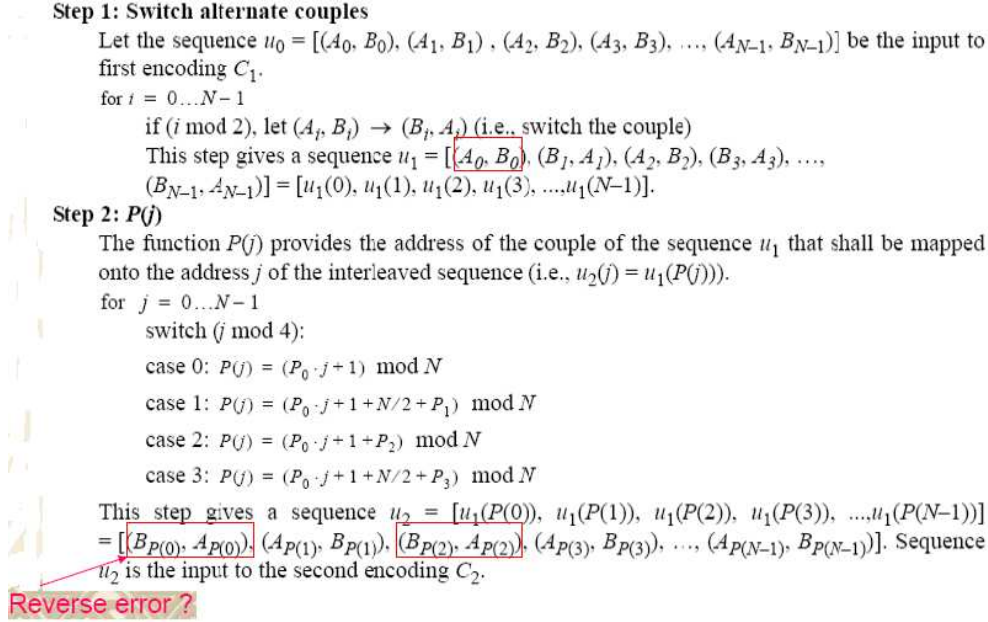
For recursive encoders, tail-biting is not as easy as it is for non-recursive encoders. To ensure that the starting state is the same as the ending state, which is called circulation state, for recursive encoders an initial encoding of the whole sequence has to be performed [4].

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 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 2.2.

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

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 2.7: 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: $P(j) = (P_0 \cdot j + 1) \mod N$

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

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

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

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.

Figure 2.8: CTC interleaver in two steps (modified from [1]).

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 [4].

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. 2.4. 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 $S0_{N-1}$.

Table 2.2: Circulation State Look-Up Table (S_{C1} and S_{C2}) [1, Table 573]

$N \bmod 7$	$S0_{N-1}$							
	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

- According to the length N of the sequence, use Table 2.2 to find S_{C1} and S_{C2}.

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

The proposed FEC structure in IEEE 802.16e OFDMA 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.

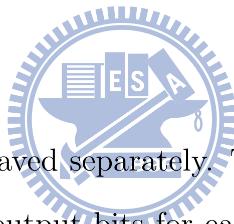
The subpacket is also used in HARQ packet transmission. Figure 2.3 shows the block diagram of subpacket generation. A rate-1/3 CTC encoded codeword goes through interleaving and puncturing. Figure 2.9 shows the block diagram of the interleaving block. The puncturing is performed to select a consecutive interleaved bit sequence that starts at some point in the 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 2.3 gives these parameters.

2. Initialize i and k to 0.

Table 2.3: Parameters for the Subblock Interleavers

Block size (bits) N_{EP}	Subblock interleaver 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

3. Form a tentative output address T_k according to

$$T_k = 2^m (k \bmod J) + BRO_m(\lfloor k/J \rfloor) \tag{2.1}$$

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 consists 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.

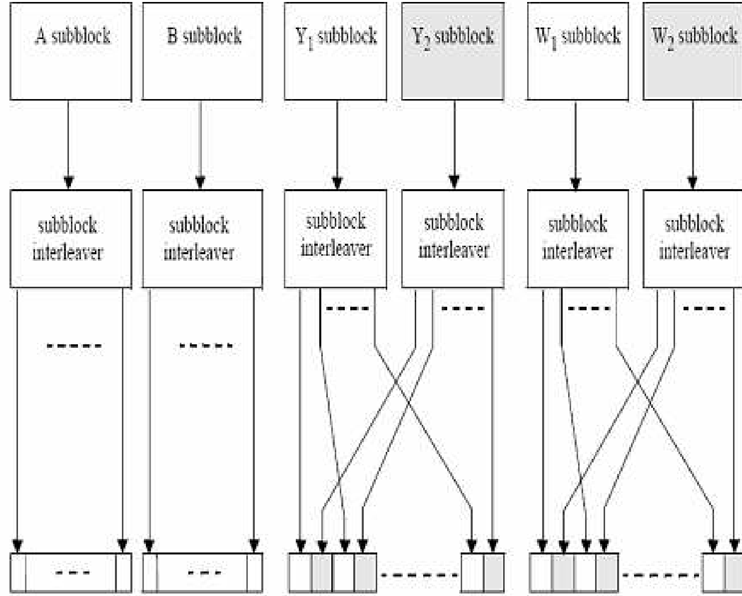


Figure 2.9: Block diagram of CTC channel interleaving scheme (from [1]).

The bit-by-bit multiplexed sequence of interleaved $Y1$ and $Y2$ subblock sequences consists 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 second output bit from the $Y2$ subblock interleaver, etc. The bit-by-bit multiplexed sequence of interleaved $W1$ and $W2$ subblock sequences consists 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 2.9 shows the interleaving scheme. The order of bit grouping sequence is as follows:

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

Bit Selection

Lastly, bit selection is performed to generate the subpacket. The puncturing block is referred to as bit 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 is 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, and 6 for 64QAM).
- $SPID_k$: the subpacket ID for the kth subpacket (for the first subpacket, $SPID_{k=0}$=0).

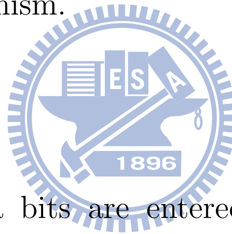
Also, let the scrambled and selected bits be numbered from zero with the 0th bit being the first bit in the sequence. Then, the index of the ith bit for the kth subpacket shall be

$$S_{k,i} = (F_k + i)\text{mod}(3 \cdot N_{EP}) \tag{2.2}$$

where $i = 0, \ldots, L_k - 1$, $L_k = 48 \cdot N_{SCHk} \cdot m_k$, and $F_k = (SPID_k \cdot L_k) \text{mod}(3 \cdot N_{EP})$. 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.



2.1.7 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. 2.10 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.

2.1.8 Demodulation for Bit-Interleaved Coded Modulation [3]

Let $a[i] = a_I[i] + ja_Q[i]$ denote the QAM symbol transmitted via 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–carrier interference) are completely eliminated, we can write the received signal of the sub-carrier

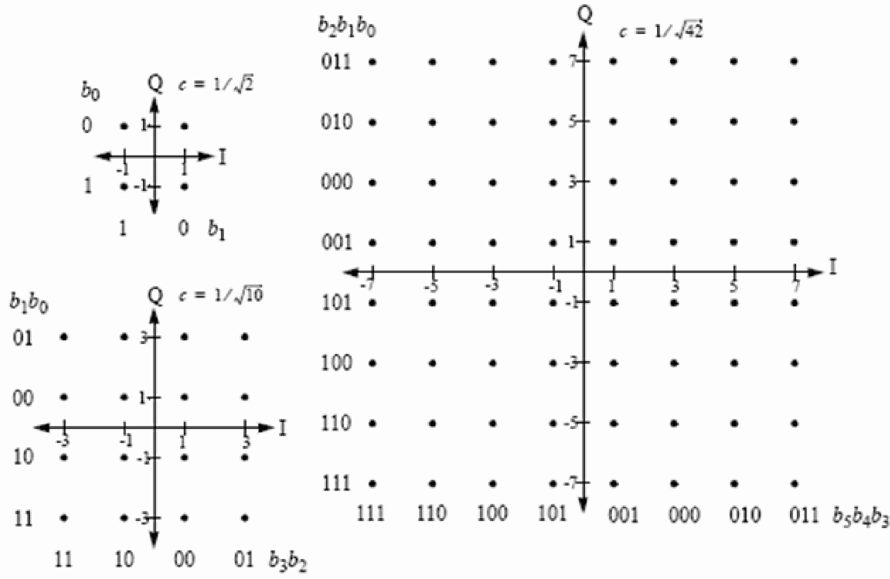


Figure 2.10: QPSK, 16-QAM, and 64-QAM constellations (from [1]).

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 (2.32) are equivalent to

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

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

From the concept of log-likelihood ratio (LLR), a method named *Method-LLR* is proposed in [3] to reduce the complexity of *Method-ML*. It defines $LLR(b_{I,k})$ as

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

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

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 2.11 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. As a consequence,

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

can be simplified as follows.

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

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

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

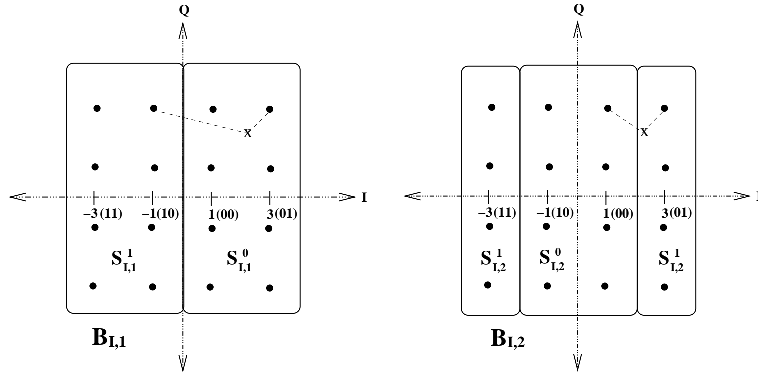


Figure 2.11: Metric partitions of the 16-QAM constellation (from [3]).

For 64-QAM,

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

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

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

2.2 Decoding of CTC

2.2.1 The Turbo Decoding Algorithm [5]

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

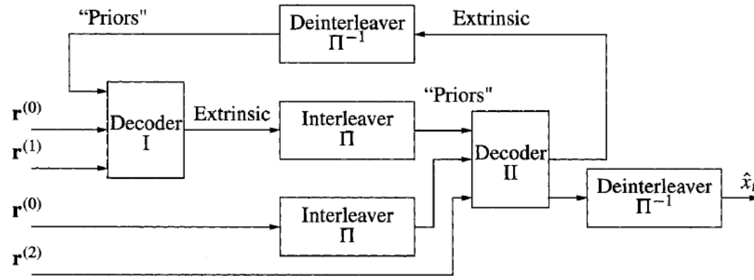


Figure 2.12: Block diagram of a turbo decoder (from [5]).

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. 2.12.

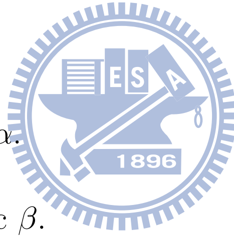
The MAP Decoding Algorithm [5], [7]

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 originally proposed it in 1974 [6]. 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.
- Compute extrinsic log likelihood ratio L_e.



A more detailed understanding can be gained from [5].

2.2.2 Decoding Rule for CRSC Codes with Non-binary Trellis [8]

The trellis of a double-binary feedback convolutional encoder has the structure shown in Fig. 2.13. The goal of the MAP algorithm is to provide us with

$$\begin{aligned} L_i(d_k) &= \ln \frac{P_r[d_k = i|\text{Observation}]}{P_r[d_k = 0|\text{Observation}]} \\ &= \ln \frac{\sum_{d_k=i}^{(S_{k-1},S_k)} p(S_{k-1}, S_k, \{y_k\})}{\sum_{d_k=0}^{(S_{k-1},S_k)} p(S_{k-1}, S_k, \{y_k\})}, \quad i = 1, 2, 3, \end{aligned} \tag{2.16}$$

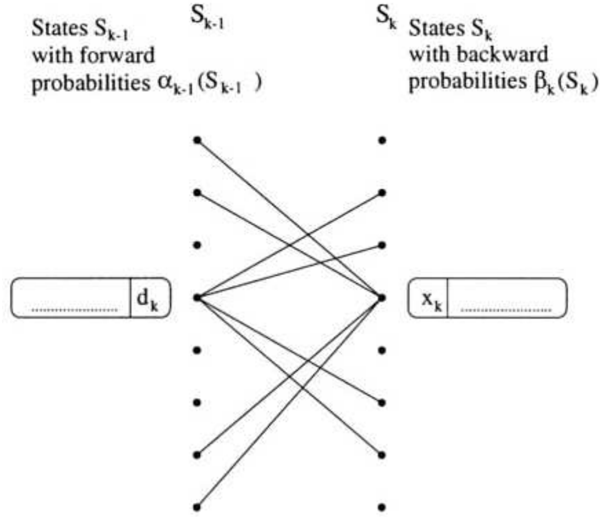


Figure 2.13: CTC trellis structure of duo-binary convolutional code with feedback encoder (from [8]).

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 (2.16) is taken over all path 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

$$\begin{aligned} p(S_{k-1}, S_k, \{y_k\}) &= p(S_{k-1}, y_{j<k}) \cdot p(S_k, y_k|S_{k-1}) \cdot p(y_{j>k}, S_k) \\ &\triangleq \alpha_{k-1}(S_{k-1}) \cdot \gamma_k(S_{k-1}, S_k) \cdot \beta_k(S_k) \end{aligned} \tag{2.17}$$

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

$$\alpha_k(S_k) = \sum_{S_{k-1}} \alpha_{k-1}(S_{k-1}) \cdot \gamma_k(S_{k-1}, S_k). \tag{2.18}$$

The backward recursion yields

$$\beta_{k-1}(S_{k-1}) = \sum_{S_k} \gamma_k(S_{k-1}, S_k) \cdot \beta_k(S_k). \tag{2.19}$$

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

$$\begin{aligned}\gamma_k(S_{k-1}, S_k) &= p(S_k, y_k|S_{k-1}) \\ &= p(S_k|S_{k-1}) \cdot p(y_k|S_{k-1}, S_k) \\ &= P(d_k) \cdot p(y_k|d_k). \end{aligned} \tag{2.20}$$

Let the natural logarithm of the branch transition probability metric be

$$\Gamma_k(S_{k-1}, S_k) = \ln \gamma_k(S_{k-1}, S_k) \tag{2.21}$$

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

$$\begin{aligned}A_k(S_k) &= \ln \alpha_k(S_k) \\ &= \ln \sum_{S_{k-1}} e^{A_{k-1}(S_{k-1})+\Gamma_k(S_{k-1},S_k)}, \end{aligned} \tag{2.22}$$

$$\begin{aligned}B_{k-1}(S_{k-1}) &= \ln \beta_{k-1}(S_{k-1}) \\ &= \ln \sum_{S_k} e^{\Gamma_k(S_{k-1},S_k)+B_k(S_k)}. \end{aligned} \tag{2.23}$$

Then the log-likelihood ratios (2.16) for $i = 1, 2, 3$ are given by

$$\begin{aligned}L_i(d_k) &= \ln \frac{\sum_{d_k=i}^{(S_{k-1},S_k)} p(S_{k-1}, S_k, \{y_k\})}{\sum_{d_k=0}^{(S_{k-1},S_k)} p(S_{k-1}, S_k, \{y_k\})} \\ &= \ln \frac{\sum_{d_k=i}^{(S_{k-1},S_k)} \alpha_{k-1}(S_{k-1}) \cdot \gamma_k^i(S_{k-1}, S_k) \cdot \beta_k(S_k)}{\sum_{d_k=0}^{(S_{k-1},S_k)} \alpha_{k-1}(S_{k-1}) \cdot \gamma_k^0(S_{k-1}, S_k) \cdot \beta_k(S_k)} \\ &= \ln \frac{\sum_{d_k=i}^{(S_{k-1},S_k)} e^{A_{k-1}(S_{k-1})+\Gamma_k^i(S_{k-1},S_k)+B_k(S_k)}}{\sum_{d_k=0}^{(S_{k-1},S_k)} e^{A_{k-1}(S_{k-1})+\Gamma_k^0(S_{k-1},S_k)+B_k(S_k)}}. \end{aligned} \tag{2.24}$$

2.2.3 Simplified Max-Log-MAP Algorithm for Double-Binary CTC [8]

Implementing (2.24) 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 (2.20) and (2.21),

$$\begin{aligned}\Gamma_k(S_{k-1}, S_k) &= \ln \gamma_k(S_{k-1}, S_k) \\ &= \ln[p(y_k|d_k) \cdot P(d_k)]. \end{aligned} \tag{2.25}$$

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

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

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,

$$\begin{aligned}\Gamma_k(S_{k-1}, S_k) &= \ln[p(y_k|d_k) \cdot P(d_k)] \\ &= 0.5 \cdot L_c \cdot [y_k^{s,I} \cdot x_k^{s,I}(i) + y_k^{s,Q} \cdot x_k^{s,Q}(i) + y_k^{p,I} \cdot x_k^{p,I}(i, S_{k-1}, S_k) \\ &\quad + y_k^{p,Q} \cdot x_k^{p,Q}(i, S_{k-1}, S_k)] + \ln P(d_k) + K \end{aligned} \tag{2.26}$$

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

$$\begin{aligned} A_k(S_k) &= \ln \sum_{S_{k-1}} e^{A_{k-1}(S_{k-1})+\Gamma_k(S_{k-1},S_k)} \\ &\approx \max_{S_{k-1}}[A_{k-1}(S_{k-1}) + \Gamma_k(S_{k-1}, S_k)], \end{aligned} \tag{2.27}$$

$$\begin{aligned} B_{k-1}(S_{k-1}) &= \ln \sum_{S_k} e^{\Gamma_k(S_{k-1},S_k)+B_k(S_k)} \\ &\approx \max_{S_k}[\Gamma_k(S_{k-1}, S_k) + B_k(S_k)]. \end{aligned} \tag{2.28}$$

The above can be derived by considering the Jacobian logarithm [5], i.e.,

$$\ln(e^{L1} + e^{L2}) = \max(L_1, L_2) + \ln(1 + e^{-|L_1-L_2|}). \tag{2.29}$$

If the correction term (i.e., the second right-hand-side [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

$$A_0(S_0) = A_N(S_N) \quad \forall S_0, \tag{2.30}$$

$$B_N(S_N) = B_0(S_0) \quad \forall S_N. \tag{2.31}$$

As a result, the log-likelihood ratios in (2.24) reduce to

$$\begin{aligned} L_i(d_k) \approx & \max_{(S_{k-1},S_k)}[A_{k-1}(S_{k-1}) + \Gamma_k^i(S_{k-1}, S_k) + B_k(S_k)] \\ & - \max_{(S_{k-1},S_k)}[A_{k-1}(S_{k-1}) + \Gamma_k^0(S_{k-1}, S_k) + B_k(S_k)]. \end{aligned} \tag{2.32}$$

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 [8]. It turns out that the extrinsic information can be expressed as

$$\begin{aligned} L_i^e(\hat{d}_k) = & L_i(\hat{d}_k) - 0.5 \cdot [y_k^{s,I} \cdot x_k^{s,I}(i) + y_k^{s,Q} \cdot x_k^{s,Q}(i)] \\ & +0.5 \cdot [y_k^{s,I} \cdot x_k^{s,I}(0) + y_k^{s,Q} \cdot x_k^{s,Q}(0)] - \ln \frac{P[d_k = i]}{P[d_k = 0]}. \end{aligned} \tag{2.33}$$

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

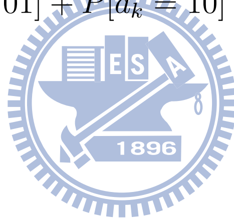
$$L_i^a(d_k) \quad = \quad \ln \frac{P[d_k = i]}{P[d_k = 0]} \tag{2.34}$$

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

$$\begin{aligned}
P[d_k = 01] &= e^{L_1^a(d_k)} \cdot P[d_k = 00], \\
P[d_k = 10] &= e^{L_2^a(d_k)} \cdot P[d_k = 00], \\
P[d_k = 11] &= e^{L_3^a(d_k)} \cdot P[d_k = 00],
\end{aligned}$$

and

$$P[d_k = 00] + P[d_k = 01] + P[d_k = 10] + P[d_k = 11] \quad = \quad 1, \tag{2.35}$$



we have

$$\begin{aligned}
P[d_k = 00] &= \frac{1}{1 + e^{L_1^a(d_k)} + e^{L_2^a(d_k)} + e^{L_3^a(d_k)}}, \\
P[d_k = 01] &= \frac{L_1^a(d_k)}{1 + e^{L_1^a(d_k)} + e^{L_2^a(d_k)} + e^{L_3^a(d_k)}}, \\
P[d_k = 10] &= \frac{L_2^a(d_k)}{1 + e^{L_1^a(d_k)} + e^{L_2^a(d_k)} + e^{L_3^a(d_k)}}, \\
P[d_k = 11] &= \frac{L_3^a(d_k)}{1 + e^{L_1^a(d_k)} + e^{L_2^a(d_k)} + e^{L_3^a(d_k)}}.
\end{aligned} \tag{2.36}$$

Using max-function approximation yields

$$\begin{aligned}
\ln P[d_k = 00] &= -\max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)], \\
\ln P[d_k = 01] &= L_1^a(d_k) - \max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)], \\
\ln P[d_k = 10] &= L_2^a(d_k) - \max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)], \\
\ln P[d_k = 11] &= L_3^a(d_k) - \max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)]. \qquad (2.37)
\end{aligned}$$

Assuming equally likely symbols initially, we have

$$A_0(S_0) = 0 \quad \forall S_0, \qquad (2.38)$$

$$B_N(S_N) = 0 \quad \forall S_N, \qquad (2.39)$$

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

After sufficient decoding iterations, the decisions are made according to

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

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

This above algorithm has 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. Table 2.5 shows the complexity analysis. We will discuss later the simulation results and the speed of our DSP implementation.

Table 2.5: Amount of Additions, Multiplications and Max Operations for Soft-Output Decoding of One Component Code Once, Where Number of Information Bits = 480

	max's	additions	multiplications
branch metric	2880	31680	30720
forward metric	7440	7680	0
backward metric	7440	7680	0
LLR	6720	16080	0
extrinsic	0	3600	0

Chapter 3

DSP Implementation Environment

In this chapter, our discussion will concentrate on the DSP system development environment, DSP chip and its features because our implementation is software-based on the DSP. The software development tool, Code Composer Studio (CCS), is also introduced.

3.1 The DSP Board [12]

The DSP card used in our implementation is Sundance's SMT395 as shown in Fig. 3.1 [11]. 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 to enable multi-DSP systems with minimum development efforts by the programmers.

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.
- Xlilinx Virtex II Pro FPGA XC2VP30-6 in FF896 package.
- 256 Mbytes of SDRAM at 133MHz.
- Eight 2 Gbit/sec Rocket serial links (RSL) for inter module communication.

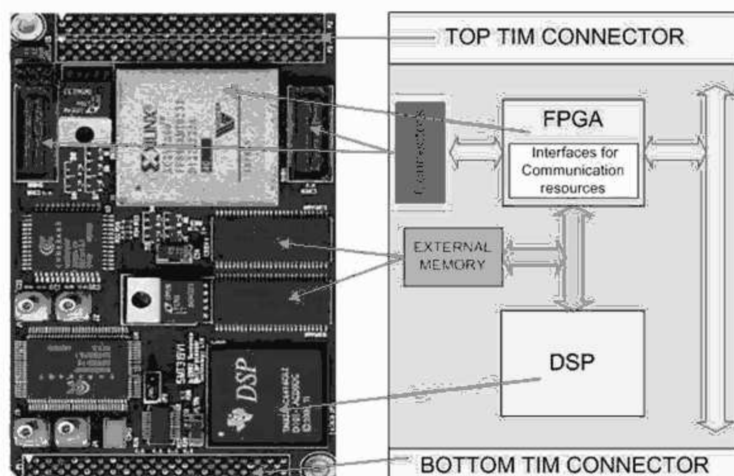
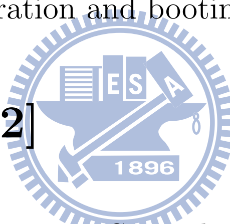


Figure 3.1: Sundance's SMT395 module (from [11]).

- Two Sundance High-speed Bus (50,100 or 200 MHz) ports at 32 bits width.
- 8-Mbyte flash ROM for configuration and booting.



3.2 The DSP Chip [12]

The TMS320C64x DSP is a fixed-point DSP in the TMS320C64x series of the TMS320C6000 DSP platform family. The TMS320C64x device use the very-long-instruction-word (VLIW) architecture developed by TI. The C6416 device has two high-performance embedded coprocessors, Viterbi Decoder Coprocessor (VCP) and Turbo Decoder Coprocessor (TCP) that can significantly speed up channel-decoding operations on chip. However the TCP is designed appropriately for the 3GPP standard and its parameter setting cannot be used to the CTC in 802.16e. Therefore, we cannot employ the TCP in our implementation.

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

- The eight functional units include two multipliers and six arithmetic units:

- Execute up to eight instructions per cycle.
- Allow designers to develop highly effective RSIC-like code for fast development time.

- Instruction packing:
  - Gives code size equivalence for eight instructions executed serially or in parallel.
  - Reduces code size, program fetches, and power consumption.
- Conditional execution of all instructions:
  - Reduces costly branching.
  - Increases parallelism for higher sustained performance.
- Efficient code execution on independent functional units:
  - Efficient C compiler on DSP benchmark suite.
  - Assembly optimizer for fast development and improved parallelization.
- 8/16/35/64-bit data support, providing efficient memory support for a variety of application.
- 40-bit arithmetic options add extra precision for applications requiring it.
- Saturation and normalization provide support for key arithmetic operations.
- Field manipulation and instruction extract, set, clear, and counting support common operation found in control and data manipulation application.
- 32×32-bit integer multiply with 32- or 64-bit result.

The C64x additional features include:

- Each multiplier can perform two 16x16 bits or four 8x8 bits multiplies every clock cycle.
- Quad 8-bit and dual 16-bit instructions set extensions with data flow support.
- Special communication-specific instruction have been added to address common operations inerror-correcting codes.
- Bit count and rotate hardware extends support for bit-level algorithms.

In the following subsections, two major parts of TMS320C64x DSP are introduced respectively. They are the central processing unit and memory .

3.2.1 Central Processing Unit [12]

The C64x DSP core contains 64 32-bit general purpose registers, program fetch unit, instruction decode unit, two data paths each with four function units, control register, control logic, test unit, emulation logic and interrupt logic. The program fetch, instruction fetch and instruction decode units can arrange eight 32-bit instructions to the eight function units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B) shown in Fig. 3.2, each of which contains four functional units and one register file. The four functional units are: one unit for multiplier operations (.M), one for arithmetic and logic operation (L.), one for branch, byte shifts, and arithmetic operation (.S), and one for linear and circular address calculation to load and store with external memory operations (.D). The details of the function units are described in Table 3.1.

Each register file consists of 32 $\times$ 32-bit registers. Each function unit in the two sets of four functional units reads and writes directly within its own data path. That is, functional units .L1, .S1, .M1 and .D1 can only write to register file A. The same holds for register file B. However, two cross-paths (1X and 2X) allow functional units from one data path to

Table 3.1: Functional Units and Operations Performed [12]

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

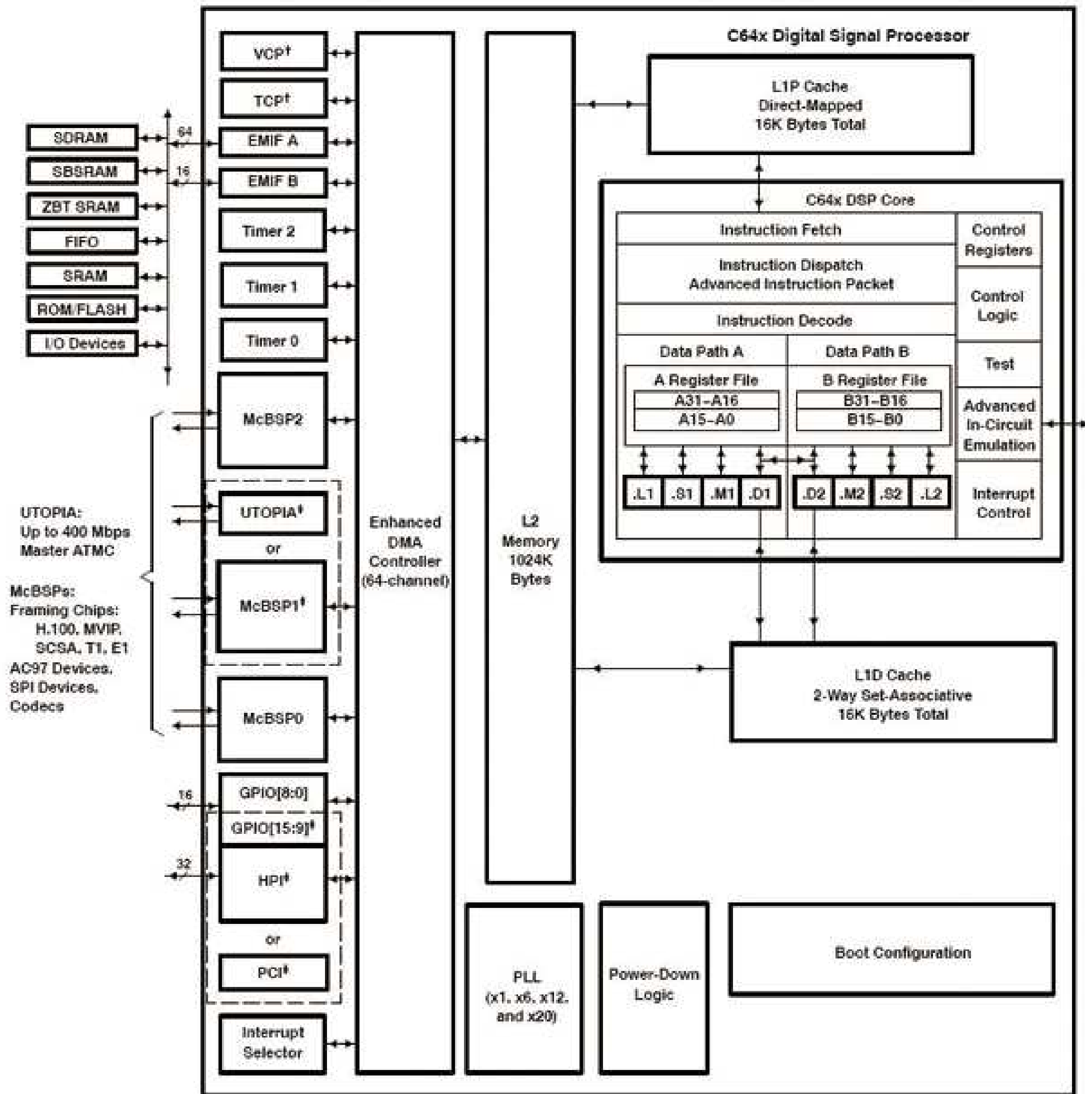


Figure 3.2: Functional block and CPU (DSP core) diagram [13].

access a 32-operand the opposite-side register file. The cross path 1X allows data path A to read its source from register file B. The cross path 2X allows data path B to read its source from register file A. In the C64x, CPU pipelines data-cross-path accesses over multiple clock cycles. This allows the same register to be used as a data-cross-path operand by multiply functional units in the same execute packet.

3.2.2 Memory [14]

Internal Memory

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

Memory Options

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

- Program cache.
- 2-level caches.
- 32-bit external memory interface supports SDRAM, SBRAM, SRAM,

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

Cache Memory

The C64x memory architecture consist of a two-level internal cache-based memory architecture plus external memory. Level cache is split into program (L1P) and data (L1D) caches. The C64x memory architecture is shown in Fig. 3.3. On C64x devices, each L1 cache is 16KB. All caches and data paths are automatically managed by cache controller. Level 1 cache is accessed by the CPU without stalls. Level 2 cache is configurable and can be split

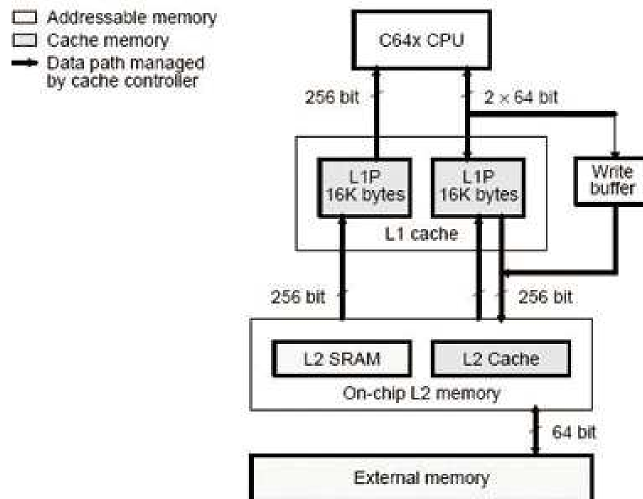


Figure 3.3: C64x cache memory architecture [14].

into L2 SRAM (addressable on-chip memory) and L2 cache for caching external memory locations. On a C6416 DSP, the size of L2 cache is 1 MB, and the external memory can be several Mbytes large. More detailed introduction to the cache system can be found in [14].

3.3 TI's Code Development Environment [15]

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

Some of CCS's fully integrated host tools include:

- Simulators for full device, CPU only and CPU plus memory for optimal performance.

- Integrated visual project manager with source control interface, multi-project support and the ability to handle thousands of project files.
- Source code debugger common interface for both simulator and emulator targets;
  - C/C++/assembly language support.
  - Simple breakpoint.
  - Advanced watch window.
  - Symbol browser.
- DSP/BIOS host tooling support (configure, real-time analysis and debugger).
- Data transfer for real time data exchange between host and target.
- Profiler to understand code performance.

CCS also delivers foundation software consisting of:

- DSP/BIOS kernel for the TMS320C6000 DSPs:
  - Pre-emptive multi-threading.
  - Inter-thread communication.
  - Interrupt handling.
- TMS320 DSP Algorithm Standard to enable software reuse.
- Chip Support Library (CSL) simplify device configuration. CSL provides C-program functions to configure and control on-chip peripherals.

- DSP library for optimum DSP functionality. The library includes many C-callable, assembly-optimized, general-purpose signal-processing and image/video processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical.

3.4 Code Development Flow [17]

The recommended code development flow involves utilizing the C6000 code generation tools to aid in optimization rather than forcing the programmer to code by hand in assembly. Hence the programmer may let the compiler do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation. This simplifies the maintenance of the code, as everything resides in a C framework that is simple to maintain, support, and upgrade. Fig. 3.4 illustrates the three phases in the code development flow. Because phase 3 is usually too detailed and time consuming, most of the time we will not go into phase 3 to write linear assembly code unless the software pipelining efficiency is too bad or the resource allocation is too unbalanced.

3.5 Code Optimization on TI DSP Platform

In this section, we describe several methods that can accelerate our code and reduce the execution time on the C64x DSP. First, we introduce two techniques that can be used to analyze the performance of specific code regions:

- Use the clock( ) and printf( ) functions in C/C++ to time and display the performance of specific code regions. Use the stand-alone simulator (load6x) to run the code for this purpose.

- Use the profile mode of the stand-alone simulator. This can be done by compiling the

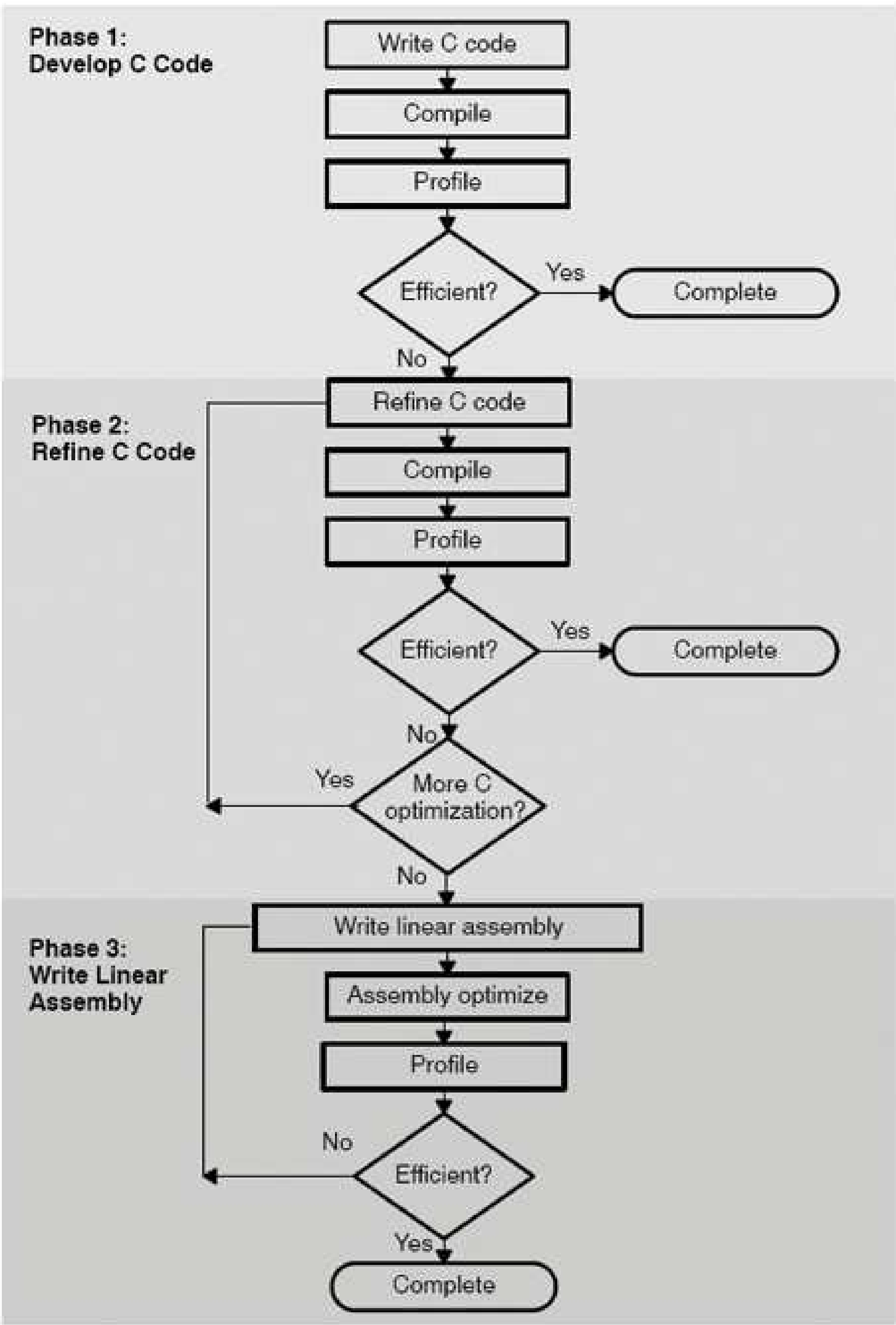


Figure 3.4: Code development flow for C6000 [17].

code with the -mg option and executing load6x with the -g option. Then enable the clock and use profile points and the RUN command in the Code Composer debugger to track the number of CPU clock cycles consumed by a particular section of code. Use "View Statistics" to view the number of cycles consumed.

Usually, we use the second technique above to analyze the C code performance. The feedback of the optimization result can be obtained with the -mw option. It shows some important results of the assembly optimizer for each code section. We take these results into consideration in improving the computational speed of certain loops in our program.

3.5.1 Compiler Optimization Options [17]

In this subsection, we introduce the compiler options that control the operation of the compiler. The 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 or the executing performance.

The major compiler options we use are -o3, -k, -pm -op2, -mh<n>, -mw, and -mi.

- -o*n*: The "*n*" denotes the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization.
  - -o3: highest level optimization, whose main features are:
    * Performs software pipelining.
    * Performs loop optimizations, and loop unrolling.
    * Removes all functions that are never called.
    * Reorders function declarations so that the attributes of called functions are known when the caller is optimized.

    * Propagates arguments into function bodies when all calls pass the same value in the same argument position.
    * Identifies file-level variable characteristics.

- -k: Keep the assembly file to analyze the compiler feedback.
- -pm -op2: In the CCS compiler option, -pm and -op2 are combined into one option.
  - -pm: Gives 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.
- -mh<n>: Allows speculative execution. The appropriate amount of padding, n, must be available in data memory to insure correct execution. This is normally not a problem but must be adhered to.
- -mw: Produce additional compiler feedback. This option has no performance or code size impact.
- -mi: Describes the interrupt threshold to the compiler. If the compiler knows that no interrupts will occur in the code, it can avoid enabling and disabling interrupts before and after software-pipelined loops for improvement in code size and performance. In addition, there is potential for performance improvement where interrupt registers may be utilized in high register pressure loops.

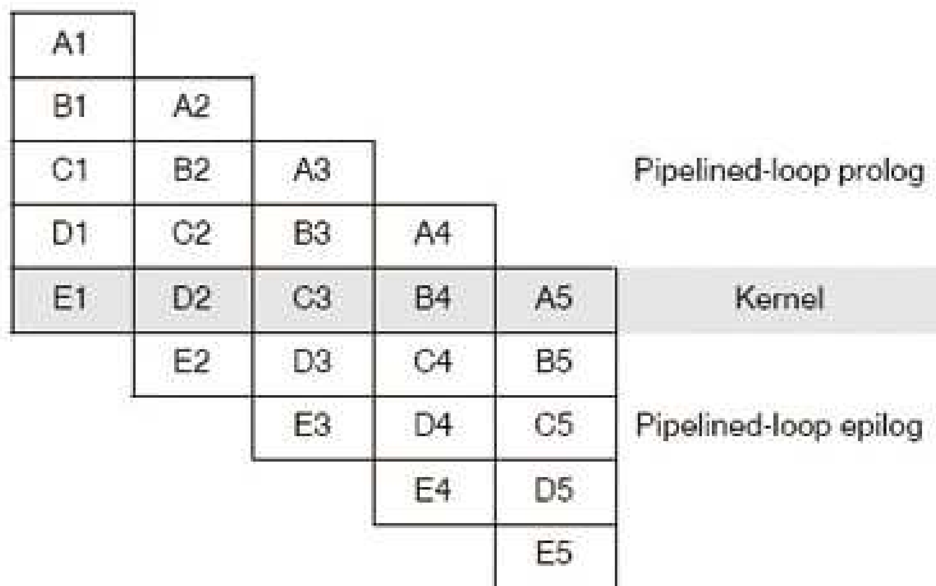


Figure 3.5: Software-pipelined loop [17].

3.5.2 Software Pipelining [18]

Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel. This is the most important feature we rely on to speed up our system. The compiler always attempts to software-pipeline. Fig. 3.5 illustrates a software pipelined loop. The stages of the loop are represented by A, B, C, D, and E. In this figure, a maximum of five iterations of the loop can execute at one time. The shaded area represents the loop kernel. In the loop kernel, all five stages execute in parallel. The area above the kernel is known as the pipelined loop prolog, and the area below the kernel the pipelined loop epilog.

But under the conditions listed below, the compiler will not do software pipelining [17]:

- If a register value lives too long, the code is not software-pipelined.
- If a loop has complex condition code within the body that requires more than five

condition registers, the loop is not software pipelined.

- A software-pipelined loop cannot contain function calls, including code that calls the run-time support routines.
- In a sequence of nested loops, the innermost loop is the only one that can be software-pipelined.
- If a loop contains conditional break, it is not software-pipelined.

Usually, we should maximize the number of loops that satisfy the requirements of software pipelining. Software pipelining is a very important technique for optimization. But how can we get the software pipeline information? The information is located in the .asm file that the compiler generates with the -mw options. In order to view software pipeline information, we must also enable the -k option which can retain the .asm output from the compiler.

3.5.3 Macros and Intrinsic Functions [17]

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

The TI C6000 compiler provides many special functions that map C codes directly to inlined C64x instructions, which increases C code efficiency. These special functions are called intrinsic functions. If some instructions have equivalent intrinsic functions, we can replace them by intrinsic functions and the execution time can be decreased. We will introduce how to use the intrinsic functions in chapter 4.

Chapter 4

Fixed-Point Implementation of CTC Encoder and Decoder

In this chapter, we present some simulation results for the CTC in IEEE 802.16e. They include both floating-point and fixed-point results and DSP implementation results.

4.1 Performance in AWGN Channel with Floating-Point Processing

In this section, we consider the performance of CTC with floating-point processing. In particular, we discuss two important parameters : the iteration number and the compensation factor for max-log-MAP operation. The iteration number of the turbo decoding affects the decoding accuracy and complexity. A large iteration number usually leads to better performance, but the complexity and latency also increase. From [22], we can conclude that reasonable results are obtained with 4 to 8 iterations. To limit the decoding complexity and maintain a reasonable performance, therefore, we choose 4 to be the iteration number in simulation and in DSP implementation. From [8], we can find the performance with log-MAP, but we don't compare in our simulation results.

Now consider the compensation factor for max-log-MAP. Although max-log-MAP algo-

rithm can reduce the implementation complexity, it results in performance loss since the approximated maximum function usually overestimates the messages. In order to compensate the performance loss, one way is to use a scaling factor ρ to scale down the extrinsic value [21] :

$$\ln \frac{P[d_k = i]}{P[d_k = 0]} = \rho \times L_i^e(\hat{d}_k). \tag{4.1}$$

In Figs. 4.1 and 4.2 we compare the performance at $\rho = 0.5, 0.75$ and 1 for code rate 1/2 with 288 information bits and code rate 3/4 with 432 information bits under three different modulation types with max-log-MAP decoding. We can see that the bit error rates (BER) are almost the same for $\rho = 0.5$ and $\rho = 1$, and $\rho = 0.75$ apparently performs better than the other choices. Applying the simple scaling to the extrinsic information improves the BER performance by 0.1 to 0.2 dB.

Figs. 4.3 and 4.4 compare the performance under the three modulations at $\rho = 0.75$ with code rates 1/2 and 3/4. The coding gains under QPSK, 16QAM, and 64QAM at BER = 10^{-5} for code rate 1/2 are 7.05, 8.35 and 9.19 dB, respectively, and that for code rate 3/4 are 5.59, 6.38 and 6.92 dB, respectively.

4.2 Performance in AWGN Channel with Fixed-Point Processing

In algorithm development, it is often convenient to employ floating-point computation to acquire better accuracy. However, for the sake of power consumption, execution speed, and hardware costs, practical implementations usually adopt fixed-point computations. The DSP chip used in our work, TI's TMS320C6416 is also of the fixed-point category and supports 8, 16, 32-bit data precisions, providing efficient memory support for a variety of applications. In our simulations, 32-bit operations should be over the requirement, and 8-bit operations do not give enough precision. Therefore, we choose 16-bit operations, which are

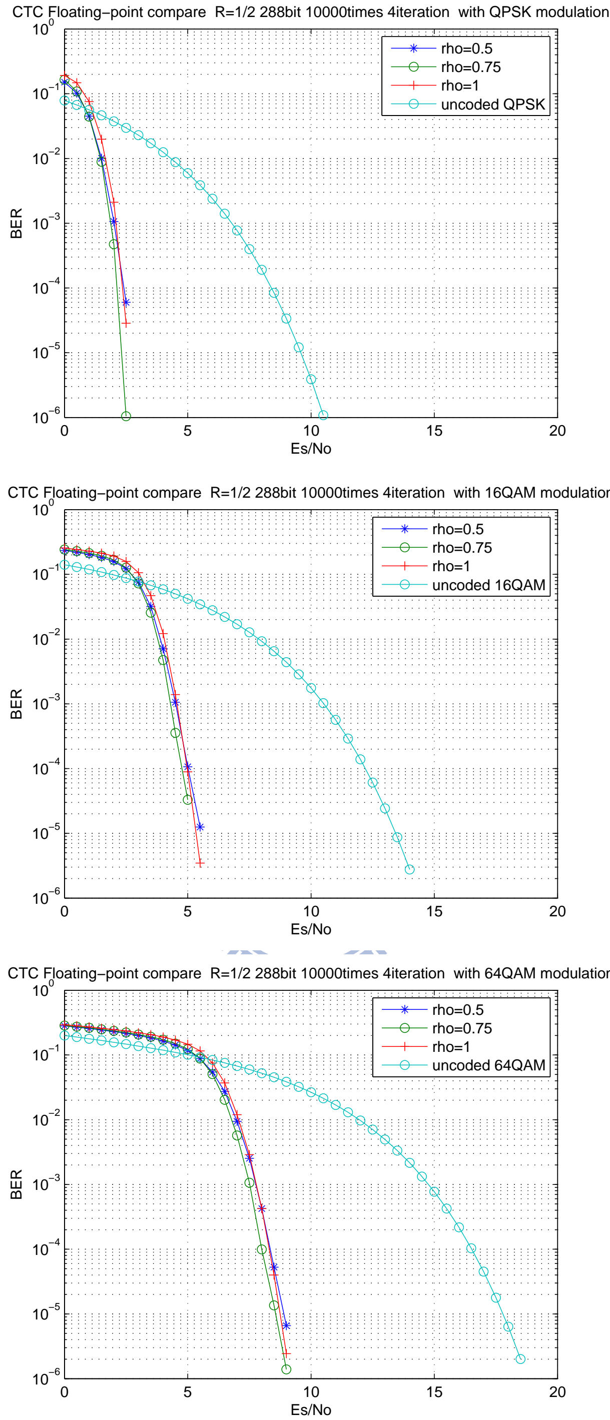


Figure 4.1: Performance of CTC at different ρ values under three modulations with 288 information bits.

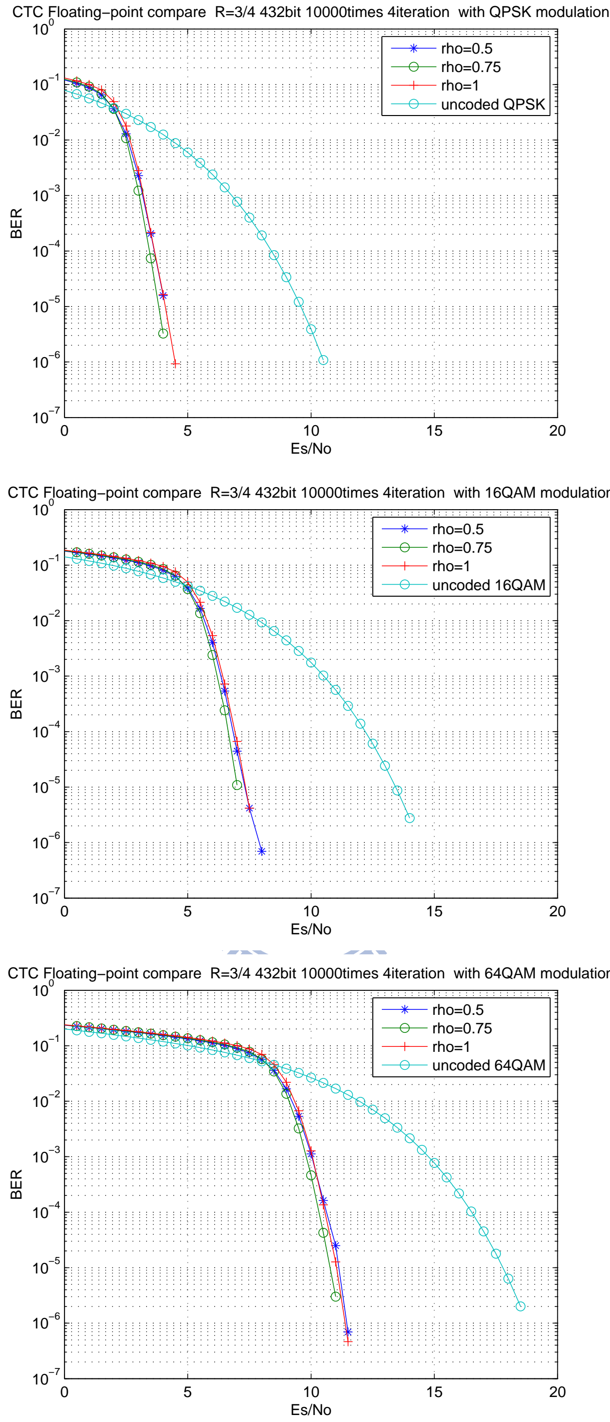


Figure 4.2: Performance of CTC at different ρ values under three different modulations with 432 information bits.

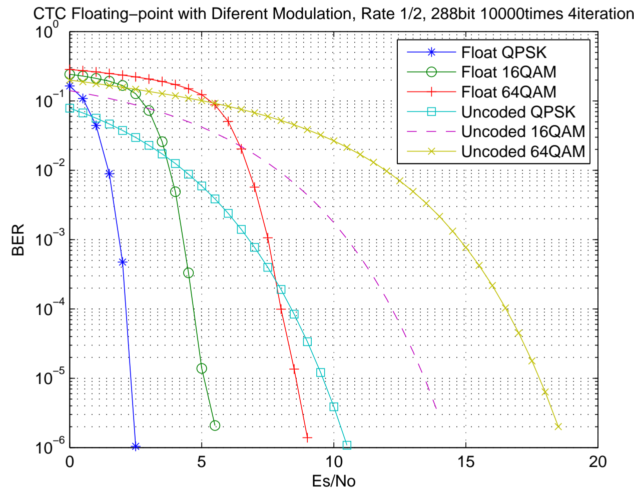


Figure 4.3: Performance of CTC at 288-bit and $\rho = 0.75$ with different modulations employing floating-point computation at 4 iterations.

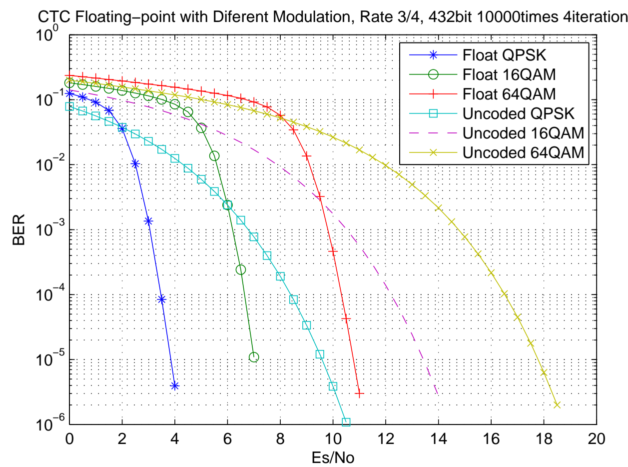


Figure 4.4: Performance of CTC at 432-bit and $\rho = 0.75$ with different modulations employing floating-point computation at 4 iterations.

also the most efficient word length for the DSP.

In Fig. 4.5 we show the fixed-point data formats in our reference CTC decoder "implementation". This design serves to illustrate the program structure but does not represent the true implementation. The hypothetical reference decoder input data format is Q4.11, which

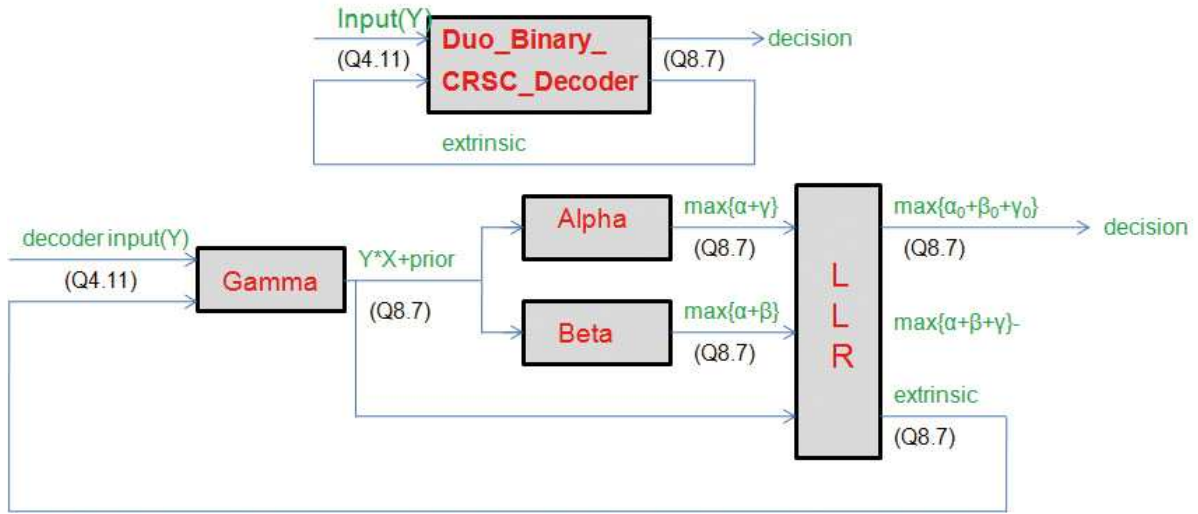


Figure 4.5: Hypothetical reference CTC decoder implementation with marking of fixed-point data format at various place.

means a 16-bit fixed-point number with one sign bit, 4 integer bits, and 11 fractional bits at right side of the dot. The alpha, beta and gamma data use the Q8.7 format. In designing our actual implementation, we first convert floating-point values to fixed-point by multiplying the original floating-point values by 1024. That means the decoder input data format is Q5.10. We find that the alpha, beta and gamma values have overflow errors. Therefore, we consider two methods, the scaling method and the clipping method to simulate result with fixed-point processing. We introduce them below.

4.2.1 Scaling Method [22]

As mentioned, we convert floating-point input values to fixed-point values by multiplying the original floating-point values by 1024 and truncating the result to 16 bits. We change the numbers of bits in the decoder input, extrinsic and gamma. The aim of scaling of the extrinsic and gamma is to try to avoid the overflows at high SNR.

In Figs. 4.6 and 4.7, we give the scaling parameters, which consist of "Scal," "Scal_E,"

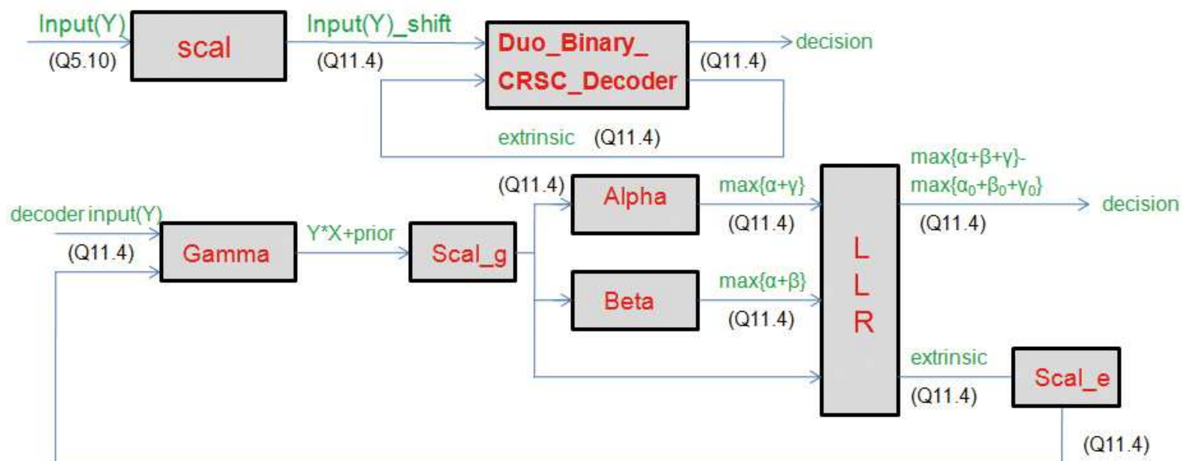
CTC Rate 1/3 CTC_FecDec.c Fixed Point Scale (S 11.4) Last Update: 20080520 Author : Uefang-Smith	
Decoder input	for (j=0; j<CTC_Input_Block; j++){ *(sys_array+j) = (*(sys_array+j)*(L_c>>1))>>(Scal+ChaReliab); *(pinfo_array+j) = (*(pinfo_array+j)*(L_c>>1))>>(Scal+ChaReliab); *(parity1_array+j) = (*(parity1_array+j)*(L_c>>1))>>(Scal+ChaReliab); *(parity2_array+j) = (*(parity2_array+j)*(L_c>>1))>>(Scal+ChaReliab);}
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 4.6: CTC fixed-point truncation parameters (modified from [22]).

and "Scal_g," standing for scale values for the decoder input, the extrinsic and the gamma, respectively. We also show how these parameters are used in the functions of our C program developed previously [22]. Note that in [22], it is no subpacket generation.

In Fig. 4.8, we compare the performance when the number of fractional bits in the decoder is between 0 to 9 (S15.0 to S6.9) for max-log-MAP decoding at rate-1/3 with 480 information bits and under three different modulations. "Scal_E" and "Scal_g" are hold at 1 and 0, respectively. 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 overflows at high SNR. Hence, we try different values of "Scal_E" and "Scal_g" to control the overflows.

In Fig. 4.9, we show the performance with different values of "Scal_E" and "Scal_g." We



Note: All sacles indicate right shifts, assumed 0 here.

Figure 4.7: Illustration of fixed-point data formats with the scaling method, where Q11.4 may be replaced by other setting (such as Q9.6 or Q14.1) depending on code rate and operating condition.

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, for 16QAM with S11.4, and for 64QAM with S11.4.

In Figs. 4.10 and 4.11, we show the performance for code rate = 1/2, 288 information bits and code rate = 3/4, 432 information bits under three modulations. We only compare the performance when the number of fractional bits in decoder is 1 and 4 (i.e., S14.1 and S11.4) for max-log-MAP decoding. In these figures, Scal_E = 1 and Scal_g = 0. We can see that S11.4 has better performance than S14.1, but has overflows at high SNR. Therefore, we use S14.1 in our implementation. The coding gains of QPSK, 16QAM, and 64QAM at BER=10^{-5} for code rate 1/2 are 6.51, 7.29, and 8.59 dB, respectively, and that for code rate 3/4 are 5.41, 5.89, and 6.62 dB, respectively.

Tables 4.1 and 4.2 show the coding gains obtained with floating-point computation and that with fixed-point for scaling method computation. We can see that the differences in

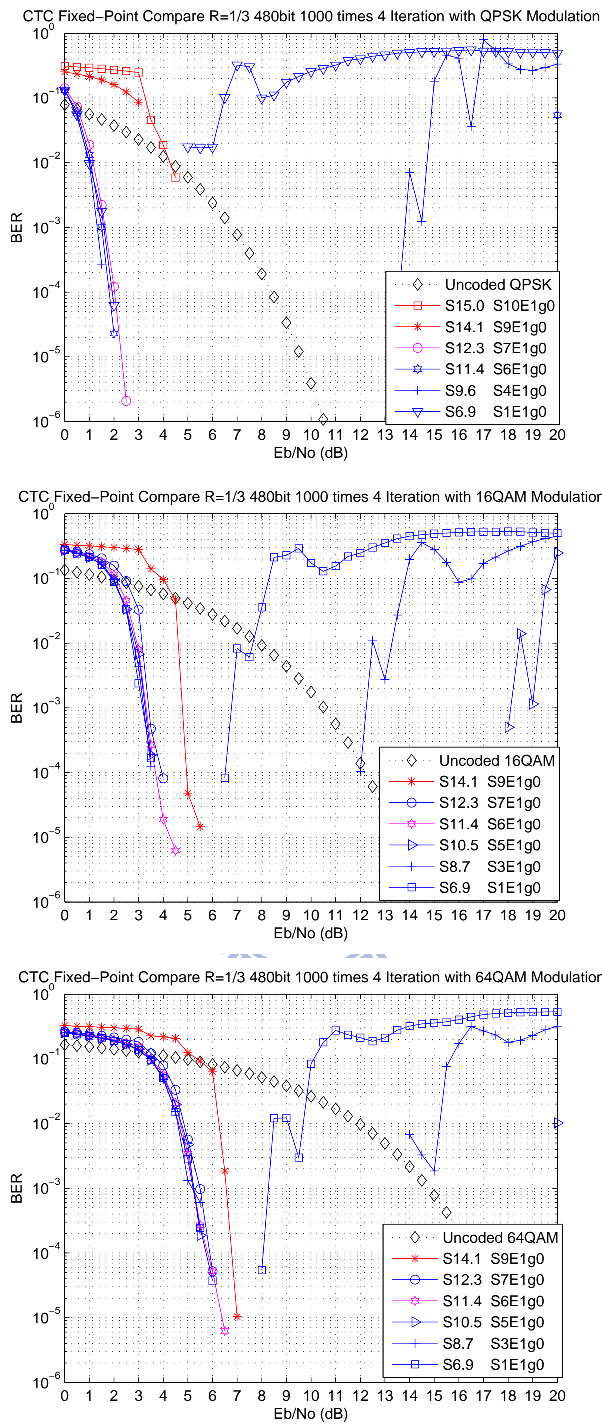
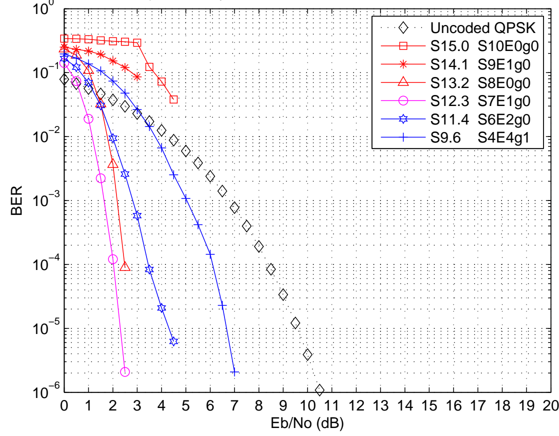
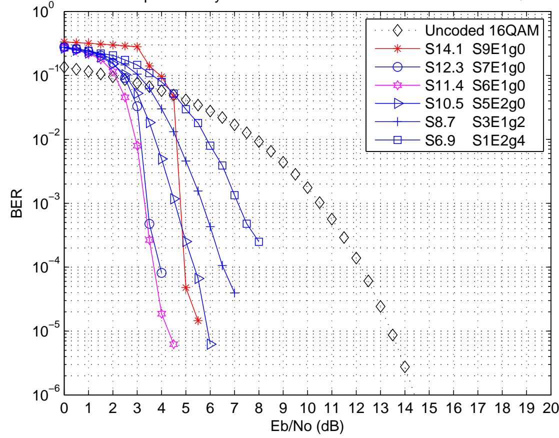


Figure 4.8: CTC decoding at different bit numbers with different modulations.





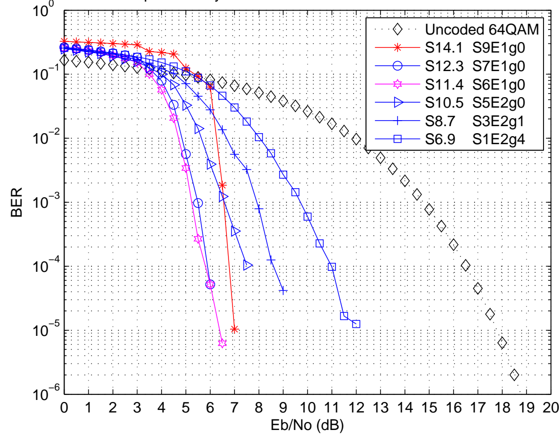


Figure 4.9: Performance with scaling of various quantities in CTC decoding to avoid overflow at high SNR.

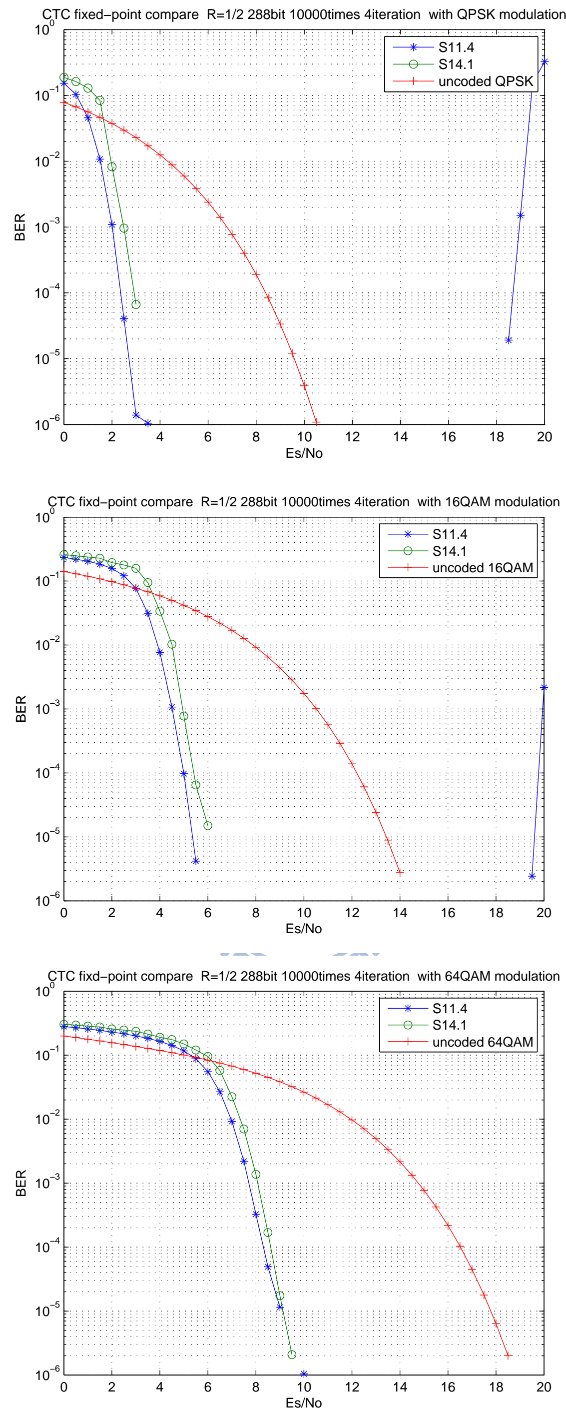


Figure 4.10: Performance of CTC with different scale factors under three modulations with 288 information bits.

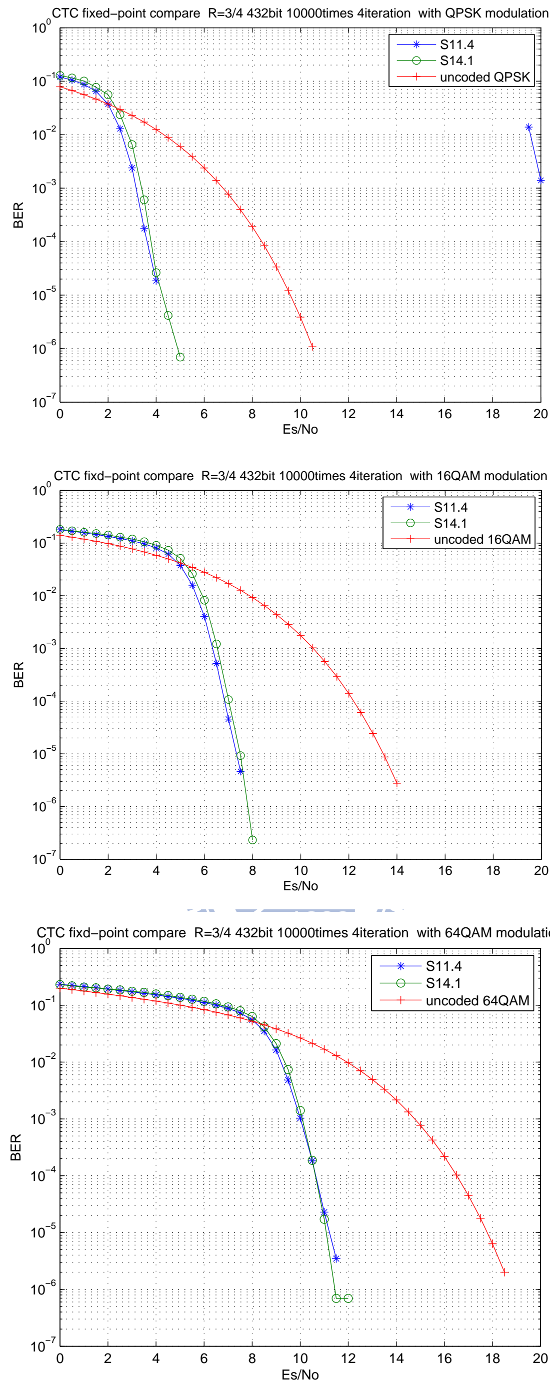


Figure 4.11: Performance of CTC with different scale factors under three modulations with 432 information bits.

Table 4.1: Coding Gain Performance of Rate-1/2 CTC in AWGN at BER = 10^{-5} with Floating-Point and Fixed-Point with Scaling Method Computation

Modulation	Floating-Point Coding Gain (dB)	Fixed-Point Coding Gain (dB)
QPSK	7.05	6.51
16QAM	8.35	7.29
64QAM	9.19	8.59

Table 4.2: Coding Gain Performance of Rate-3/4 CTC in AWGN at BER = 10^{-5} with Floating-Point and Fixed-Point with Scaling Method Computation

Modulation	Floating-Point Coding Gain (dB)	Fixed-Point Coding Gain (dB)
QPSK	5.59	5.41
16QAM	6.38	5.89
64QAM	6.92	6.62

coding gains between floating-point and fixed-point computations are 0.5 to 1 dB.

4.2.2 Clipping Method [19], [20]

In this method, we also convert floating-point input values to fixed-point values by multiplying the original floating-point value by a factor and clipping the result to 16 bits. From Fig. 4.5 we can see the integer part is 8-bit at least. For this reason, we let the decoder input data format be Q8.7 to avoid overflow for alpha, beta and gamma computations. The input multiplication factor is thus 128. Besides, we clip the decoder input and the extrinsic to avoid overflow at high SNR. The data format used is shown in Fig. 4.12.

In Figs. 4.13 and 4.14, we compare the performance when the decoder input ranges are -

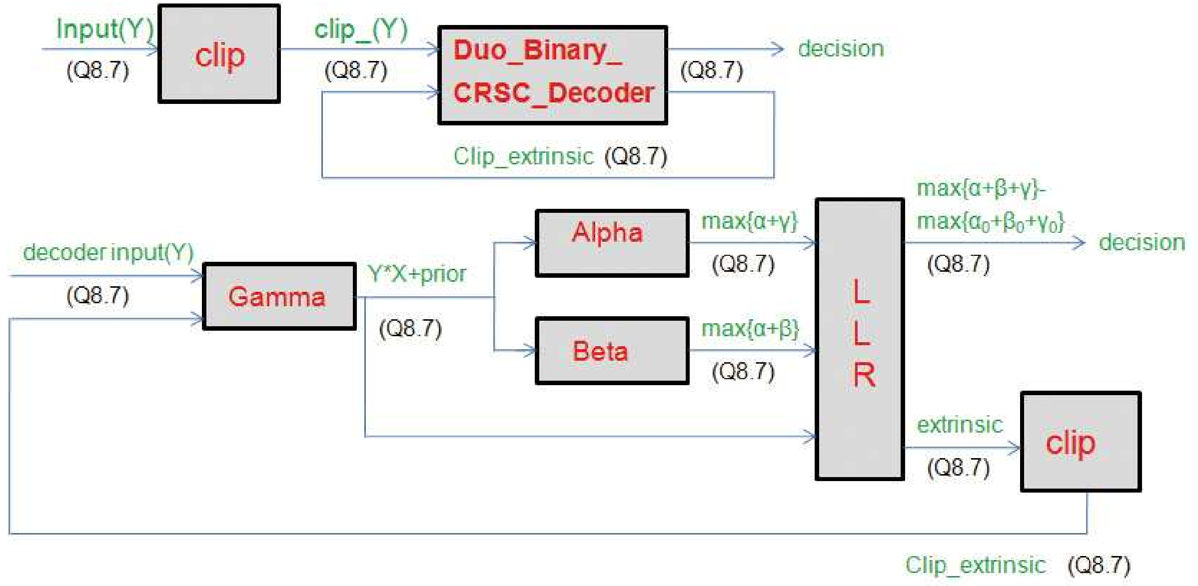


Figure 4.12: Fixed-point data format with the clipping method.

4–3.9921875 (Din4) and -8–7.9921875 (Din8), extrinsics information ranges are -8–7.9921875 (Ex8), -16–15.9921875 (Ex16), and -32–31.9921875 (Ex32) for max-log-MAP decoding at rate 1/2 with 288 information bits and rate 3/4 with 432 information bits under different modulations. If any of the ranges are exceeded, the corresponding value is clipped (i.e., saturated) to the boundary of the range. In Figs. 4.13 and 4.14, we have used Din4-Ex8, Din4-Ex16, Din8-Ex16, and Din8-Ex32. The BER curves are almost the same for QPSK, 16QAM, and 64QAM. We can see that the performance of the clipping method for rate 1/2 with 288 information bits is better than that of the scaling method, and for rate 3/4 with 432 information bits the performance is close.

In Figs. 4.15 and 4.16, we show the performance of CTC decoding with fixed-point computation under the clipping method vs. floating-point computation. The BER curves of fixed-point results are close to that of floating-point. Tables 4.3 and 4.4 show the coding gains with floating-point computation and fixed-point computations under the clipping method (Din4-Ex16) and the scaling method (S14.1) at BER=10^{-4}.

Table 4.3: Coding Gain at Rate 1/2 with 288 Information Bits CTC in AWGN at BER = 10^{-4} with Floating-Point Computation and Fixed-Point Computations with Scaling Method and Clipping Method

Modulation	Floating-Point Coding Gain (dB)	Fixed-Point (Scaling) Coding Gain (dB)	Fixed-Point (clipping) Coding Gain (dB)
QPSK	6.0299	5.4438	5.9302
16QAM	7.3522	6.7752	7.2955
64QAM	8.5274	7.800	8.1233

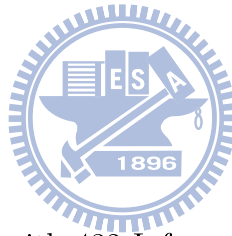
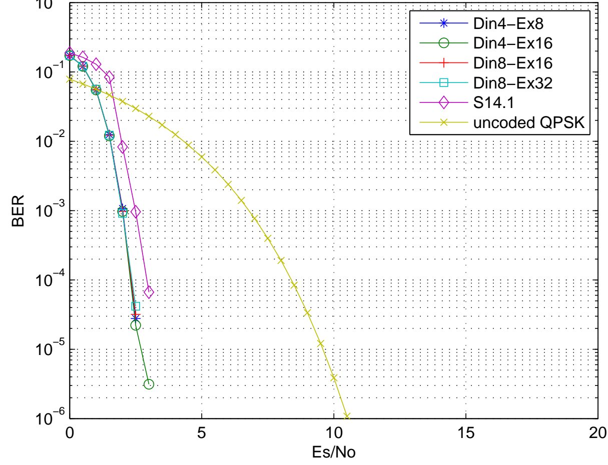
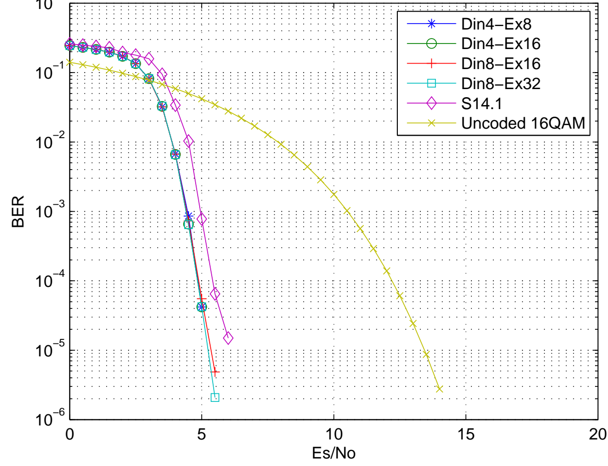


Table 4.4: Coding Gain at Rate 3/4 with 432 Information Bits CTC in AWGN at BER = 10^{-4} with Floating-Point Computation and Fixed-Point Computations with Scaling Method and Clipping Method Computation

Modulation	Floating-Point Coding Gain (dB)	Fixed-Point (Scaling) Coding Gain (dB)	Fixed-Point (Clipping) Coding Gain (dB)
QPSK	4.9313	4.4888	4.7544
16QAM	5.4409	5.2105	5.3609
64QAM	6.0954	5.7722	6.0019





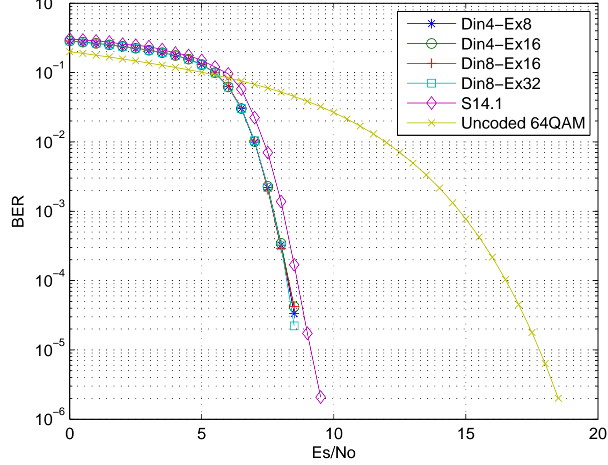
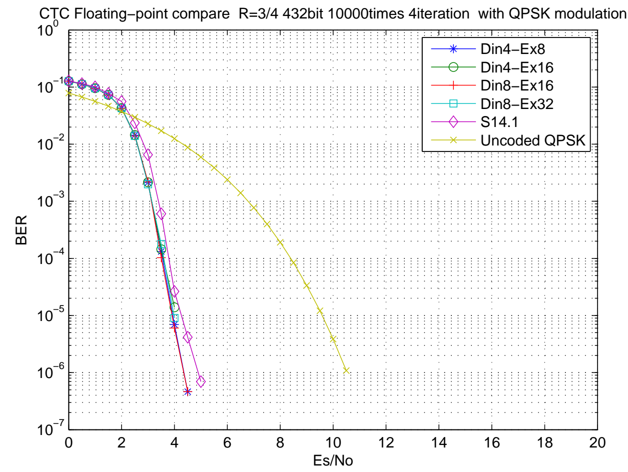
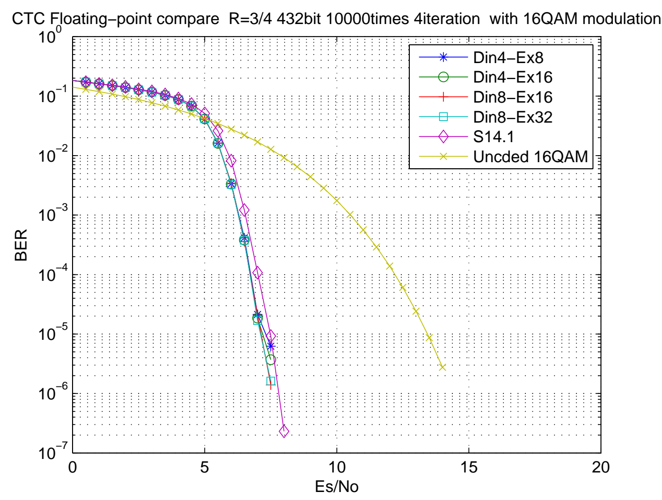


Figure 4.13: Performance of CTC at different clipping ranges under three modulations with 288 information bits.





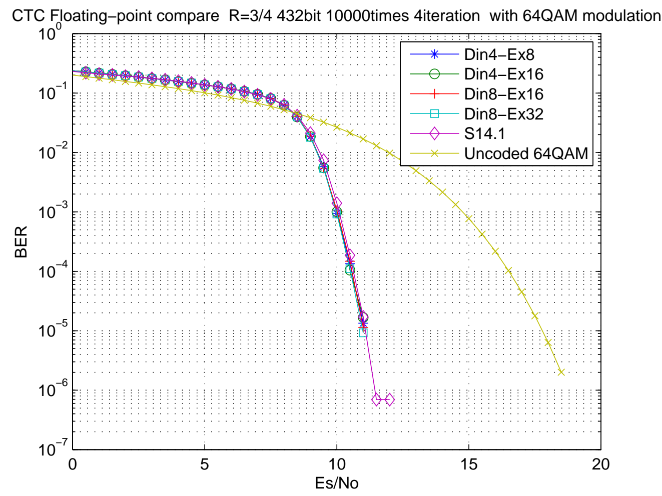
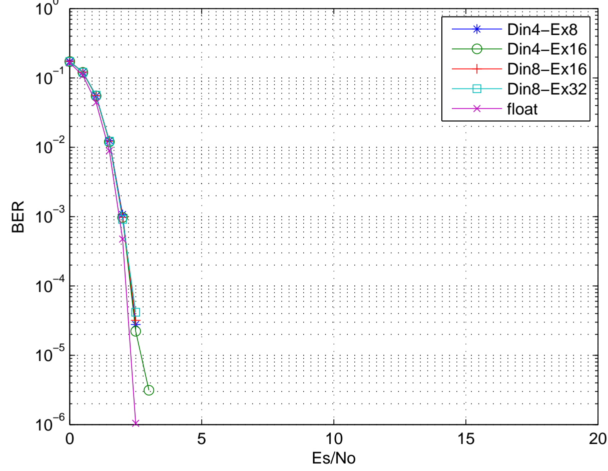
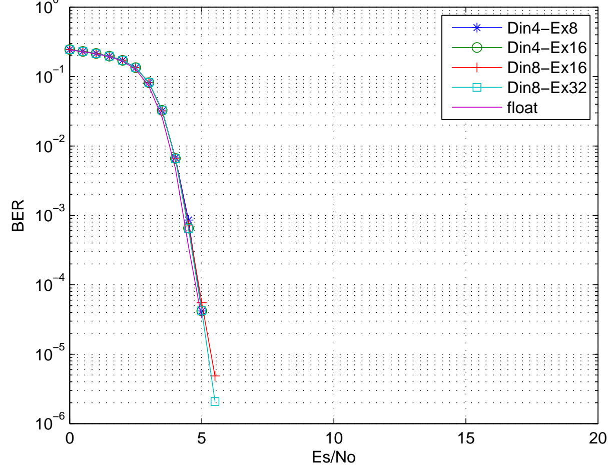


Figure 4.14: Performance of CTC at different clipping ranges under three modulations with 432 information bits.





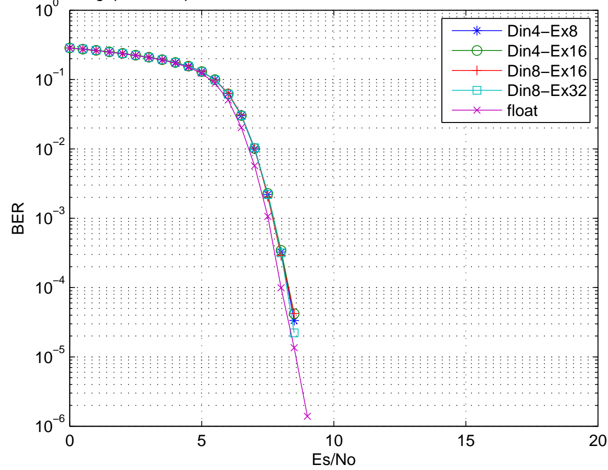
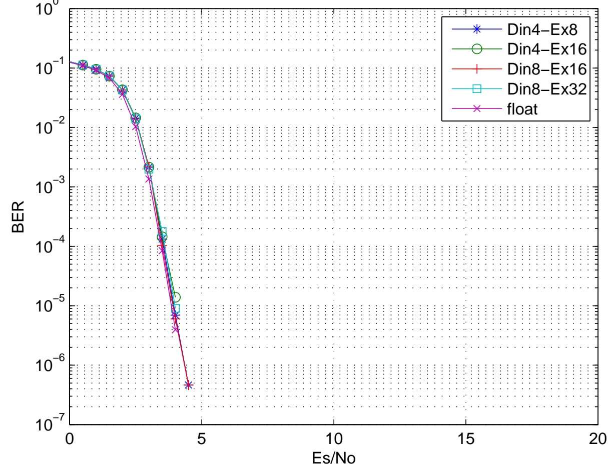
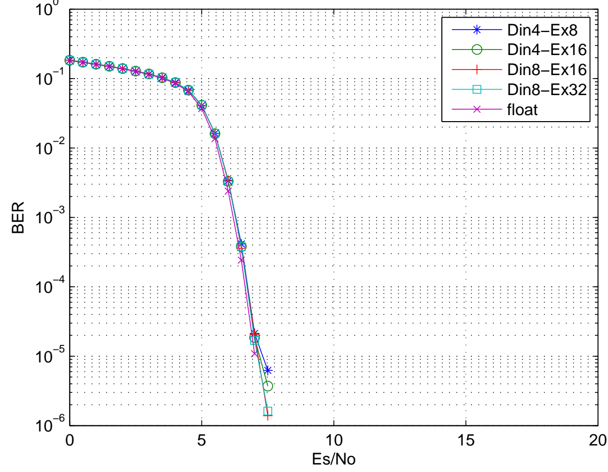


Figure 4.15: Performance of rate 1/2 CTC with 288 information bits with floating-point decoding vs. fixed-point under clipping method.





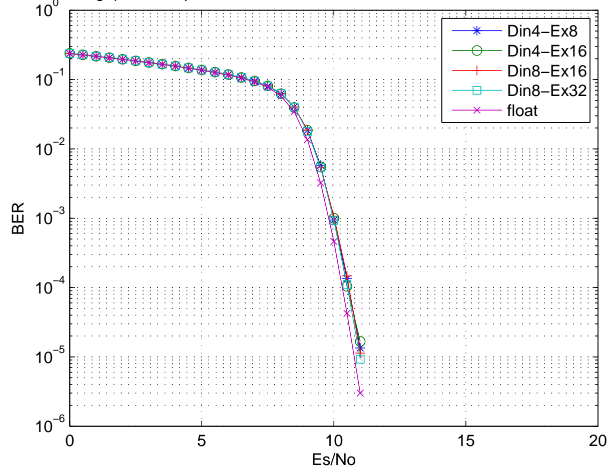


Figure 4.16: Performance of rate 3/4 CTC with 432 information bits with floating-point decoding vs. fixed-point under clipping method.

Chapter 5

Speeding Up of DSP Implementation

In this chapter, we consider how to speed up the DSP implementation of the CTC decoder, especially ways of employing intrinsic functions to reduce cycle counts. In thus, we change the ordering of alpha and beta to achieve parallelism. In the following we discuss how to use the intrinsics and arrange the ordering of alpha and beta order. Then we compare the parallelism of the fixed-point C programs with and without using the intrinsics.

5.1 Speed of DSP [17]

According to [17], we can realize substantial gains in the performance of the C code by refining it in the following areas:

- Using intrinsics to replace complicated C/C++ code.
- Using word access to operate on 16-bit stored in the high and low parts of a 32-bit register.
- Using double access to operate on 32-bit data stored in a 64-bit register pair.

In order to maximize data throughput on DSP, it is often desirable to use a single load or store instruction to access multiple data values consecutively located in memory. All C6000

Table 5.1: TMS320C64X Compiler Intrinsics [17].

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_dotp2**(int *src1*, int *src2*); double **_ldotp2**(int *src1*, int *src2*);	DOTP2 LDOTP2	The product of the signed lower 16-bit values of src1 and src2 is added to the product of the signed upper 16-bit values of src1 and src2. The _lo and _hi intrinsics are needed to access each half of the 64-bit integer result.
int **_max2** (int *src1*, int *src2*); int **_min2**(int *src1*, int *src2*); uint **_maxu4**(uint *src1*, uint *src2*); uint **_minu4**(uint *src1*, uint *src2*);	MAX2 MIN2 MAX4 MINU4	Places the larger/smaller of each pair of values in the corresponding position in the return value. Values can be 16-bit signed or 8-bit unsigned.
uint **_pack2** (uint *src1*, uint *src2*); uint **_packh2** (uint *src1*, uint *src2*);	PACK2 PACKH2	The lower/upper halfwords of src1 and src2 are placed in the return value.
uint **_packh4** (uint *src1*, uint *src2*); uint **_packl4** (uint *src1*, uint *src2*);	PACKH4 PACKL4	Packs alternate bytes into return value. Can pack high or low bytes.
uint **_packhl2** (uint *src1*, uint *src2*); uint **_packlh2** (uint *src1*, uint *src2*);	PACKHL2 PACKLH2	The upper/lower halfword of src1 is placed in the upper halfword the return value. The lower/upper halfword of src2 is placed in the lower halfword the return value.
uint **_rotl** (uint *src1*, uint *src2*);	ROTL	Rotates src2 to the left by the amount in src1
int **_sadd2**(int *src1*, int *src2*); int **_saddus2**(uint *src1*, int *src2*);	SADD2 SADDUS2	Performs saturated addition between pairs of 16-bit values in src1 and src2. Values for src1 can be signed or unsigned.
uint & **_amem4**(void **ptr*);	LDW STW	Allows aligned loads and stores of 4 bytes to memory†

devices have instructions with corresponding intrinsics. That operate on 16-bit data stored in the high and low parts of a 32-bit register. When operating on a stream of 16-bit data, we can use word (32-bit) accesses to read two 16-bit at a time.

Table 5.1 displays some intrinsic functions used in our DSP implementation, which include $_amem4()$, $_dotp2()$, $_max2()$, $_packXX2()$ group, $_sadd2()$, and $_rotl()$. We show how to use these intrinsic functions in the following:

- $_amem4()$: Fig. 5.1 show that this intrinsic function tells the compiler that the following access is a 4-byte (or word) aligned access address of an unsigned (or signed) int.
- $_dotp2()$: Fig. 5.2 illustrates how the $_dotp2()$ intrinsic operates. We can see that two 32-bit registers, which are divided into two 16-bit register high parts (hi) and low parts (lo), respectively. It multiplies corresponding parts in the two words separately and then sum the two products together. Therefore, one $_dotp2()$ intrinsic function can complete two multiplications and one addition.

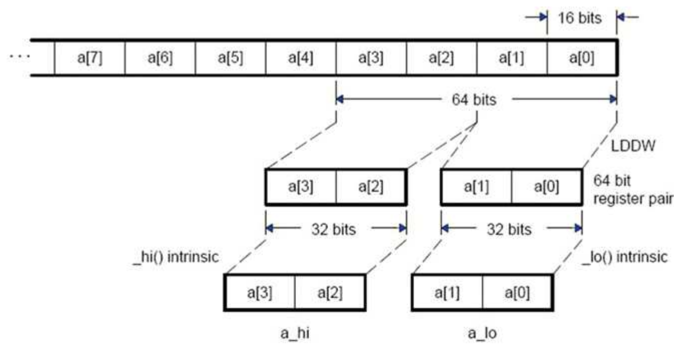


Figure 5.1: Graphical representation of the $_amem4()$ and the $_max2()$ intrinsics [17].

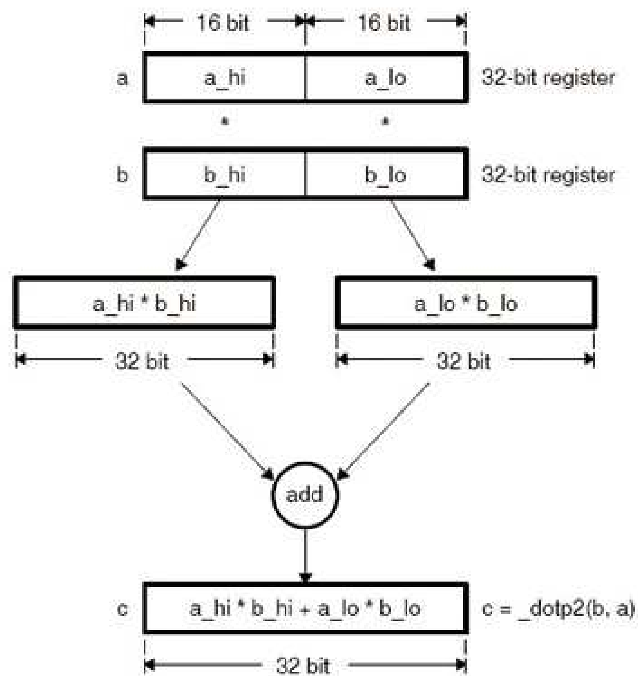


Figure 5.2: Graphical representation of the $_dotp2()$ intrinsic [17].

- $_max2()$: The $_max2()$ intrinsic function is compares two pairs of numbers and selects the larger in each pair. For example, in Fig. 5.1, a[0] is compared with a[2] and a[1] with a[3] at the same time.

- $_packXX2()$: The $_packXX2()$ group of intrinsics works by extracting selected half-

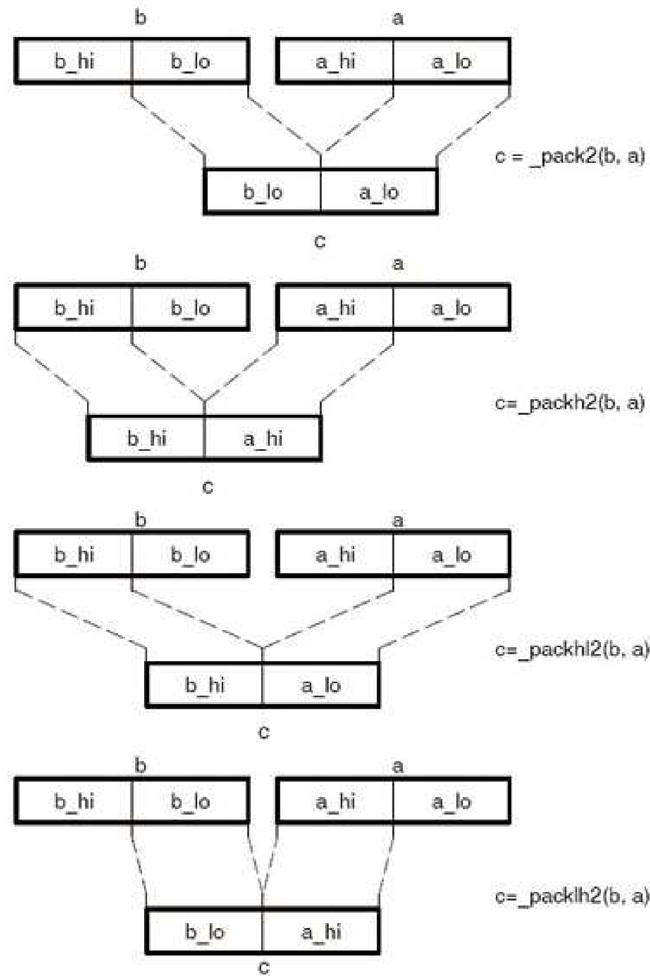


Figure 5.3: Graphical representation of *_packXX2()* intrinsics[17].

words from two 32-bit registers and returning the result packed into 32-bit word. This is primarily useful for manipulating packed 16-bit data, although they may be used for manipulating pairs of 8-bit quantities. Fig. 5.3 illustrates the four *_packXX2()* intrinsics, *_pack2()*, *_packlh2()*, *_packhl2()*, and *_packh2()*. (The l and the h in the names refer to which half of each 32-bit input is being copied to the output.)

- *_sadd2()*: The *_sadd2()* intrinsic provides saturating pack and adds for corresponding packed elements in two different words, producing two packed sums.

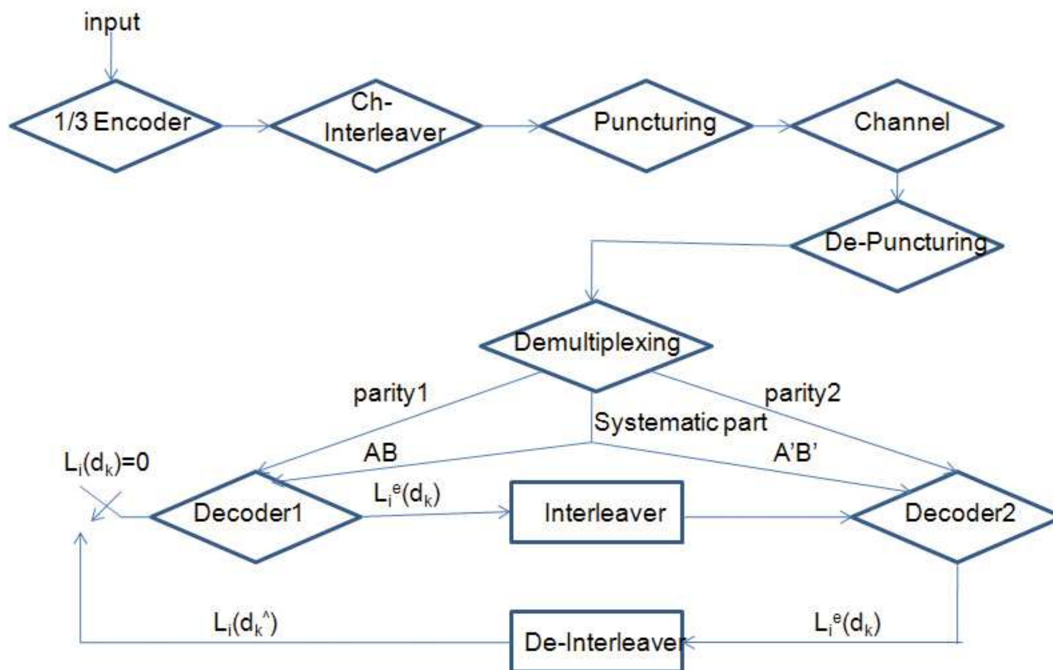


Figure 5.4: Overall encoder and decoder architecture.

- *_rotl*(): The *_rotl*() intrinsic function rotates the 32-bit value in src2 to the left by the amount in src1.

5.2 Original State Order [22]

In Fig. 5.4, we show the overall encoder and decoder architecture. Note that Demultiplexing is including de-channel interleaver and de-CTC interleaver. In Table 5.2, we show the every block cycles with QPSK modulation for 480 information bits, rate 1/2 coding in one iteration. We can see that block of Ch-Interleaver (channel interleaver), Demultiplexing and Decoder (*Duo_binary_CRSC_decoder*) spend much more cycles than others. For Ch-Interleaver and Demultiplexing, we compute interleave position in advance and then build the table to stored. Table 5.3 shows the improvement in speed of Ch-Interleaver and Demultiplexing.

Table 5.2: Overall Encoder and Decoder Block Cycles

Block	Times Called	CPU Cycles
1/3 Encoder	2	9906
Ch-Interleaver	1	54212
Puncturing	1	496
De-puncturing	1	563
Demultiplexing	1	46995
Decoder	2	122352
Interleaver	2	4598
De-Interleaver	1	2307

Table 5.3: Speed Up in Channel Interleaver

Block	Times Called	CPU Cycles	Reduction in Complexity (%)
Ch-Interleaver (Original)	1	54212	N/A
Ch-Interleaver (Improved)	1	3239	99.39
Demultiplexing (Original)	1	46995	N/A
Demultiplexing (Improved)	1	6352	86.48

For *Duo_binary_CRSC_decoder*, we can know the max-log-MAP decoding algorithm cost the most execution time. In [22], the intrinsic functions *_max2()* and *_sadd2()* are used, but the SIMD (single-instruction multiple-data) features of the DSP are not used. In Table 5.4, we show the component functions in the *Duo_binary_CRSC_decoder* function that performs the max-log MAP decoding algorithm. They consist of *gamma*, *alpha*, *beta*, *LLR* and *extrinsics* functions. We can see that the *extrinsics* function needs much fewer cycles then the other function. Therefore, we do not discuss it in the next section.

Table 5.4: Profile of *Duo_Binnary_CRSC_decoder* with QPSK Modulation for 480 Information Bits, Rate 1/2 Coding in One Iteration

Function	Times Called	CPU Cycles	Percentage (%)
gamma	1	17830	32.41
alpha	1	13028	23.68
beta	1	13225	24.04
LLR	1	8495	15.44
extrinsics	1	2433	4.42

5.3 Arrange State Order to Achieve Parallelism

In this section, we illustrate how to arrange the state order to parallelize decoder processing. We try to access two 16-bit data values consecutively located in memory and then do operation on two 32-bit words at a time. Note that we have three sets of parameters to arrange, from the BCJR algorithm for MAP decoding, which consist of the following:

- The forward state metrics α.
- The backward state metrics β.
- The branch metrics γ.

In Fig. 5.5, we show the original state order, whose sequence is from state 0 to state 7, and every state is arranged based on input order 00, 01, 10, 11. When using intrinsics to compute forward state metrics, we cannot use word (32-bit) accesses to read two 16-bit quantities at a time. Therefore, we propose an other arrangement to achieve parallelism.

First, we observe that the state order of forward metrics have some properties, In Fig. 5.5 we can see that the sets (α_0^{k-1}, α_1^{k-1}, α_6^{k-1}, α_7^{k-1}) and (α_2^{k-1}, α_3^{k-1}, α_4^{k-1}, α_5^{k-1}) of forward metrics at time $k-1$ have the sets (α_0^{k}, α_3^{k}, α_4^{k}, α_7^{k}) and (α_1^{k}, α_2^{k}, α_5^{k}, α_6^{k}), respectively,

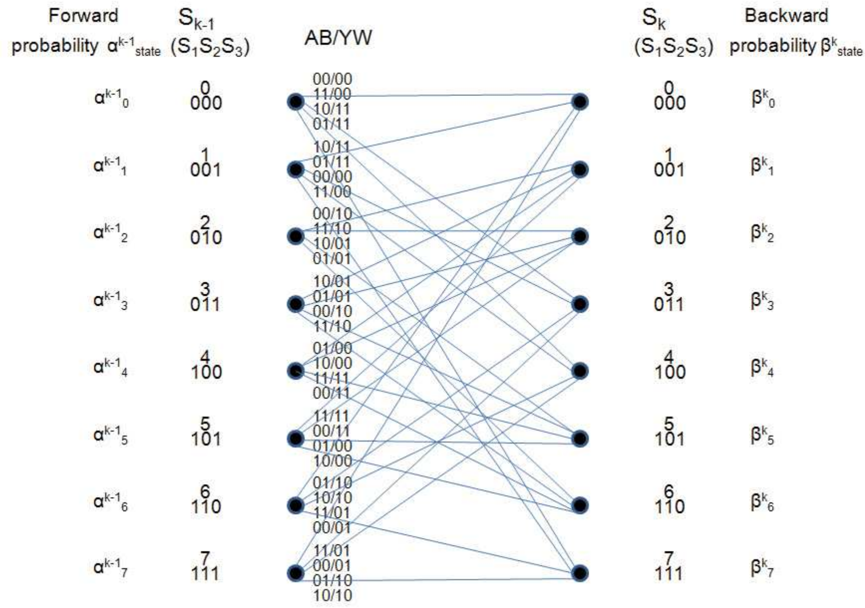


Figure 5.5: Trellis diagram, every branch in the trellis connecting at time $k-1$ to a state at time k.

as possible state at time k. Therefore, we allocate memory for them in the order of α_0^{k-1}, α_2^{k-1}, α_1^{k-1}, α_3^{k-1}, α_7^{k-1}, α_5^{k-1}, α_6^{k-1}, α_4^{k-1}, consecutively. In Fig. 5.6, we show the trellis with rearranged state order for forward metrics. Note that, after computing the forward metrics, the state order is changed to α_0^k, α_1^k, α_3^k, α_2^k, α_7^k, α_6^k, α_4^k, α_5^k, which is not the same as previous time $k-1$. In Fig. 5.7, we show how to use the intrinsics "$_packXX2()$" to put the states at the correct positions.

Second, in Fig. 5.6, we also show the arrangement of state order for backward metrics. Similarly, from Fig. 5.5 we can see that the sets (β_0^k, β_3^k, β_4^k, β_7^k) and (β_1^k, β_2^k, β_5^k, β_6^k) of backward metrics at time k have the (β_0^{k-1}, β_1^{k-1}, β_6^{k-1}, β_7^{k-1}) and (β_2^{k-1}, β_3^{k-1}, β_4^{k-1}, β_5^{k-1}), respectively, as possible state at time $k-1$. Therefore, we allocate the state order as β_0^k, β_1^k, β_3^k, β_2^k, β_7^k β_6^k, β_4^k, β_5^k. After computation of backward metrics, the state order changes to β_0^{k-1}, β_2^{k-1}, β_1^{k-1}, β_3^{k-1}, β_7^{k-1}, β_5^{k-1}, β_6^{k-1}, β_4^{k-1}. Similar to Fig. 5.7, we use the intrinsic

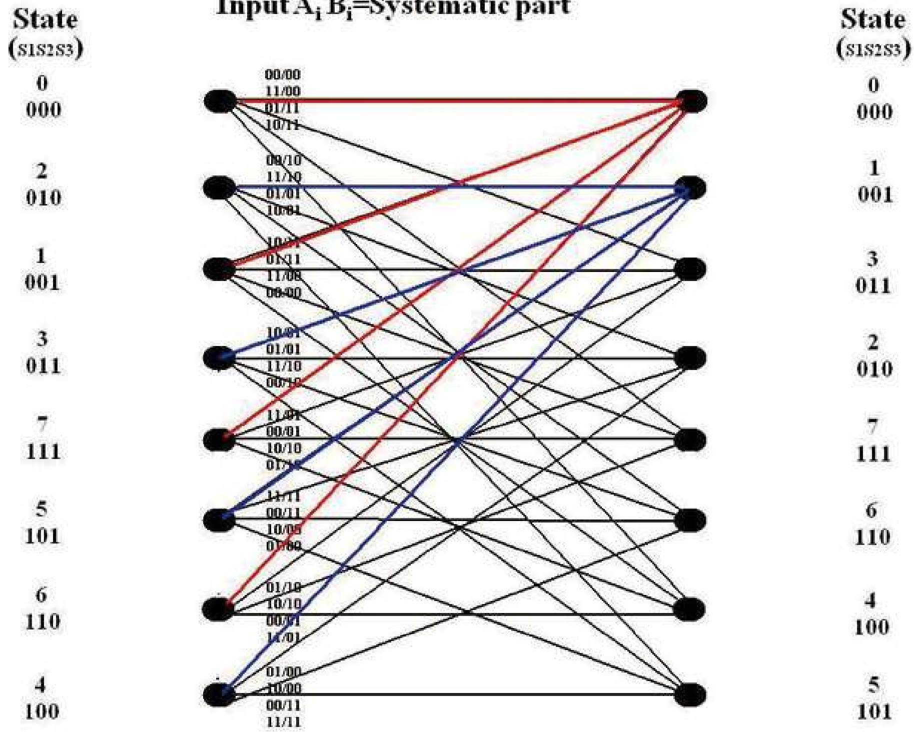


Figure 5.6: Arrangement of trellis order for forward and backward metrics.

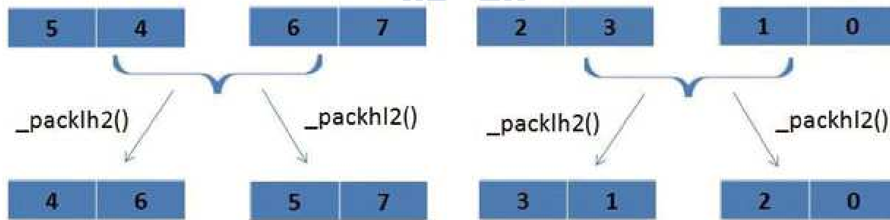


Figure 5.7: Use of the *_packXX2()* intrinsics for forward metric .

"*_packXX2*" to put the backward metrics at the correct positions. Third, for the branch

metric,

$$\begin{aligned}\Gamma_k(S_{k-1}, S_k) &= \ln[p(y_k|d_k) \cdot P(d_k)] \\ &= 0.5 \cdot L_c \cdot [y_k^{s,I} \cdot x_k^{s,I}(i) + y_k^{s,Q} \cdot x_k^{s,Q}(i) + y_k^{p,I} \cdot x_k^{p,I}(i, S_{k-1}, S_k) \\ &\quad + y_k^{p,Q} \cdot x_k^{p,Q}(i, S_{k-1}, S_k)] + \ln P(d_k) + K, \end{aligned} \tag{5.1}$$

we can use one intrinsic *_dotp2* to compute $y_k^{s,I} \cdot x_k^{s,I}(i) + y_k^{s,Q} \cdot x_k^{s,Q}(i)$, which computes the branch metrics from the received systematic and parity bits. This needs to be done only for the first iteration, but not for later iterations. In Fig. 5.8, we show the improved C code of the alpha function for computing the forward metrics, and in Figs. 5.9 to 5.13, we show the corresponding assembly code. The software pipeline information is shown in Fig. 5.14. Table 5.5 shows the improvement in speed of *gamma*, *alpha*, *beta*, *LLR* and *Gamma_Table* functions with QPSK modulation for 480 information bits, 1/2 rate coding in one iteration. They account for 84.9, 51.67, 52.38 and 8.56%, respectively, of the complexity of the improved *Duo_binary_CRSC_Decoder*. Due to the iteration is one, therefore, we can sum the cycles of *gamma* and *Gamma_Table* functions of the improved code to compare with the *gamma* function of the original code. We see that there is 34.7% reduction in complexity. For the *LLR* function we only improve 8.56% in speed, we conjecture that this has to do with the amount of memory. The original code only uses three memory spaces (01, 10, 11) in every time k. But the improved code uses four memory spaces (00, 01, 10, 11) in every time k to accessed two 16-bit words at a time.

Table 5.6 compares the original cycles to the improved cycles for the *Duo_Binary_CRSC_Deoder* function and the *CTC_Decoder* function.

Table 5.5: Profile of Improve *Duo_Binnary_CRSC_Decoder* with QPSK Modulation for 480 Information Bits, Rate 1/2 Coding in One Iteration

Function	Original (Cycles)	Improved (Cycles)	Reduction in Complexity (%)
gamma	17830	2692	84.9
alpha	13028	6296	51.67
beta	13225	6297	52.38
LLR	8495	7767	8.56
Gamma_Table	*N/A*	8947	*N/A*

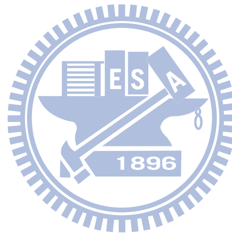


Table 5.6: Speed Up in Decoding of One Data Block with QPSK Modulation for One Iteration

Function	Times Called	Cycles	Reduction in Complexity (%)
Duo_Binnary_CRSC_Decoder (Original)	2	122352	*N/A*
Duo_Binnary_CRSC_Decoder (Improved)	2	60908	50.21
CTC_Decoder (Original)	2	209900	*N/A*
CTC_Decoder (Improved)	2	126918	39.53

```
void ALPHA(short *alpha1,short *gamma1,int length,int state)
{

short i,m,k,n,u,s;
int temp_alpha[4]={0};
int Normal_alpha,rotl_alpha,max_alpha;

for(i=0;i<Length;i++)
{

 m=i<<5;
  k=(i+1)<<3;
  n=i<<3;

/*s:2,s:0*/_amem4(&alpha1[k])  =_max2(_max2(_sadd2(_amem4(&alpha1[n]   ),_amem4(&gamma1[m]   ))
                                          ,_sadd2(_amem4(&alpha1[2+n] ),_amem4(&gamma1[18+m])))
                                    ,_max2(_sadd2(_amem4(&alpha1[4+n] ),_amem4(&gamma1[28+m]))
                                          ,_sadd2(_amem4(&alpha1[6+n ]),_amem4(&gamma1[14+m]))));/*s:1,s:0*/
/*s:3,s:1*/_amem4(&alpha1[2+k])=_max2(_max2(_sadd2(_amem4(&alpha1[n]   ),_amem4(&gamma1[24+m]))
                                          ,_sadd2(_amem4(&alpha1[2+n] ),_amem4(&gamma1[10+m])))
                                    ,_max2(_sadd2(_amem4(&alpha1[4+n] ),_amem4(&gamma1[4+m] ))
                                          ,_sadd2(_amem4(&alpha1[6+n] ),_amem4(&gamma1[22+m]))));/*s:2,s:3*/
/*s:5,s:7*/_amem4(&alpha1[4+k])=_max2(_max2(_sadd2(_amem4(&alpha1[n]   ),_amem4(&gamma1[8+m] ))
                                          ,_sadd2(_amem4(&alpha1[2+n] ),_amem4(&gamma1[26+m])))
                                    ,_max2(_sadd2(_amem4(&alpha1[4+n] ),_amem4(&gamma1[20+m]))
                                          ,_sadd2(_amem4(&alpha1[6+n] ),_amem4(&gamma1[6+m] ))));/*s:6,s:7*/
/*s:4,s:6*/_amem4(&alpha1[6+k])=_max2(_max2(_sadd2(_amem4(&alpha1[n]   ),_amem4(&gamma1[16+m]))
                                          ,_sadd2(_amem4(&alpha1[2+n] ),_amem4(&gamma1[2+m] )))
                                    ,_max2(_sadd2(_amem4(&alpha1[4+n] ),_amem4(&gamma1[12+m]))
                                          ,_sadd2(_amem4(&alpha1[6+n ]),_amem4(&gamma1[30+m]))));/*s:5,s:4*/


temp_alpha[0]=_packhl2(_amem4(&alpha1[2+k]),_amem4(&alpha1[k]));
temp_alpha[1]=_packlh2(_amem4(&alpha1[2+k]),_amem4(&alpha1[k]));
temp_alpha[2]=_packhl2(_amem4(&alpha1[6+k]),_amem4(&alpha1[k+4]));
temp_alpha[3]=_packlh2(_amem4(&alpha1[6+k]),_amem4(&alpha1[k+4]));

Normal_alpha=_max2(_max2(temp_alpha[0],temp_alpha[1]),_max2(temp_alpha[2],temp_alpha[3]));
rotl_alpha=_rotl(Normal_alpha,16);
max_alpha=_max2(Normal_alpha,rotl_alpha);

for(u=0;u<4;u++)
{  s=u<<1;
      _amem4(&alpha1[s+k])=_sub2(temp_alpha[u],(max_alpha));
}

}
```

Figure 5.8: Improved C code for the alpha() function.

5.4 Comparison of Speed

In this section we investigate the processing rates of the original code and the improved code. Besides, we compare the numbers of additions, multiplications and the intrinsics between the original code and the improved code.

5.4.1 Comparison of Original and Improved Codes in Additions, Multiplications and Intrinsic Functions

We evaluate roughly the numbers of intrinsic functions for the CTC decoder in the original code and the improved code. The equations for a priori probability computation in

```
void ALPHA(short *alpha1,short *gamma1,int length,int state)
{
00004DD0          ALPHA:
00004DD0 07BF09C2               SUB.D2       SP,0x18,SP
00004DD4 0A100FD8               OR.L1        0,A4,A20

short i,m,k,n,u,s;
int temp_alpha[4]={0};
00004DD8 01CE5828               MVK.S1       0xffff9cb0,A3
00004DDC 018001E8               MVKH.S1      0x30000,A3
00004DE0 040C0364               LDDW.D1T1    *+A3[0x0],A9:A8
00004DE4 030C2364               LDDW.D1T1    *+A3[0x1],A7:A6
00004DE8 00002000               NOP          2
00004DEC 02BD005A               ADD.L2       8,SP,B5
00004DF0 041403C4               STDW.D2T1    A9:A8,*+B5[0x0]
00004DF4 031423C4               STDW.D2T1    A7:A6,*+B5[0x1]
int Normal_alpha,rotl_alpha,max_alpha;

for(i=0;i<Length;i++)
00004DF8 0C8403E3               MVC.S2       CSR,B25
00004DFC 09900FDA  ||           OR.L2        0,B4,B19
00004E00 0000782B               MVK.S2       0x00f0,B0
00004E04 0280A35B  ||           MVK.L2       0,B5
00004E08 0180A359  ||           MVK.L1       0,A3
00004E0C 0267C9F2  ||           AND.D2       -2,B25,B4
00004E10 009003A3               MVC.S2       B4,CSR
00004E14 098F0059  ||           SUB.L1       A3,8,A19
00004E18 0C17E05B  ||           SUB.L2       B5,1,B24
00004E1C 09000029  ||           MVK.S1       0x0000,A18
00004E20 00002040  ||           MVK.D1       1,A0
00004E24          L193:
00004E24 D41823E7      [!A0]    LDDW.D2T2    *+B6[0x1],B9:B8
00004E28 D3202364  ||  [!A0]    LDDW.D1T1    *+A8[0x1],A7:A6
00004E2C DB1863E6      [!A0]    LDDW.D2T2    *+B6[0x3],B23:B22
00004E30 DA1843E6      [!A0]    LDDW.D2T2    *+B6[0x2],B21:B20
00004E34 D21982E6      [!A0]    LDW.D2T2     *+B6[0xC],B4
00004E38 044A1049               EXT.S1       A18,16,16,A8
00004E3C D31942E6  ||  [!A0]    LDW.D2T2     *+B6[0xA],B6
00004E40 DD1C03E7      [!A0]    LDDW.D2T2    *+B7[0x0],B27:B26
00004E44 0B10BC30  ||           SADD2.S1X    A5,B4,A22
00004E48 09101FDB               OR.L2X       0,A4,B18
00004E4C D81883E5  ||  [!A0]    LDDW.D2T1    *+B6[0x4],A17:A16
00004E50 04DCFC30  ||           SADD2.S1X    A7,B23,A9
00004E54 D31C02E7      [!A0]    LDW.D2T2     *+B7[0x0],B6
00004E58 039D3C33  ||           SADD2.S2X    B9,A7,B7
```

Figure 5.9: Assembly code of the alpha() function (1/5).

```
00004E5C 0254BC30  ||          SADD2.S1X     A5,B21,A4
00004E60 D29922E7      [!A0]   LDW.D2T2      *+B6[0x9],B5
00004E64 02124C33  ||          SADD2.S2      B18,B4,B4
00004E68 01ADDC30  ||          SADD2.S1X     A6,B8,A3
00004E6C 04424C33              SADD2.S2      B18,B16,B8
00004E70 0210985B  ||          MAX2.L2X      B4,A4,B4
00004E74 0A98DC30  ||          SADD2.S1X     A6,B6,A21
00004E78 081F7C33              SADD2.S2X     B27,A7,B16
00004E7C 0214FC30  ||          SADD2.S1X     A7,B5,A4
00004E80 02D24C33              SADD2.S2      B18,B20,B5
00004E84 02106859  ||          MAX2.L1       A3,A4,A4
00004E88 01CA1C31  ||          SADD2.S1X     A16,B18,A3
00004E8C 03D4F85A  ||          MAX2.L2X      B7,A21,B7
00004E90 03D8DC31              SADD2.S1X     A6,B22,A7
00004E94 04D8B85A  ||          MAX2.L2X      B5,A22,B9
00004E98 0344BC31              SADD2.S1X     A5,B17,A6
00004E9C 0498DC33  ||          SADD2.S2X     B6,A6,B9
00004EA0 031D285A  ||          MAX2.L2       B9,B7,B6
00004EA4 0194BC31              SADD2.S1X     A5,B5,A3
00004EA8 02986859  ||          MAX2.L1       A3,A6,A5
00004EAC 029E185A  ||          MAX2.L2X      B16,A7,B5
00004EB0 01D11A41              ADDAH.D1      A20,A8,A3
00004EB4 03A07858  ||          MAX2.L1X      A3,B8,A7
00004EB8 030D01A1              ADD.S1        8,A3,A6
00004EBC 02909859  ||          MAX2.L1X      A4,B4,A5
00004EC0 0394B85A  ||          MAX2.L2X      B5,A5,B7
00004EC4 D3180347      [!A0]   STDW.D1T2     B7:B6,*+A6[0x0]
00004EC8 02253859  ||          MAX2.L1X      A9,B9,A4
00004ECC 0218E37B  ||          PACKLH2.L2    B7,B6,B4
00004ED0 0298E222  ||          PACKHL2.S2    B7,B6,B5
00004ED4 0210E859              MAX2.L1       A7,A4,A4
00004ED8 0310A85A  ||          MAX2.L2       B5,B4,B6
00004EDC D20C0345      [!A0]   STDW.D1T1     A5:A4,*+A3[0x0]
00004EE0 0410A379  ||          PACKLH2.L1    A5,A4,A8
00004EE4 0290A220  ||          PACKHL2.S1    A5,A4,A5
00004EE8 081406A1              OR.S1         0,A5,A16
00004EEC 08A008F1  ||          OR.D1         0,A8,A17
00004EF0 0220A859  ||          MAX2.L1       A5,A8,A4
00004EF4 02900FDB  ||          OR.L2         0,B4,B5
00004EF8 021406A2  ||          OR.S2         0,B5,B4
00004EFC 04989859              MAX2.L1X      A4,B6,A9
00004F00 D23C43C6  ||  [!A0]   STDW.D2T2     B5:B4,*+SP[0x2]
00004F04 09CD0059              ADD.L1        8,A19,A19
00004F08 0C60205B  ||          ADD.L2        1,B24,B24
00004F0C D83C23C5  ||  [!A0]   STDW.D2T1     A17:A16,*+SP[0x1]
```

Figure 5.10: Assembly code of the alpha() function (2/5).

```
00004F10 03A607B0  ||          ROTL.M1     A9,0x10,A7
00004F14 2003E05B     [ B0]    SUB.L2      B0,1,B0
00004F18 0362B04B  ||          EXT.S2      B24,21,16,B6
00004F1C 024E1048  ||          EXT.S1      A19,16,16,A4
00004F20 039D2859              MAX2.L1     A9,A7,A7
00004F24 2FFFE092  || [ B0]    B.S2        L193
00004F28 029D0099              SUB2.L1     A8,A7,A5
00004F2C 021CA461  ||          SUB2.S1     A5,A7,A4
00004F30 04509A41  ||          ADDAH.D1    A20,A4,A8
00004F34 034CDA42  ||          ADDAH.D2    B19,B6,B6
00004F38 021C909B              SUB2.L2X    B4,A7,B4
00004F3C D20C0345  || [!A0]    STDW.D1T1   A5:A4,*+A3[0x0]
00004F40 0219A2E6  ||          LDW.D2T2    *+B6[0xD],B4
00004F44 029CB09B              SUB2.L2X    B5,A7,B5
00004F48 0B1803E7  ||          LDDW.D2T2   *+B6[0x0],B17:B16
00004F4C 02200364  ||          LDDW.D1T1   *+A8[0x0],A5:A4
00004F50 029962E6              LDW.D2T2    *+B6[0xB],B5
00004F54 C003E059     [ A0]    SUB.L1      A0,1,A0
00004F58 094901A1  ||          ADD.S1      8,A18,A18
00004F5C D2180347  || [!A0]    STDW.D1T2   B5:B4,*+A6[0x0]
00004F60 0398FEC2  ||          ADDAD.D2    B6,0x7,B7
00004F64          DW$L$_ALPHA$3$E:
00004F64 03202365              LDDW.D1T1   *+A8[0x1],A7:A6
00004F68 041823E6  ||          LDDW.D2T2   *+B6[0x1],B9:B8
00004F6C 0B1863E6              LDDW.D2T2   *+B6[0x3],B23:B22
00004F70 0A1843E6              LDDW.D2T2   *+B6[0x2],B21:B20
00004F74 021982E6              LDW.D2T2    *+B6[0xC],B4
00004F78 031942E7              LDW.D2T2    *+B6[0xA],B6
00004F7C 01CA1048  ||          EXT.S1      A18,16,16,A3
00004F80 04D07A41              ADDAH.D1    A20,A3,A9
00004F84 0D1C03E7  ||          LDDW.D2T2   *+B7[0x0],B27:B26
00004F88 0B10BC30  ||          SADD2.S1X   A5,B4,A22
00004F8C 09101FDB              OR.L2X      0,A4,B18
00004F90 081883E5  ||          LDDW.D2T1   *+B6[0x4],A17:A16
00004F94 01DCFC30  ||          SADD2.S1X   A7,B23,A3
00004F98 031C02E7              LDW.D2T2    *+B7[0x0],B6
00004F9C 0254BC31  ||          SADD2.S1X   A5,B21,A4
00004FA0 039D3C32  ||          SADD2.S2X   B9,A7,B7
00004FA4 02124C33              SADD2.S2    B18,B4,B4
00004FA8 0420DC31  ||          SADD2.S1X   A6,B8,A8
00004FAC 029922E6  ||          LDW.D2T2    *+B6[0x9],B5
00004FB0 0210985B              MAX2.L2X    B4,A4,B4
00004FB4 0A98DC31  ||          SADD2.S1X   A6,B6,A21
00004FB8 04424C32  ||          SADD2.S2    B18,B16,B8
00004FBC 081F7C33              SADD2.S2X   B27,A7,B16
```

Figure 5.11: Assembly code of the alpha() function (3/5).

```
00004FC0 0214FC30  ||        SADD2.S1X     A7,B5,A4
00004FC4 02110859            MAX2.L1       A8,A4,A4
00004FC8 044A1C31  ||        SADD2.S1X     A16,B18,A8
00004FCC 03D4F85B  ||        MAX2.L2X      B7,A21,B7
00004FD0 02D24C32  ||        SADD2.S2      B18,B20,B5
00004FD4 00E403A3            MVC.S2        B25,CSR
00004FD8 03D8DC31  ||        SADD2.S1X     A6,B22,A7
00004FDC 04D8B85A  ||        MAX2.L2X      B5,A22,B9
00004FE0 0344BC31            SADD2.S1X     A5,B17,A6
00004FE4 031D285B  ||        MAX2.L2       B9,B7,B6
00004FE8 0498DC32  ||        SADD2.S2X     B6,A6,B9
00004FEC 03250941            ADD.D1        A9,0x8,A6
00004FF0 029E185B  ||        MAX2.L2X      B16,A7,B5
00004FF4 02990859  ||        MAX2.L1       A8,A6,A5
00004FF8 0394BC30  ||        SADD2.S1X     A5,B5,A7
00004FFC 03A0F858            MAX2.L1X      A7,B8,A7
00005000 02909859            MAX2.L1X      A4,B4,A5
00005004 0394B85A  ||        MAX2.L2X      B5,A5,B7
00005008 0218E37B            PACKLH2.L2    B7,B6,B4
0000500C 0298E223  ||        PACKHL2.S2    B7,B6,B5
00005010 03180347  ||        STDW.D1T2     B7:B6,*+A6[0x0]
00005014 02247858  ||        MAX2.L1X      A3,B9,A4
00005018 029006A3            OR.S2         0,B4,B5
0000501C 021408F3  ||        OR.D2         0,B5,B4
00005020 0310A85B  ||        MAX2.L2       B5,B4,B6
00005024 0210E858  ||        MAX2.L1       A7,A4,A4
00005028 023C43C7            STDW.D2T2     B5:B4,*+SP[0x2]
0000502C 0290A399  ||        PACKHL2.L1    A5,A4,A5
00005030 0190A421  ||        PACKLH2.S1    A5,A4,A3
00005034 02240344  ||        STDW.D1T1     A5:A4,*+A9[0x0]
00005038 081406A1            OR.S1         0,A5,A16
0000503C 088C08F1  ||        OR.D1         0,A3,A17
00005040 020CA858  ||        MAX2.L1       A5,A3,A4
00005044 083C23C5            STDW.D2T1     A17:A16,*+SP[0x1]
00005048 02189858  ||        MAX2.L1X      A4,B6,A4
0000504C 039207B0            ROTL.M1       A4,0x10,A7
00005050 00000000            NOP
00005054 039C8858            MAX2.L1       A4,A7,A7
00005058 029C6099            SUB2.L1       A3,A7,A5
0000505C 021CA460  ||        SUB2.S1       A5,A7,A4
00005060 02240345            STDW.D1T1     A5:A4,*+A9[0x0]
00005064 029CB09A  ||        SUB2.L2X      B5,A7,B5
00005068 021C909A            SUB2.L2X      B4,A7,B4
0000506C 02180346            STDW.D1T2     B5:B4,*+A6[0x0]
```

Figure 5.12: Assembly code of the alpha() function (4/5).

```
{

 m=i<<5;
  k=(i+1)<<3;
  n=i<<3;

/*s:2,s:0*/_amem4(&alpha1[k])  =_max2(_max2(_sadd2(_amem4(&alpha1[n]   ),_amem4(&gamma1[m]   ))
                                         ,_sadd2(_amem4(&alpha1[2+n] ),_amem4(&gamma1[18+m])))
                                   ,_max2(_sadd2(_amem4(&alpha1[4+n] ),_amem4(&gamma1[28+m]))
                                         ,_sadd2(_amem4(&alpha1[6+n ]),_amem4(&gamma1[14+m])))); /*s:1,s:0*/
/*s:3,s:1*/_amem4(&alpha1[2+k])=_max2(_max2(_sadd2(_amem4(&alpha1[n]   ),_amem4(&gamma1[24+m]))
                                         ,_sadd2(_amem4(&alpha1[2+n] ),_amem4(&gamma1[10+m])))
                                   ,_max2(_sadd2(_amem4(&alpha1[4+n] ),_amem4(&gamma1[4+m] ))
                                         ,_sadd2(_amem4(&alpha1[6+n] ),_amem4(&gamma1[22+m])))); /*s:2,s:3*/
/*s:5,s:7*/_amem4(&alpha1[4+k])=_max2(_max2(_sadd2(_amem4(&alpha1[n]   ),_amem4(&gamma1[8+m] ))
                                         ,_sadd2(_amem4(&alpha1[2+n] ),_amem4(&gamma1[26+m])))
                                   ,_max2(_sadd2(_amem4(&alpha1[4+n] ),_amem4(&gamma1[20+m]))
                                         ,_sadd2(_amem4(&alpha1[6+n] ),_amem4(&gamma1[6+m] )))); /*s:6,s:7*/
/*s:4,s:6*/_amem4(&alpha1[6+k])=_max2(_max2(_sadd2(_amem4(&alpha1[n]   ),_amem4(&gamma1[16+m]))
                                         ,_sadd2(_amem4(&alpha1[2+n] ),_amem4(&gamma1[2+m] )))
                                   ,_max2(_sadd2(_amem4(&alpha1[4+n] ),_amem4(&gamma1[12+m]))
                                         ,_sadd2(_amem4(&alpha1[6+n ]),_amem4(&gamma1[30+m])))); /*s:5,s:4*/


temp_alpha[0]=_packhl2(_amem4(&alpha1[2+k]),_amem4(&alpha1[k]));
temp_alpha[1]=_packlh2(_amem4(&alpha1[2+k]),_amem4(&alpha1[k]));
temp_alpha[2]=_packhl2(_amem4(&alpha1[6+k]),_amem4(&alpha1[k+4]));
temp_alpha[3]=_packlh2(_amem4(&alpha1[6+k]),_amem4(&alpha1[k+4]));

Normal_alpha=_max2(_max2(temp_alpha[0],temp_alpha[1]),_max2(temp_alpha[2],temp_alpha[3]));
rotl_alpha=_rotl(Normal_alpha,16);
max_alpha=_max2(Normal_alpha,rotl_alpha);

for(u=0;u<4;u++)
{  s=u<<1;
     _amem4(&alpha1[s+k])=_sub2(temp_alpha[u],(max_alpha));
}

}

}
00005070 07800C52           ADDK.S2      24,SP
00005074 008CA362           BNOP.S2      B3,5
```

Figure 5.13: Assembly code of the alpha() function (5/5).

CTC decoder are as follows :

$$
\begin{aligned}
\ln P[d_k = 00] &= -\max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)], \\
\ln P[d_k = 01] &= L_1^a(d_k) - \max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)], \\
\ln P[d_k = 10] &= L_2^a(d_k) - \max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)], \\
\ln P[d_k = 11] &= L_3^a(d_k) - \max[0, L_1^a(d_k), L_2^a(d_k), L_3^a(d_k)]. \qquad (5.2)
\end{aligned}
$$

We see that it requires 4 additions per trellis stage over 240 trellis stages for 480 information bits, for a total of $4{\cdot}240 = 960$ additions. About three $_max2()$ function calls are needed per trellis stage, for a total of $3{\cdot}240 = 720$ over 480 information bits. For the gamma function, 4 additions are needed per branch (see 5.1). With 8 states per trellis stage, 4 branches per

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 1380
;*      Loop opening brace source line   : 1381
;*      Loop closing brace source line   : 1449
;*      Known Minimum Trip Count         : 240
;*      Known Maximum Trip Count         : 240
;*      Known Max Trip Count Factor      : 240
;*      Loop Carried Dependency Bound(^) : 19
;*      Unpartitioned Resource Bound     : 14
;*      Partitioned Resource Bound(*)    : 17
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     9        7
;*      .S units                    10       10
;*      .D units                    14       10
;*      .M units                     1        0
;*      .X cross paths              12       13
;*      .T address paths            10       10
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           2        2     (.L or .S unit)
;*      Addition ops (.LSD)         15       13     (.L or .S or .D unit)
;*      Bound(.L .S .LS)            11       10
;*      Bound(.L .S .D .LS .LSD)    17*      14
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 19 Did not find schedule
;*         ii = 20 Did not find schedule
;*         ii = 21 Did not find schedule
;*         ii = 22 Did not find schedule
;*         ii = 23 Did not find schedule
;*         ii = 24 Did not find schedule
;*         ii = 25 Did not find schedule
;*         ii = 26 Schedule found with 2 iterations in parallel
;*      Done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
;*      Collapsed prolog stages     : 1
;*      Minimum required memory pad : 0 bytes
;*
;*      Minimum safe trip count     : 1
;*----------------------------------------------------------------------------*
L193:    ; PIPED LOOP PROLOG
;** --------------------------------------------------------------------------*
L194:    ; PIPED LOOP KERNEL
```

Figure 5.14: Software pipeline information of the alpha() function.

Table 5.7: Numbers of Intrinsic calls and arithmetic operations in Original Code for CTC Decdoding

	$_max2()$	$_sadd2()$	additions	multiplications
branch metrics	720	0	31680	30720
forward metrics	7440	7680	1920	0
backward metrics	7440	7680	1920	0
LLR	6720	15360	720	0

state, and 240 trellis stages for 480 information bits, the total is 4·4·8·240 = 30720. The total number of multiplications is similar : 4·4·8·240 = 30720. Note that we also use one shift operation for multiplying with 0.5 in (5.1). For the forward metrics, there are 4 calls to $_sadd2()$ × 8 states × 240 trellis stages, giving a total of 4·8·240 = 7680. The $_max2()$ are called 3 times × 8 states × 240 trellis stages, for a total of 3·8·340 = 5760 times per 480 information bits. In addition, there are 8 subtractions and 7 $_max2()$ calls per trellis stage for normalization. The backward metrics are the same with the forward metrics in amount of computation. For the LLR values, there are 28 $_max2()$ calls per trellis stage and 2 $_sadd2()$ calls per branch × 4 branches × 8 states × 240 trellis stages, plus 3 subtractions per trellis stage. For convenience, we summarize the above analysis in Table 5.7.

For the branch metric computation in the improved code, the additions needed in computing the a priori probability are the same as the original, but the *Gamma_Table* function uses the $_dotp2()$ intrinsic to replace 2 multiplications and one addition. The total number of $_dotp2()$ calls are 2 per branch × 4 branches × 8 states × 240 trellis stages, yielding 2·4·8·240 = 15360. The number of additions is 1·4·8·240 = 7680. For gamma function there is 1 addition × 4 branches × 8 states × 240 trellis stages, which total to 7680. For the forward metrics, we use 12 $_max2()$ and 16 $_sadd2()$ intrinsics per trellis stage and 3 $_max2()$ and 4 $_sub2()$ calls to compute normalization. The backward metrics are the same as forward. The

Table 5.8: Numbers of Intrinsic Calls and Arithmetic Operation in Improved Code

	$_max2()$	$_sadd2()$	$_dotp2()$	$_sub2()$	additions
branch metrics	720	0	15360	0	16320
forward metrics	3600	3840	0	960	0
backward metrics	3600	3840	0	960	0
LLR	2880	7680	0	0	720

LLR values uses 12 $_max2()$, 32 $_sadd2()$ and 2 $_sub2()$ calls per trellis stage. In Table 5.8, we show the resulting amount of computation for ease of comparison with that of the original code.

5.4.2 Processing Rate of CTC Decoder

Overall, we can get the cycles of 480 information bits, rate 1/2 CTC decoder between original code and improved code are 209900 cycles and 126918 cycles with QPSK modulation in one iteration, respectively. For CCS operation speed is 10^9 cycles/second, so we can get their decoding information processing rates which are 2286 Kbps and 3782 Kbps, respectively. Note that the CTC decoder is not be included the external functions, like as, de-modulator and de-puncturing.

In Tables 5.9 and 5.10, we show the processing rates of original code and improved code for two iterations and four iterations. Obviously, the processing rate is decreased with iteration count. But the amount of decrease of processing rate is different. For original code, the processing rate from 2 iterations to 4 iterations is decreased about 44%, and for the improved code is decreased about 41%. That means, for higher iteration counts the processing rate of improved code is better and better than original code. The reason is that we do not repeat the computation of the branch metrics from the received systematic and

ES 1896

Table 5.9: Information Data Processing Rate Calculated from CCS for Original Code for 480 Information Bits, Rate 1/2 Coding

Number of Iterations	CTC_Decoder Cycles	Information Data Rate (Kbps)
2	336,568	1,426
4	592,273	810

Table 5.10: Information Data Processing Rate Calculated from CCS for Improved Code for 480 Information Bits, Rate 1/2 Coding

Number of Iterations	CTC_Decoder Cycles	Information Data Rate (Kbps)
2	193,015	2487
4	324,278	1480

parity bits.

Finally, we compare the decoder processing rates of rate-1/2 tail-biting CC without using the VCP, rate-1/2 tail-biting CC with the VCP, rate-1/2 CTC with 4 iterations of improved code and rate-1/2 LDPC. Source of these implementation are described in [22], [23]. We get the information data processing rates in decoding for tail-biting CC and LDPC codes from [23] and tail-biting CC with the VCP from [22]. And we use CCS profiles to estimate the decoding processing rate for CTC. The results are shown in Table 5.11

Table 5.11: Comparison of Decoder Speeds for Tail-Biting CC, CTC, and LDPC Calculated from CCS

CC Information Data Rate Without Using VCP for Rate-1/2 QPSK (Kbps) [23]	CC Information Data Rate With VCP for Rate-1/2 QPSK (Kbps) [22]	CTC Information Data Rate for Rate-1/2 QPSK with 4 Iterations (Kbps)	LDPC Information Data Rate for Rate-1/2 QPSK (Kbps) [23]
832	8,938	1480	7.6

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we first present a part of FEC in IEEE 802.16e, which contain CTC encoder, channel interleaver and bit selection. Second, we present the turbo decoding algorithm BCJR with max-log-MAP, evaluating the performance of CTC and compared the results with numerical results. Finally, we optimize CTC decoder on DSP implementation.

In performance simulation, in order to compensate the max-log-MAP performance loss, we use a scaling factor to scale down the extrinsic message, and it improve performance about 0.1 to 0.2 dB. Then we focused on complexity-reducing max-log-MAP decoding algorithm. We convert the floating-point input values to fixed-point, proposing scaling method and clipping method. For scaling method we could use S14.1 and for clipping method using (Din4-Ex16) to implement the decoder. In our last simulation, the clipping method is better than scaling method about 0.5 dB under QPSK, 16QAM and 64QAM at rate-1/2, respectively. On DSP implementation, we arrange state order to reduce decoder complexity, which contain forward metrics, backward metrics and branch metrics. In order to achieve parallelism, we using a lot intrinsic functions to access two 16-bit at a time, the 50.21% is reduction in complexity. In conclusion, in our decoder with 4 iterations, we can approach data rate

about 1500 Kbps.

6.2 Future Work

There are several possible extension for our research:

- In CTC decoder, the LLR function is execute many cycles, the parallelism is failed, we may rewrite our code to achieve software pipeline.
- The procedure of HARQ (Hybrid Automatic Repeat reQuest) is important implementation in FEC, it can correct appear frequently error to reduce re-transmit times.
- For HARQ, it will longer length for encoder, we can used sliding window to reduce complexity.

Bibliography

[1] IEEE Std 802.16^{TM}-2009 (Revision of IEEE Std 802.16-2004), *IEEE Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Broadband Wireless Access Systems.* New York: IEEE, May 2009.

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

[3] 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.

[4] 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.

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

[6] 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.

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

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

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

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

[11] Sundance, SMT6400 Help.chm.

[12] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide.* Lit. no. SPRU189, Oct. 2000.

[13] Texas Instruments, *TMS320C6414T, TMS320C6415T, TMS320C6416T Fixed-Point Digital Signal Processors.* Lit. no. SPRS226A, Mar. 2004.

[14] Texas Instruments, *TMS320C6000 DSP cache User's Guide.* Lit. no. SPRU656A, May 2003.

[15] Texas Instruments, *Code Composer Studio User's Guide.* Lit. no. SPRU328B, Feb. 2000.

[16] Texas Instruments, *TMS320C6000 Code Composer Studio Tutorial.* Lit. no. SPRU301C, Feb. 2000.

[17] Texas Instruments, *TMS320C6000 Programmer's Guide.* Lit. no. SPRU198I, Mar. 2006.

[18] Texas Instruments, *TMS320C6000 Optimizing Complier User Guide.* Lit. no. SPRU187K, Oct. 2002.

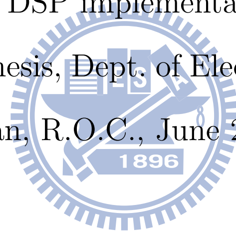
[19] Y. Wu, B. D. Woerner and T. K. Blankenship, "Data width requirements in SISO decoding with modulo normalization," *IEEE Trans on Commun,* vol. 49, pp. 1861–1868, Nov. 2001.

[20] G. Jeong and D. Hsia, "Optimal quantization for soft-decision turbo decoder," in *Proc. IEEE Vehicular Technology Conference,* Amsterdam, The Netherlands, Sep. 1999.

[21] J. Vogt and A. Finger, "Improving the max-log-MAP turbo decoder," *Electronics Letters,* vol. 36, pp. 1937–1939, Nov. 2000.

[22] Jia-Fong Chen,"Study in WiMAX channel coding techniques and associated digital signal processor implementation,"M.S. thesis, Dept. Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C, June 2008.

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



作者簡歷

姓名：曾劭學 (Shao-Hsueh Tseng)

出生地：台北市

學歷：國立南港高中

中興大學電機工程系學士

交通大學電子研究所碩士(2007.9~2009.11)

研究領域：通訊系統、通道編碼及數位訊號處理

論文題目：WiMAX 迴旋渦輪碼技術

與數位訊號處理器實現

(WiMAX Convolutional Turbo Code Technology and

Digital Signal Processor Implementation)