國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

IEEE 802.16a 標準之前向誤差改正編碼於數位訊號處理器平台上之實現與最佳化

DSP Implementation and Optimization of the Forward Error Correction Scheme in IEEE 802.16a Standard

研 究 生：李仰哲

指導教授：杭學鳴 博士

中 華 民 國 九 十 三 年 六 月

IEEE 802.16a 標準之前向誤差改正編碼於數位訊號處理器平台上之實現與最佳化

DSP Implementation and Optimization of the Forward Error Correction Scheme in IEEE 802.16a Standard

研 究 生：李仰哲　　Student：Young-Tse Lee

指導教授：杭學鳴 博士　　Advisor：Dr. Hsueh-Ming Hang

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 Institute of Electronics
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of Requirements
for the Degree of
Master of Science
in
Electronics Engineering
June 2004
Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 三 年 六 月

IEEE 802.16a 標準之前向誤差改正編碼於數位訊號處理器平台上之實現與最佳化

研究生: 李仰哲　　指導教授: 杭學鳴博士

國立交通大學
電子工程學系　電子工程研究所

摘要

在 IEEE 802.16a 無線通訊標準中，於系統的傳送端與接收端都分別訂定了前向誤差改正編碼的機制，藉此減低通訊頻道中雜訊失真的影響。本篇論文的重點在於，實現標準所訂定的前向誤差改正編碼系統於數位訊號處理器(DSP)平台上，並且針對 DSP 平台(此平台包含 DSP 與 FPGA)的特性以及前向誤差改正編碼的演算法進行程式的改進。在此篇論文中、我們將標準中制訂的四個必備的前向誤差改正編碼系統，實現在以德州儀器公司所發展的 DSP 為核心的平台上。由於我們關注的重點在於程式的執行效率，因此在簡短地介紹過我們所使用的前向誤差改正編碼的演算法以及 DSP 平台的架構與軟體最佳化技巧後，我們將逐步地闡述如何在 DSP 平台上最佳化我們的程式。最後我們可以在程式執行效率上達到明顯的進步，前向誤差改正編碼的編碼器部分，經過改進後，於 DSP 模擬器上、可以達到每秒 7984K 位元的處理速度，而解碼器的部分可以達到每秒 750K 位元的處理速度。此外、針對我們所使用的數位處理器平台上內建的 Xilinx FPGA，我們也做了兩項模擬來評估在 Viterbi 解碼器最佳化中的瓶頸:”加-比-選”(ACS)單元在 FPGA 上的處理效率。受限於 DSP 與 FPGA 之間的傳輸頻寬，原本在 FPGA 上的處理速度應為每秒 45M 64 狀態的 ACS 單元，實際上僅能達到每秒 32M 64 狀態的處理速度。

DSP Implementation and Optimization of the Forward Error Correction Scheme in IEEE 802.16a Standard

Student: Young-Tse Lee *Advisor: Dr. Hsueh-Ming Hang*

Department of Electronics & Institute of Electronics
National Chiao Tung University

Abstract

In the IEEE 802.16a wireless communication standard, a Forward Error Correction (FEC) mechanism is presented at both the transmitter and the receiver sides to reduce the noisy channel effect. The focus of this thesis is DSP implementation of the FEC scheme defined in IEEE 802.16a standard and modifying FEC algorithms to match the architecture of DSP and FPGA platforms. We have implemented four required FEC schemes defined in the standard on the Texas Instruments (TI) TMS320C6416 digital signal processor (DSP). After a brief review of the algorithms, we describe the DSP hardware architecture and its software optimization techniques. We then explain how we optimize the FEC programs on the DSP platform step by step since the speed performance is our major concern. Finally, we achieve a significant improvement on the speed performance. At the end, the improved FEC encoder can achieve a data processing rate of 7984 Kbits/sec and the improved FEC decoder can achieve a processing rate of 750 Kbits/sec on the TI C64xx DSP simulator. Furthermore, we have done two simulations to evaluate the

data processing rate of the Add-Compare-Select (ACS) unit implemented on the Xilinx FPGA since the ACS unit is the speed bottleneck of the Viterbi decoder. Due to the constraint on transmission bandwidth between DSP and FPGA, the processing rate of the ACS unit on FPGA can only approach 32M (64 states/sec), while the actual processing rate on FPGA is 45M (64 states/sec).

誌謝

這篇論文能夠順利完成，首先要感謝我的指導教授 杭學鳴 博士這兩年來的悉心指導。在研究的過程中，常有遇到阻礙而停滯不前的時候，老師總是能夠適時的敦促和提供寶貴的意見，使我得以克服在研究中所遭遇的瓶頸。除了本身的研究方向，老師也不斷地鼓勵我們多接觸其它相關領域，厚實未來作進一步研究的基礎。除此之外，老師也常能關心並體諒我們在生活與學業上的種種問題，使我在研究的過程中沒有遭受到額外的壓力。

此外也要特別感謝通訊電子與訊號處理實驗室，提供了充足的軟硬體資源，供我們在研究中充分的利用。也感謝實驗室全體成員，營造了一個充滿活力與和諧的環境氣氛，為平常稍嫌沉悶的研究生活增添了不少趣味。感謝陳繼大、楊政翰與蔡家揚學長，在我的研究過程中不吝提供寶貴的經驗與協助，減少我許多摸索的時間。另外，也要感謝曾建統與劉明瑋同學，在論文的研究中經常和我討論切磋，使我感到受益良多。

最後，要感謝的是我的家人，他們讓我能全心地從事研究工作而無後顧之憂。沒有家人在背後的支持與體諒，也就沒有這篇論文的誕生。

在此僅以這篇論文獻給所有幫助過我，陪我走過這一段歲月的師長、同儕與家人，謝謝！

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Digital wireless transmission of multimedia contents is a trend in the consumer electronics field in the future. Due to the demand for wireless communication of multimedia contents, high data transmission rate and mobility are needed. Thus, the OFDM modulation technique for wireless communication has been the main stream in the recent years. IEEE has completed several standards such as IEEE 802.11 series for LAN (Local Area Network) and IEEE 802.16 series for MAN (Metropolitan Area Network) based on OFDM technology. Our study is based on the IEEE 802.16a standard, which specifies the air interface of fixed broadband wireless access systems providing multiple accesses.

The advantage of digital wireless communication is based on a fact that it is convenient for consumers to receive or transmit digital contents without connecting to the transmission lines. However, there are still problems to solve in wireless communication system. One major problem is that the transmission channel is not noiseless. The transmission signals are easily interfered and distorted by several different types of noise source such as the crowd traffic, bad weather, the obstacle of buildings, etc. Multimedia service contains a broad range of contents such as audio, video, still image, and the traditional speech. These services would have unacceptable quality if they cannot detect and recover the errors introduced by the noisy channel. To improve the robustness of the wireless communication against the noisy channel

condition, the FEC (Forward–Error-Correcting Coding) mechanism and the FED (Forward–Error-Correcting Decoding) mechanism are a must to combat the channel error. Therefore, they exist in almost every commercial communication standards, including the IEEE 802.16a standard we mentioned earlier.

In this thesis, we focus on the study of the implementation of the FEC/FED scheme of the IEEE 802.16a standard on the II Quixote DSP/FPGA board. We first review the algorithms the FEC/FED used in 802.16a to understand the encoding and decoding procedure. Then, we write C programs to check the correctness of our algorithms. Finally, we implement the FEC/FED algorithm on DSP and improve its speed by optimizing the DSP programs. Furthermore, to increase the processing speed of the FED scheme further, we also use the Xilinx FPGA which is also embedded in the Quixote board as an extra hardware accelerator.

In Chapter 2, we briefly introduce the forward error correction scheme of IEEE 802.16a standard and discuss the major algorithm blocks. Also, we discuss how to implement these algorithms in C language to reduce the computational complexity.

In Chapter 3, we give a brief description of our implementation environment; it includes both the II's Quixote DSP baseboard and its communication mechanism between host PC and target DSP.

In Chapter 4, we first prelude the architecture of C6x DSP shortly and explain the impact of data and instruction types on the program execution. Then, we describe the techniques of software pipeling used by the compiler, which is helpful for writing efficient high-level programs. We then describe the optimization we have done on the Reed-Solomon decoder implemented on the TI C64 DSP. Next, the optimization of the Viterbi decoder implemented on the TI C64 DSP is discussed. We also describe the techniques used for improving the overall processing speed of the decoder. We check the processing speed before optimization and after optimization. Finally, the simulation profile of the improved FEC encoder and decoder is given to show how fast the processing rate may be achieved after optimization.

In Chapter 5, we introduce the Xilinx FPGA as an extra hardware accelerator, we discuss the implementation issues based on FPGA platform and evaluate how much improvement we may achieved with the assistance of FPGA.

At the end, we give some observations and conclusions. Possible subjects for future works are also included.

Chapter 2

Overview of IEEE 802.16a FEC Scheme

2.1 Introduction to IEEE 802.16a Standard

The IEEE 802.16a standard amends IEEE standard 802.16 by enhancing the medium access control layer and providing additional physical layer specifications in support of broadband wireless access at frequencies from 2-11GHz. The resulting standard specifies the air interface of fixed (stationary) broadband wireless access systems providing multiple services. The medium access control layer is capable of supporting multiple physical layer specifications optimized for the frequency bands of application. The standard includes particular physical layer specifications applicable to systems operating between 2 and 66 GHz. It supports point-to-multipoint and optional mesh topologies [1].

This standard is part of a family of standards for local and metropolitan area networks. The relationship between the standard and other members of the family is shown in Fig. 2.1 (The numbers in the figure refer to IEEE standard designations). The family of standards deals with the Physical and Data Link Layers as defined by the international Organization for Standardization (ISO) Open Systems Interconnection Basic Reference Model. The access standards define several types of medium access technologies and associated physical media, each appropriate for particular applications or system objectives. Other types are under investigation [1].

This thesis focus on the DSP/FPGA joint implementation and optimization issues of the IEEE 802.16a Forward Error Correction (FEC) Coding/Decoding scheme. Therefore we will concentrate on introducing the FEC specifications defined in IEEE 802.16a physical layer part in next section. At the last part of this chapter, we show the block diagrams of our simulation programs and explain what we have initiatively done to modify the implementation structure and hence reduce the computational complexity.

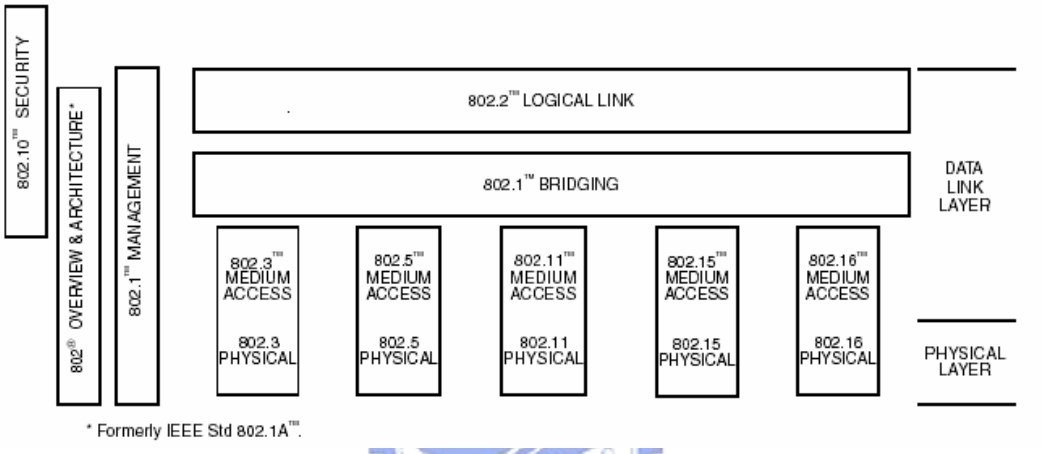


Figure 2.1: IEEE local and metropolitan area networks standards family.

2.2 IEEE 802.16a FEC Specifications

The overall physical layer structure of the channel coding scheme is shown in Fig. 2.2, whereas the Reed-Solomon Code and the Convolutional Code are major parts of the FEC scheme, the randomizer and the interleaver are additional modules for further improving the error performance of the FEC scheme. The detailed specifications of each block are introduced in the following subsections, excluding the modulator which is not implemented in our research subproject.

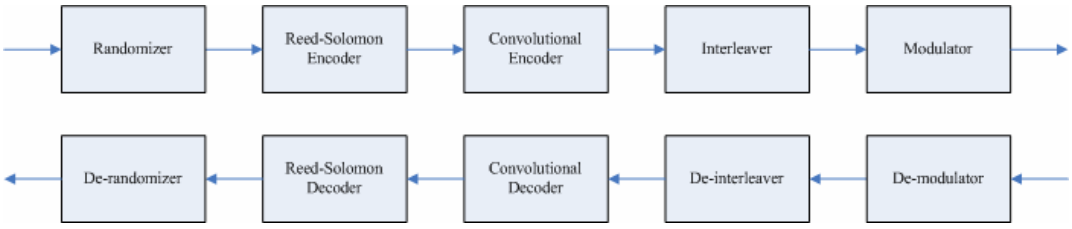


Figure 2.2: Channel coding structure in transmitter side (top) and receiver side (bottom).

2.2.1 Randomizer

Data randomization is performed on data transmitted on the DL and UL. The randomization is performed on each allocation (DL or UL), which means that for each allocation of a data block (subchannels on the frequency domain and OFDM symbols on the time domain) the randomizer shall be used independently. 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.

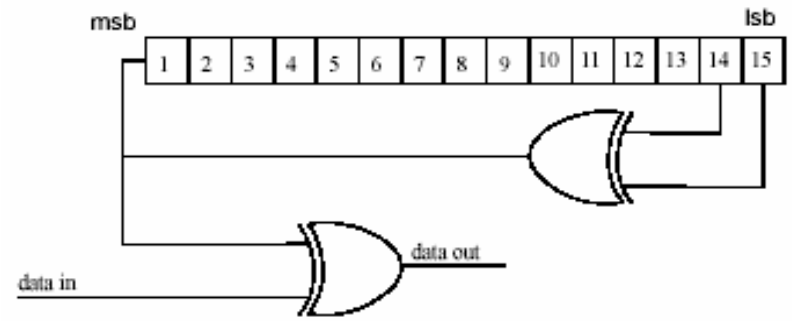


Figure 2.3: PRBS for Data Randomization.

The randomizer is a Pseudo Random Binary Sequence (PRBS) generator depicted in Fig. 2.3. As shown in the figure, the generator polynomial of the randomizer is $1+X^{14}+X^{15}$. 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 randomizer sequence is applied only to information bits. Preambles are not randomized.

The shift-register of the randomizer shall be initialized for every 1250 bytes passed through (if the allocation is larger then 1250 bytes).

In the downlink, the randomizer shall be re-initialized at the start of each frame with the sequence

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

In the uplink, the randomizer is initialized with the vector created as shown in Fig. 2.4.

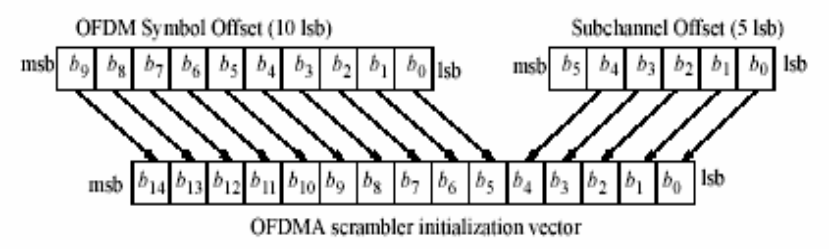


Figure 2.4: Creation of OFDMA Randomizer Initialization Vector.

2.2.2 Forward Error Correction Coding

Forward error correction is used to decrease bit error rate (BER) on noisy communication channels. This is achieved by a method known as channel coding, which adds redundant information to the transmitted data. With forward error correction, transmission errors are corrected at the decoder, without requesting a retransmission. Convolutional encoding and block coding are two major forms of channel coding [2]. In our IEEE 802.16a OFDMA project, both convolutional code and block code (Reed-Solomon Code) are employed.

The Forward Error Correction scheme used in the IEEE 802.16a standard, as shown in Fig. 2.5, consisting of the concatenation of a Reed-Solomon outer code and a rate-compatible convolutional inner code, is supported on both UL and DL. The input data streams are first divided into RS (Reed-Solomon) blocks, the block size is

determined by parameter *k* defined in RS code specification, then encoded by a RS encoder, and each RS coded block is then encoded by a convolutional encoder. Convolutional code is one kind of sequential codes, but RS code is a block code. Overall it makes the whole concatenated code a block-based coding scheme.

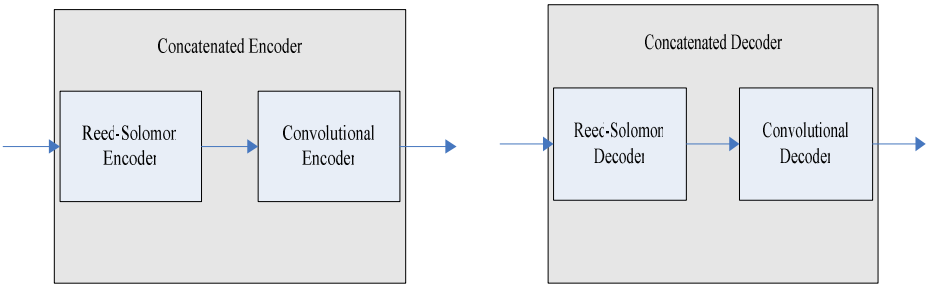


Figure 2.5: Forward Error Correction structure in transmitter side (left) and receiver side (right).

In order to make the system more flexible and adaptable to the channel condition, there are six coding-modulation schemes provided in the standard, as shown in Table 2.1(notice that 64QAM is an optional mode). The different coding rates are made by shortening and puncturing the original RS code and with puncturing of the original convolutional code. The shortened- and- punctured mechanisms in RS code can provide different block size and hence different error correcting capability through the same RS Codec (Coder / Decoder). Similarly, the convolutional code can provide variable code rates through the same codec by applying the puncturing rule. Thus it can suit the variable block size of the shortened-and-punctured RS code to achieve a desired overall coding rate.

Modulation	Uncoded Block Size (bytes)	Overall Coding Rate	Coded Block Size (bytes)	RS Code	CC Code Rate
QPSK	18	1/2	36	(24,18,3)	2/3
QPSK	26	~3/4	36	(30,26,2)	5/6
16-QAM	36	1/2	72	(48,36,6)	2/3
16-QAM	54	3/4	72	(60,54,3)	5/6
64-QAM	72	2/3	108	(81,72,4)	3/4
64-QAM	82	~3/4	108	(90,82,4)	5/6

Table 2.1: Mandatory Channel Coding per Modulation.

2.2.2.1 Reed-Solomon Code Specification

The Reed-Solomon encoding is derived from a systematic RS (N=255, K=239, T=8) code using GF(2^8),where N is the number of overall bytes after encoding, K is the number of data bytes before encoding, and T is the number of data bytes which can be corrected from errors. The galois field used in this code is generated by the field generator polynomial: $p(x) = x^8 + x^4 + x^3 + x^2 + 1$, and the codeword is generated by the code generator polynomial: $g(x) = (x + \lambda^0)(x + \lambda^1)(x + \lambda^2)\ldots(x + \lambda^{2T-1})$.

This code is shortened and punctured to enable variable block sizes and variable error-correction capability. When a block is shortened to K' data bytes, the first 239 – K' bytes of the encoder block are filled with "0"s. When a codeword is punctured to permit T' bytes to be corrected, only the first 2T' of the total 16 codeword bytes are employed.

2.2.2.2 Convolutional Code Specification

After the RS encoding process, each RS block is then encoded by the binary convolutional encoder, which has native rate of 1/2, a constraint length equal to K=7, and uses the following generator polynomials codes to derive its two code bits:

$$G_1 = 171_{OCT} \qquad \textit{FOR } \boldsymbol{X}$$

$$G_2 = 133_{OCT} \qquad \textit{FOR } \boldsymbol{Y}$$

The generator is depicted in Fig. 2.6.

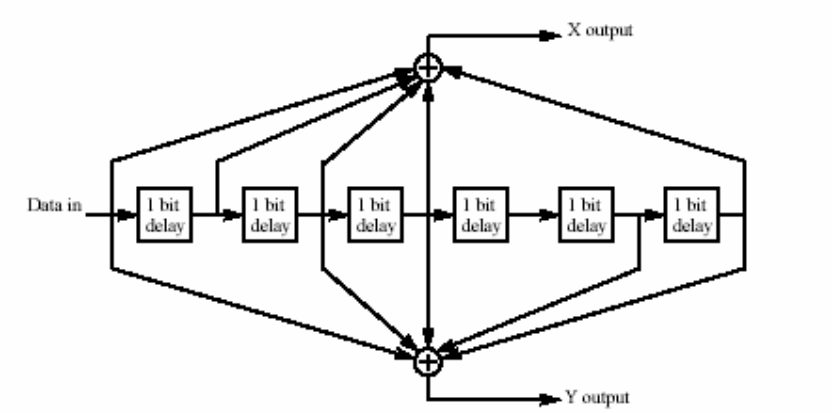


Figure 2.6: Convolutional Encoder of Rate 1/2.

Puncturing patterns and serialization order which is used to realize different code rates are defined in Table 2.2. In the table, "1" denotes a transmitted bit and "0" denotes a removed bit, whereas ***X*** and ***Y*** are in reference to Fig. 2.6.

	Code Rates		
Rate	2/3	3/4	5/6
$\mathbf{d}_{free}$	6	5	4
X	10	101	10101
Y	11	110	11010
XY	$X_1Y_1Y_2$	$X_1Y_1Y_2X_3$	$X_1Y_1Y_2X_3Y_4X_5$

Table 2.2: The Inner Convolutional Code with Puncturing Configuration.

Furthermore, a tail-biting mechanism is adopted in our convolutional code, by initializing the encoder's memory with the last data bits of the RS block being encoded.

2.2.3 Interleaver

All encoded data bits are interleaved by a block interleaver with a block size corresponding to the number of coded bits per the specified allocation, N_{cbps} (see Table 2.3) to protect the convolutional code from severe impact of burst errors and therefore increase the coding performance. The interleaver is defined by a two step permutation. The first permutation ensures that adjacent coded bits are mapped onto nonadjacent carriers. The second permutation ensures that adjacent coded bits are mapped alternately onto less or more significant bits of the constellation, thus avoiding long runs of lowly reliable bits.

Modulation	Coded Bits per Bit Interleaved Block (Ncbps)	Modulo Used (d)
QPSK	288	16
16-QAM	576	18
64-QAM	864	16

Table 2.3: Bit Interleaved Block Sizes and Modulo.

Now let N_{cpc} be the number of coded bits per carrier, i.e. 2, 4 or 6 for QPSK, 16QAM or 64QAM, respectively. Let $s = N_{cpc}/2$. Let k be the index of the coded bit before the first permutation at transmission, m be the index after the first and before the second permutation and j be the index after the second permutation, just prior to modulation mapping, and d be the modulo used for the permutation.

The first permutation is defined by the rule:

$$m = (N_{cbps}/d) * k_{mod(d)} + floor(k/d), \qquad k = 0, 1, \ldots, N_{cbps} - 1$$

The second permutation is defined by the rule:

$$J = s * floor(m/s) + (m + N_{cbps} - floor(d*m\, N_{cbps}))_{mod(s)}, \qquad m = 0, 1, \ldots, N_{cbps} - 1$$

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

The first permutation is defined by the rule:

$$m = s * floor(j/s) + (j + floor(d*j/ N_{cbps}))_{mod(s)}, \quad j = 0, 1, \ldots, N_{cbps} -1$$

The second permutation is defined by the rule:

$$K = d * m – (N_{cbps} -1) * floor (d*m/ N_{cbps}), \quad m = 0, 1, \ldots, N_{cbps} -1$$

The first permutation in the de-interleaver is the inverse of the second permutation in the interleaver, and conversely.

2.3 Implementation Issues of the FEC Scheme

Detailed explanation of the FEC coding and decoding algorithms is given in this section. The block diagrams of our simulation programs are also provided in each section. Also we will describe how we reduce the computational complexity on PCs.

2.3.1 Reed-Solomon Code

2.3.1.1 Encoding of Shortened and Punctured Reed-Solomon Codes

The Reed-Solomon code defined in IEEE 802.16a standard is a modified RS code which is derived from the standard systematic (255, 239, 8) RS code as mentioned in section 2.2.2. In this section, we first give an example to illustrate how the encoding

process has been done. Secondly, the block diagram of our RS encoder program is given too.

The (48, 36, 6) RS code is chosen from Table 2.2 as an example to show the details of encoding process. Before talking about the encoding process, we must note one thing that the galois field defined in the IEEE 802.16a standard is GF(2^8), it means that each element, i.e. $\mathbf{I_{238} \sim I_0}$, $\mathbf{R_{15} \sim R_0}$, mentioned below denotes a byte (8 bits). First we let the information data bytes which are inputs to the systematic (255, 239, 8) RS code be represented as polynomial form shown below:

$$\mathbf{I}(x) = \mathbf{I_{238}}x^{238} + \mathbf{I_{237}}x^{237} + \ldots\ldots\ldots + \mathbf{I_{36}}x^{36} + \mathbf{I_{35}}x^{35} + \ldots\ldots + \mathbf{I_1}x + \mathbf{I_0}$$

$$= (\mathbf{I_{238}}, \mathbf{I_{237}}, \ldots, \mathbf{I_{36}}, \mathbf{I_{35}}, \ldots, \mathbf{I_1}, \mathbf{I_0})$$

Then the resulting systematic (255, 239, 8) RS codeword is given by

$$\mathbf{C}(x) = \mathbf{I}(x) \cdot x^{16} + \mathbf{R}(x)$$

$$= (\mathbf{I_{238}}, \mathbf{I_{237}}, \ldots, \mathbf{I_{36}}, \mathbf{I_{35}}, \ldots, \mathbf{I_1}, \mathbf{I_0}, \mathbf{R_{15}}, \mathbf{R_{14}}, \ldots, \mathbf{R_3}, \mathbf{R_2}, \mathbf{R_1}, \mathbf{R_0})$$

The remainder polynomial $\mathbf{R}(x)$ can be represented as below:

$$\mathbf{R}(x) = \mathbf{I}(x) \cdot x^{16} \bmod \boldsymbol{g}(x)$$

$$= (\mathbf{R_{15}}, \mathbf{R_{14}}, \ldots, \mathbf{R_3}, \mathbf{R_2}, \mathbf{R_1}, \mathbf{R_0})$$

Where the exponent of x is derived from N – K = 16.

The encoding process shown above is the standard (255, 239, 8) RS code. In order to match the (48, 36, 6) code requirement, shortening and puncturing are needed. In other words, we have to modify the existing codeword further. Initially we set the first (239 – 36) = 203 input data bytes to zero and pad with 36 information data bytes, for example, the input data bytes becomes:

$\mathbf{I}(x) = (\mathbf{0}, \mathbf{0}, \mathbf{0}, \ldots, \mathbf{0}, \mathbf{I_{35}}, \mathbf{I_{34}}, \mathbf{I_{33}}, \ldots, \mathbf{I_2}, \mathbf{I_1}, \mathbf{I_0})$, totally 203 zeros in the beginning.

Then let the 239 data bytes be encoded by the standard (255, 239, 8) RS encoder, after it has been encoded, we discard the last 4 bytes of the codeword. Finally we have 48 bytes codeword, for example, the 48 bytes codeword is shown as below:

$$\mathbf{C}(x) = (\mathbf{I}_{35}, \mathbf{I}_{34}, \mathbf{I}_{33}, \ldots, \mathbf{I}_2, \mathbf{I}_1, \mathbf{I}_0, \mathbf{R}_{15}, \mathbf{R}_{14}, \ldots, \mathbf{R}_7, \mathbf{R}_6, \mathbf{R}_5, \mathbf{R}_4)$$

Similarly, the other types of shortened-and-punctured RS code listed in Table 2.2 can be acquired by performing the same procedure as discussed above, except for the (81, 72, 4) RS code which is derived from (80, 72, 4) shortened-and-punctured RS code by inserting a zero byte in the beginning of codeword.

The block diagram of our RS encoder is shown in Fig. 2.7, where the block named as shortened-and-punctured block is to discard the first 203 zero bytes (shortening) and the last 4 bytes (puncturing) of the RS codeword. The details of the LFSR block is shown in Fig. 2.8, we employ the Linear Feedback Shift Register (LFSR) structure to implement the RS encoder block diagram as shown in Fig. 2.9 [3].

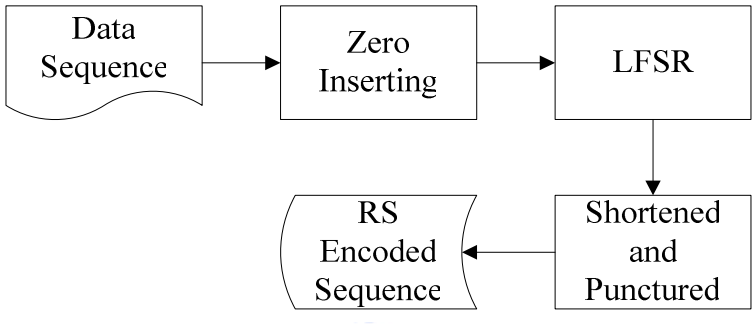


Figure 2.7: Block Diagram of the RS Encoder Program.

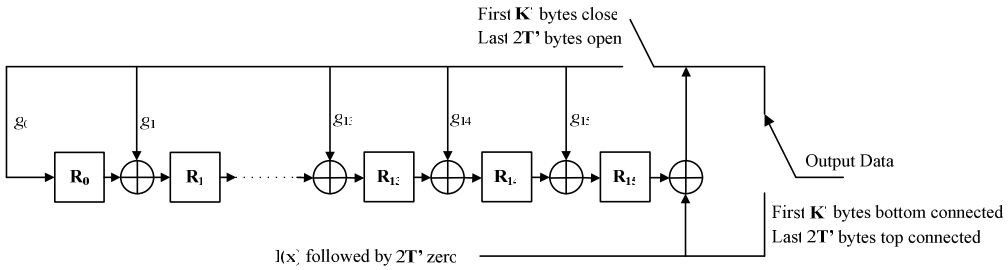


Figure 2.8: The Linear Feedback Shift Register Structure of RS Encoder.

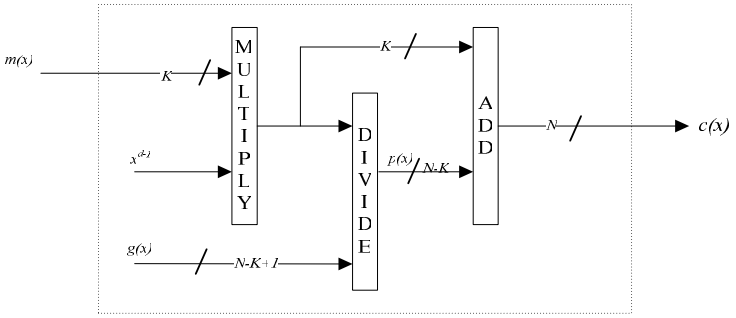


Figure 2.9: Block Diagram of a Conventional RS Encoder.

2.3.1.2 Decoding of Shortened and Punctured Reed-Solomon Codes

In order to understand how to decode a shortened-and-punctured RS code, we also take the (48, 36, 6) RS code as an example. First we acquire 48 data bytes from the receiver side, prepending with 203 zero bytes and padding with 4 zero bytes in the end. Then, we have a data block whose size equals 255 bytes. Afterwards we can employ a standard (255, 239, 8) RS decoder to decode it with the last 4 zero bytes of the codeword marked as erasures.

A (48, 36, 6) RS decoder consists of the following main steps:

1. Syndrome computation:

   Insert 203 bytes of zero before the 48 bytes received data and insert 4 bytes of zero in the locations marked as erasure then compute the syndromes.

   $S_k = \sum_{i=0}^{254} r_i \alpha^{ik}$, for $1 \le k \le 16$, whereas the r_i is the received data after zero inserting.

2. Erasure locator polynomial computation:

   $\Lambda(x) = \prod_{j=1}^{s} (1 - Z_j x) = \sum_{j=0}^{s} \Lambda_j x^j$, whereas the Z_j is the jth erasure location and the s is the number of erasures.

3. Find the error location polynomial coefficient by solving

$$\begin{bmatrix} S_1 & S_2 & \cdots & S_7 & S_8 \\ S_2 & S_3 & \cdots & S_8 & S_9 \\ \vdots & \vdots & & \vdots & \vdots \\ S_8 & S_7 & & S_{14} & S_{15} \end{bmatrix} \begin{bmatrix} \Lambda_8 \\ \Lambda_7 \\ \vdots \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} -S_9 \\ -S_{10} \\ \vdots \\ -S_{16} \end{bmatrix} \quad (1)$$

Then find the error location by finding the roots of $\Lambda(x)$.

(When performing erasure and error decoding, the syndrome shown in (1) shall be replaced by Forney syndrome : $T_k = \sum_{j=0}^{s} \Lambda_j S_{k+s-j}$, for $1 \le k \le d-1-s$)

4. Find the error and erasure magnitude by solving

$$\begin{bmatrix} X_1 & X_2 & \cdots & X_v \\ X_1^2 & X_2^2 & \cdots & X_v^2 \\ \vdots & \vdots & & \vdots \\ X_1^v & X_2^v & \cdots & X_v^v \end{bmatrix} \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_v \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_v \end{bmatrix} \quad (2)$$

5. Let t denote the number of errors, s denote the number of erasures If 2s + t > T (T = 6 in the case of (48, 36, 6) RS code), it means that the number of errors and erasures exceed the amount that can be recovered by this RS code. Thus, the received data bytes would be left unchanged.

For computing (1) and (2), there are two well-known and conventional algorithms existing. One is called Euclidean's algorithm, and the other is called Berlekamp-Massey (BM) algorithm. The Euclidean's algorithm is used to compute the eqns. (1) and (2). The BM algorithm is used to compute eqn. (1). In our case, we choose the BM algorithm to compute (1) and employ the Forney algorithm to solve (2).

A flowchart of the BM algorithm for computing the error/erasure locator polynomial in RS decoder is shown in Fig. 2.10 [3]. At the end of the iterations, we can obtain the error/erasure locator polynomial via the $\Lambda^{(n-k)}(x)$ polynomial.

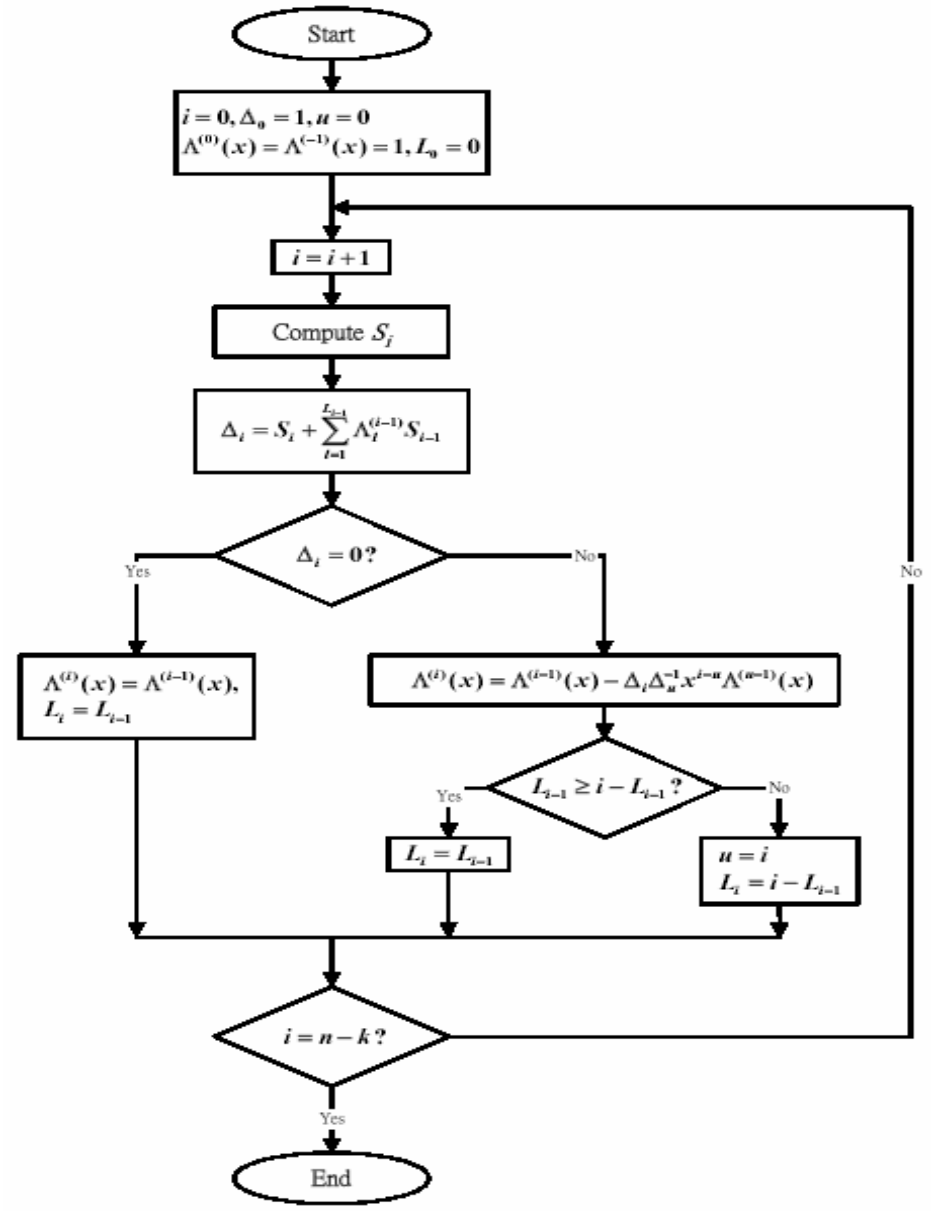


Figure 2.10: Flowchart of the Berlekamp-Massey Algorithm.

In addition, the RS decoding procedure introduced previously can be further simplified based on an improved time-domain RS decoder proposed in [19]. The major difference between the new decoder and the previous one is that decoding in the new decoder do not require pre-computating the Forney syndrome and post-computing the errata locator polynomial, it just simply initialize the BM algorithm with the erasure locator polynomial and afterward the errata locator polynomial can be obtained in the end of iteration of BM algorithm.

The block diagram of our RS decoder is shown in Fig. 2.11, where the syndrome computation is done by employing the circuit shown in Fig. 2.12 then fed to the BM algorithm, the chein search performed after BM algorithm is used to find the roots of the error/erasure locator polynomial and the forney algorithm is for the purpose of computing the magnitude of the error/erasure .

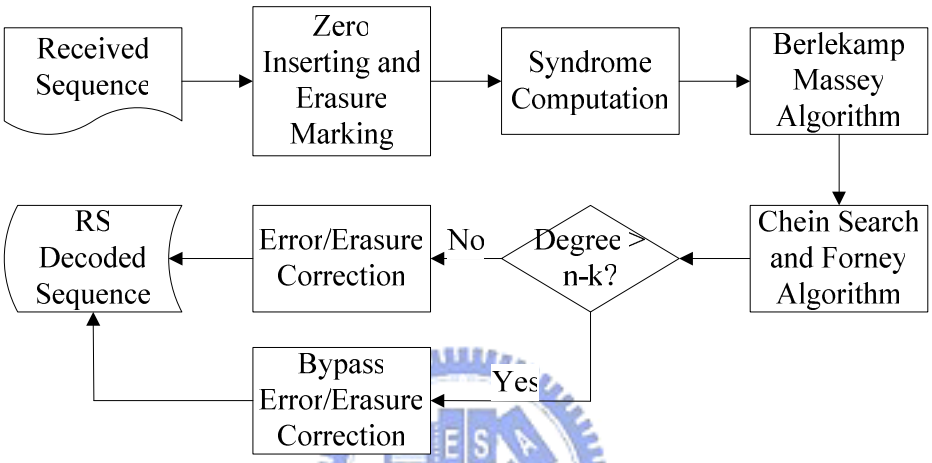


Figure 2.11: Block Diagram of the RS Decoder Program.

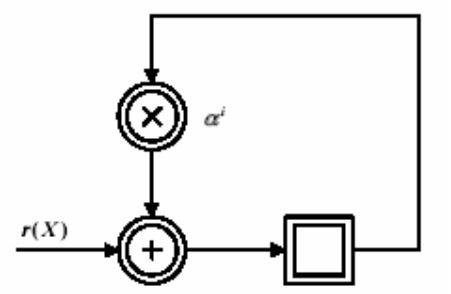


Figure 2.12: Syndrome Computation Circuit.

2.3.1.3 Galois Field Arithmetic

The major computational complexity of RS code is resulted from the galois field arithmetic architecture. In the case of GF(2^8), 8 bits by 8 bits operation are needed in galois field arithmetic when engaged in field element addition, multiplication or

inversion. The addition operation only requires the 8 bits by 8 bits XOR operation, but the multiplication and inversion are much more complicated than the addition and hence require a lot of computational time [3].

For the purpose of reducing the complexity of the galois field arithmetic (especially for the multiplication and inversion operations), several methods have been presented. For instance, By using Mastrovito algorithm [4], which introduce the concept of "product matrix", we can avoid degree 8 (Since the considered galois field is $GF(2^8)$) polynomial multiplication/division, or by exploiting the serial multiplier structure [5] which is commonly used in VLSI, we can simplify the polynomial multiplication/division to 64 bit multiplication and 7 polynomial reduction operation only. However, due to the special architecture of DSP, the computational complexity is still high for the two previous methods, which are mainly developed for VLSI architecture. In our case we finally employ the logarithmic table lookup algorithms [6] to handle the galois field arithmetic. Further explanation of the table lookup method and its profile respect to the computational speed are given in Chapter 4. Also we will discuss the difference between the two methods mentioned above and the table-lookup method in Chapter 4 for DSP optimization.

2.3.2 Convolutional Code

2.3.2.1 Encoding of Punctured Convolutional Code

The convolutional code encoding structure is shown in Fig. 2.6. It consists of one input bit, six memory elements (shift registers) and two output bits, which are generated by first performing AND operations on the generator polynomial coefficients, then pad the contents of the memory elements with the input bit, and then perform operation of modulo 2(XOR) on each bit generated by the previous AND operation. For the purpose to reduce computational complexity, we avoid performing XOR operation directly but

employing the table-lookup method to replace it. That is, we build a table that contains all possible 7 bit (6 memory element bits plus 1 input bit) XOR results and store them in memory. From the fact that the XOR operation is used frequently during the encoding process, we can just search the XOR results in the table and avoid the computations thus slightly speed up the encoding process.

According to the puncturing rule shown in Table 2.2, a "1" means a transmitted bit and a "0" means a skipped bit. The X and Y in the table denote the two output bits shown in Fig. 2.6. Note that the d_{free} has been changed from that of the original convolutional code with rate 1/2, which is equal to 10. The operations stated above are represented by a block diagram shown in Fig. 2.13. The input and output buffers shown in this figure are used for reducing the number of times on memory access when concerning DSP implementation. Since the convolutional encoder processes a piece of 1-bit input data each time step, if we do not setup buffers for input and output, we have to do memory accessing frequently during the encoding period, which decreases the processing rate on the TI DSP platform.

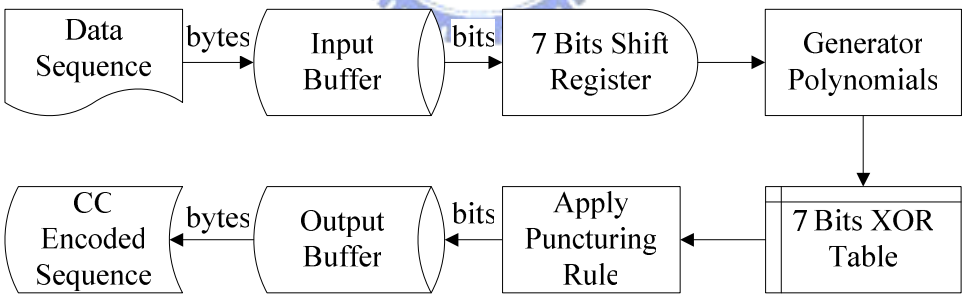


Figure 2.13: Block Diagram of the Convolutional Encoder Program.

2.3.2.2 Viterbi Decoding of Punctured Convolutional Code

Viterbi algorithm is the most well known technique in convolutional decoding process. The operation of Viterbi algorithm can be explained easily using the trellis diagram, which is generated by the encoder with all possible inputs. As we know, the

convolutional encoder consists of the memory elements, one input bit and two output bits. The output bits are decided by the suitable combinations (AND and XOR) of the past input bits. The changes of the value in the memory elements are viewed as the transition from one state to another. So we can model the encoder as a finite state machine, which is useful in the analysis of trellis diagram. An example of the finite state machine is shown in Fig. 2.14, whereas x(n-1) and x(n-2) denote the previous input and the input prior to the previous input, respectively. When we acquire a new input bit, the state of memory elements is changed and the finite state machine generates the corresponding output bits.

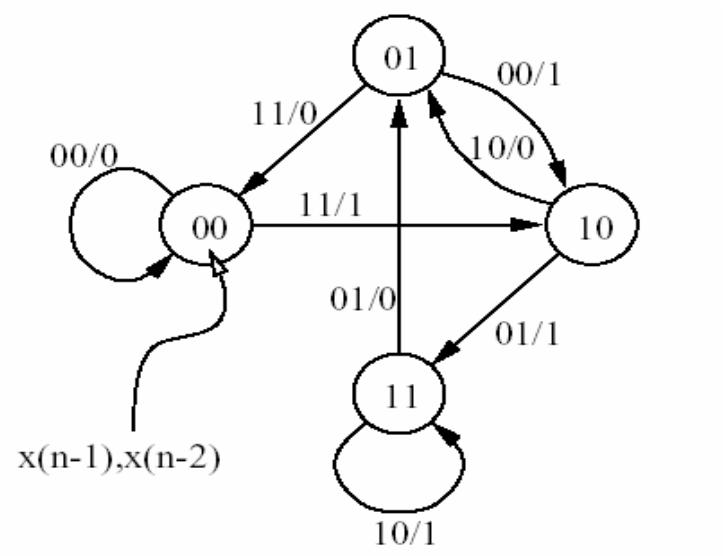


Figure 2.14: State Transition Diagram Example.

The trellis diagram can be derived from the state transition diagram. First, the finite state machine output is constructed by the given input and the current state. We expand the finite state machine to a trellis diagram by introducing the concept of time. The trellis diagram is consisting of all the features of finite state machine and can be viewed as the time axis expansion of the finite state machine diagram. A simple trellis diagram is shown in Fig. 2.15 as an example. We can easily see all the state transition for any possible input for every propagation time instance. In this trellis diagram, the upper

outgoing branch for each state corresponds to an input of 0, and the lower outgoing branch corresponds to an input of 1. Each state has two incoming and two outgoing branches. Each information sequence, uniquely encoded into an encoded sequence, corresponds to a unique path in the trellis. Equivalently, for a given path through the trellis, we can obtain the corresponding information sequence by reading off the input labels on all the branches that make up the path, and the procedure is also called "Traceback". The Viterbi algorithm is used to find the optimal path in the trellis diagram that results in the minimum errors. Then we do the traceback procedure to retrieve the information sequence, which has been the inputs to the encoder, and the details are discussed below.

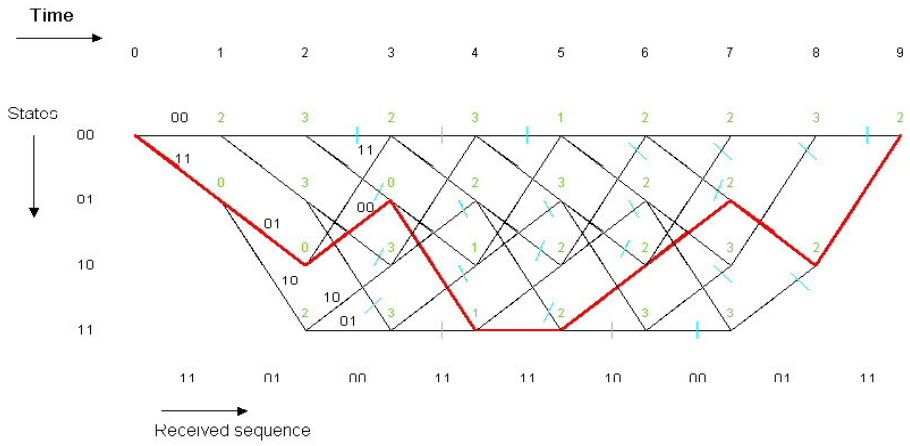


Figure 2.15: Trellis Diagram Example for a Viterbi Decoder.

The Viterbi algorithm computes the branch metric of each path at each stage of the trellis. The metric is first calculated and stored as a partial metric for each branch as the trellis traversed. Since there are two paths merge at each node, the path with a smaller metric is retained while the other is discarded. This is based on the principle that the optimum path must contain the sub-optimum survivor path just like as the one shown in Fig. 2.16 [7]. The survivor path for a given state at time instance n is the sequence of symbols closest to the received sequence up to time n. For the case of puncturing convolutional code, the metric associated with the punctured bits are simply disregarded

in metric calculation stage. The overall operation discussed in the above constitutes the computational core of the Viterbi algorithm and is so-called the Add-Compare-Select (ACS) operation.

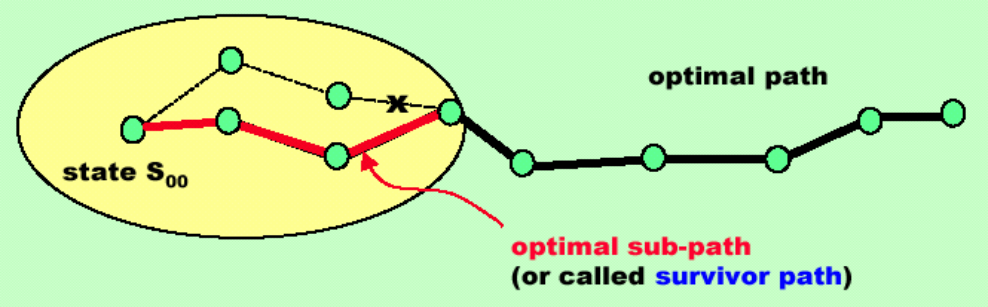


Figure 2.16: Survivor path of the Trellis Diagram.

In conclusion, the Viterbi algorithm can be divided into four major steps, the first step is the branch metric calculation and state metric loading, the second step is the ACS, the third step is the state metric storing and path recording, and the last one is the traceback. The block diagram of our Viterbi decoder program is shown in Fig. 2.17, and the structure of the Viterbi algorithm is shown in Fig. 2.18. The extend received sequence block shown in Fig. 2.17 is included for decoding the puncturing and tail-biting convolutional code and will be discussed later in this subsection.

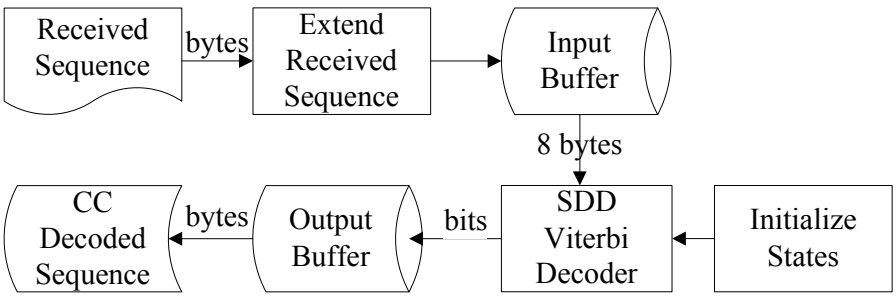


Figure 2.17: Block Diagram of the Viterbi Decoder Program.

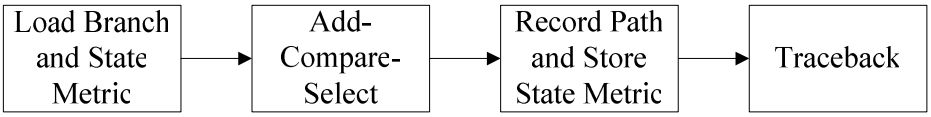


Figure 2.18: Structure of the Viterbi Algorithm.

Notice that we have named our Viterbi decoder in the block diagram as an SDD Viterbi decoder, where the SDD stands for Soft-Decision-Decoding. In fact, there are two kinds of decision types used in Viterbi decoding, one is called hard-decision, and another is called soft-decision. If hard-decision is adopted, then the metric value we used for calculating branch metric and state metric is the Hamming distance, which only counts the bit errors between each trellis path and the hard-limited output of the demodulator. For the case of soft-decision, the metric we used should be the Euclidean distance between each trellis path and the soft-output of the demodulator. The major difference on performance between these two decision types is the coding gain and the computational speed. For hard-decision, the calculation of Hamming distance is a simple XOR operation, On the other hand, the soft-decision in metric calculation requires a floating-point arithmetic. The hard-decision based Viterbi decoder is much faster than the soft-decision based algorithm. However, its coding gain will lose 2 to 3 dB compared to soft-decision decoding, and cannot satisfy the requirements of IEEE 802.16a standard [8]. Hence, the soft-decision decoding is adopted to implement our Viterbi decoder.

2.3.2.3 Bit Interleaved Soft Decision Viterbi Decoding

In the specific FEC scheme defined by IEEE 802.16a, there is a block interleaver between the convolutional code and modulator. Therefore, the optimal SDD should take the joint trellis structure which consists of the convolutional code, the block interleaver and the modulator into account. In consequence, it leads to a complicated solution to be realized in practice. To be more practical, we consider a suboptimal solution based on a

bit-by-bit metric mapping and calculation concept, which is proposed in [9]. To begin with, we can generalize our major problems to how to obtain the metric values used in the SDD Viterbi decoder while concerning the de-interleaving process. Here we are not going to discuss or prove the detailed algorithm that has already been well-defined in [9], but just showing the procedure on acquiring metric values.

According to the suboptimal solution, we first calculate the Euclidean distance between the received symbol and its nearest reference modulated symbol with respect to a decided bit "0" and "1". Let us take 16-QAM modulation as example. Referring to Fig. 2.19, if a received symbol lies in the coordinate (2.5, 2.7) (represented by a square point in the figure), then its branch metric of the first bit with respect to a decided bit "0" should be the Euclidean distance between the received symbol and the rightmost reference symbol whose in-phase coordinate is 3 and the result is $|3 - 2.5|^2 = 0.25$. And the branch metric with respect to a decided bit "1" should be $|-1 - 2.5|^2 = 12.25$. The branch metric of the second bit, third bit, and fourth bit of this received symbol can be calculated in a similar way. Consequently, we have four pairs of branch metric for each received symbol. Before sending them to the SDD Viterbi decoder, these pairs of branch metric should be mapped to the corresponding bit position since the original convolutional encoded sequence has been interleaved. In order to be consistent with the newly defined branch metrics, our SDD Viterbi decoder should be modified to be able to treat these de-interleaved (or to say "demapped", alternatively) branch metric as the input data sequence instead of the soft-demodulated symbol. Except for the branch metric calculation step, all the other parts in a conventional SDD Viterbi decoder are still the same.

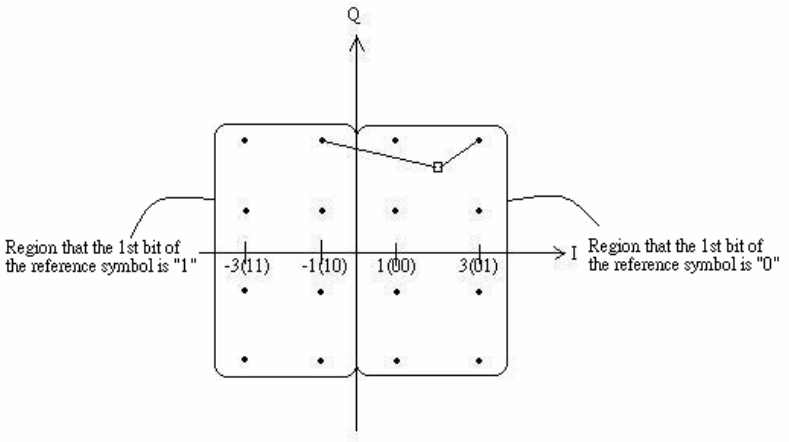


Figure 2.19: Partition of the 16-QAM Constellation.

2.3.2.4 Viterbi Decoding of Tail-Biting Convolutional Code

According to [8] and [10], the practical suboptimal tail-biting Viterbi decoder is shown in Fig. 2.20, where the "SDD Viterbi Decoder" block denotes the Viterbi decoder with puncturing mechanism and bit-interleaved SDD. The parameter α and β are both chosen to be 24 to achieve the balance of computational complexity and the performance of error correction based on the analysis done in [8].

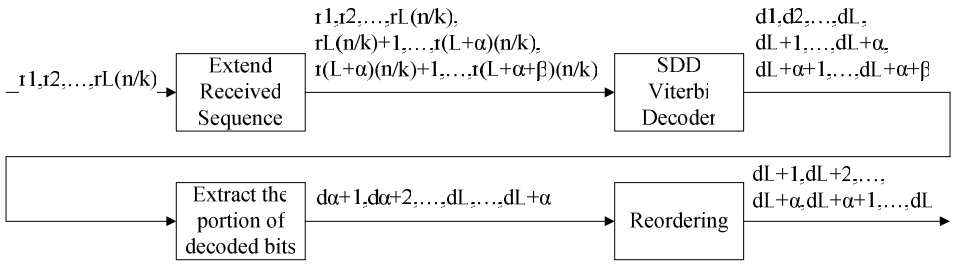


Figure 2.20: Block Diagram of the Suboptimal Tail-Biting Viterbi Decoder.

2.3.2.5 The Butterfly Structure in the Trellis Diagram

In order to reduce the computational complexity in the ACS part, we bring in the concept of butterfly structure from the trellis diagram. The Symmetry in the trellis diagram, which forms the butterfly structure, can be used to reduce the number of branch metric calculations. Fig. 2.21 shows the butterfly structure associated with the Viterbi decoder — pairing new states 2i and 2i+1 with previous states i and i+s/2, where s is the number of total possible states. In our case of constraint length K=7, s equals 64 (2^6). Even though there are four incoming branches, there are only two different branch costs.

Path metrics for each new state are calculated using each incoming branch cost plus the previous path cost associated with that branch. The maximum of the two incoming path metrics is selected as the survivor. The butterfly computations consist of two "Add-Compare-Select" (ACS) operations and updating the survivor path history. The two ACS operations are:

$$S_n(2i) = \mathbf{min}\ \{S_{n-1}(i) + b\ ,\ S_{n-1}(i+s/2) + a\},\ \text{and}$$

$$S_n(2i+1) = \mathbf{min}\ \{S_{n-1}(i) + a\ ,\ S_{n-1}(i+s/2) + b\}$$

After completing N stages of decoding, one of the M survivor paths is selected for trace-back. Obviously, the number of branch metric calculation has been reduced greatly by introducing the butterfly structure.

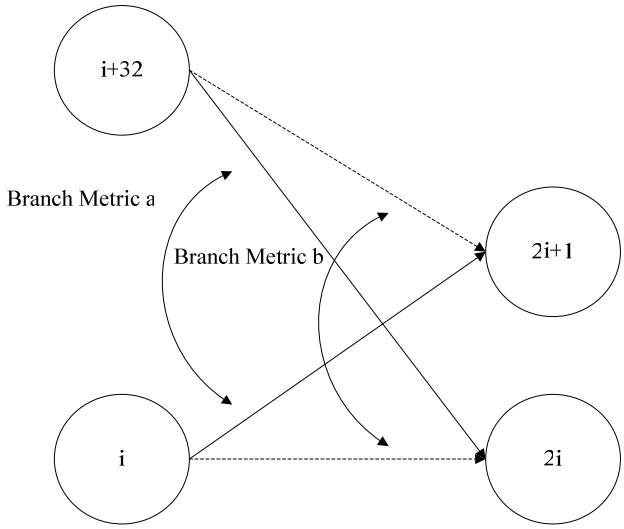


Figure 2.21: Butterfly Structure Showing Branch Cost Symmetry.

Chapter 3

DSP Implementation Environment

In our IEEE 802.16a OFDMA project, for the ease to link up all the subprojects, we choose the digital signal processor (DSP) platform to implement the whole system. The DSP baseboard we use is Innovative Integration's (II's) new product in year 2003 named Quixote, which houses the Texas Instruments TMS320C6416 DSP chip. In this chapter, in addition to an introduction to the DSP chip and the DSP baseboard, the data communication process between the host PC and the target DSP is also described.

3.1 The TI DSP Chip

The DSP chip we adopt is one of the TMS320C64x series DSP. According to [11], TMS320C64x series is a member of the TMS320C6000 (C6x) family. The C6000 device is capable of executing up to eight 32-bit instructions per cycle and its core CPU consists of 64 general-purpose 32-bit registers (for C64x only) and eight functional units. The detailed features of the C6000 family devices include:

- Advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic units.
- Instruction packing (Reduce Code Size).
- Conditional execution of all instructions.
- Efficient code execution on independent functional units.

- 8/16/32-bit data support, providing efficient memory support for a variety of applications.
- 40-bit arithmetic options add extra precision for computationally intensive applications.
- Saturation and normalization provide support for key arithmetic operations.
- Field manipulation and instruction extract, set, clear, and bit counting support common operation found in control and data manipulation applications.

The block diagram of the C6000 family is shown in Fig. 3.1. The C6000 devices come with program memory, which, on some devices, can be used as a program cache. The devices also have varying sizes of data memory. Peripherals such as a direct memory access (DMA) controller, power-down logic, and external memory interface (EMIF) usually come with the CPU, while peripherals such as serial ports and host ports are available only for certain models.

In the following subsections, the TMS320C64x DSP Chip is introduced in the three major parts: Central processing unit (CPU), Memory, and Peripherals.

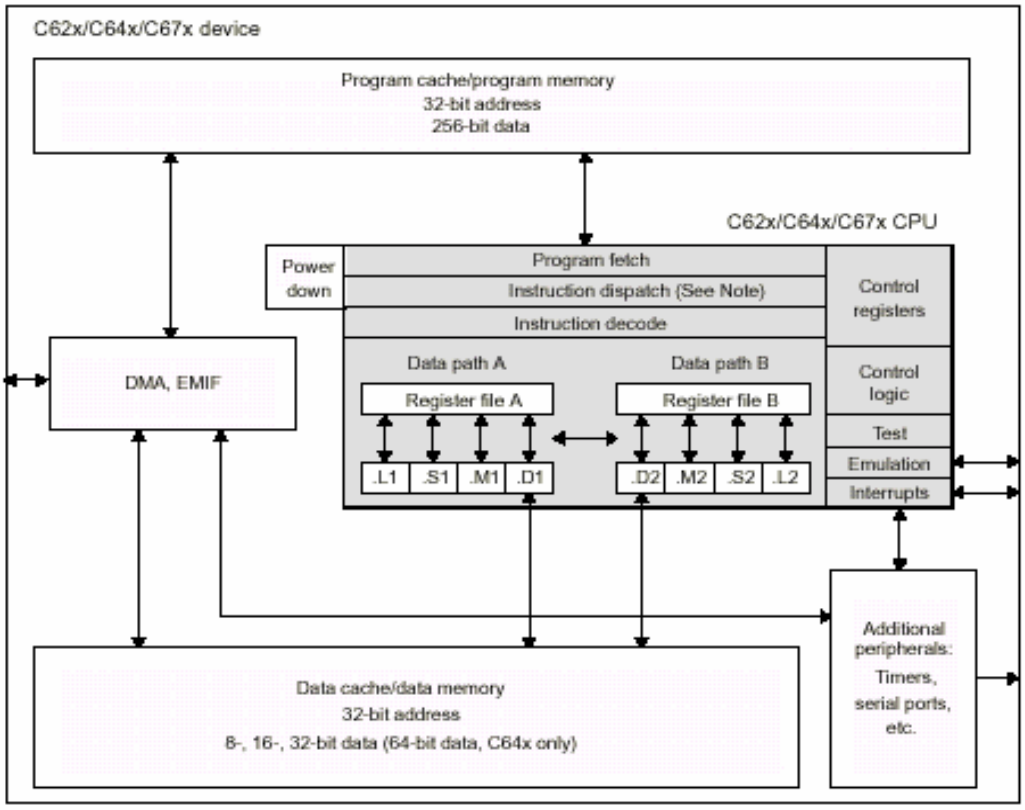


Figure 3.1: The Block Diagram of TMS320C6x DSP Chip.

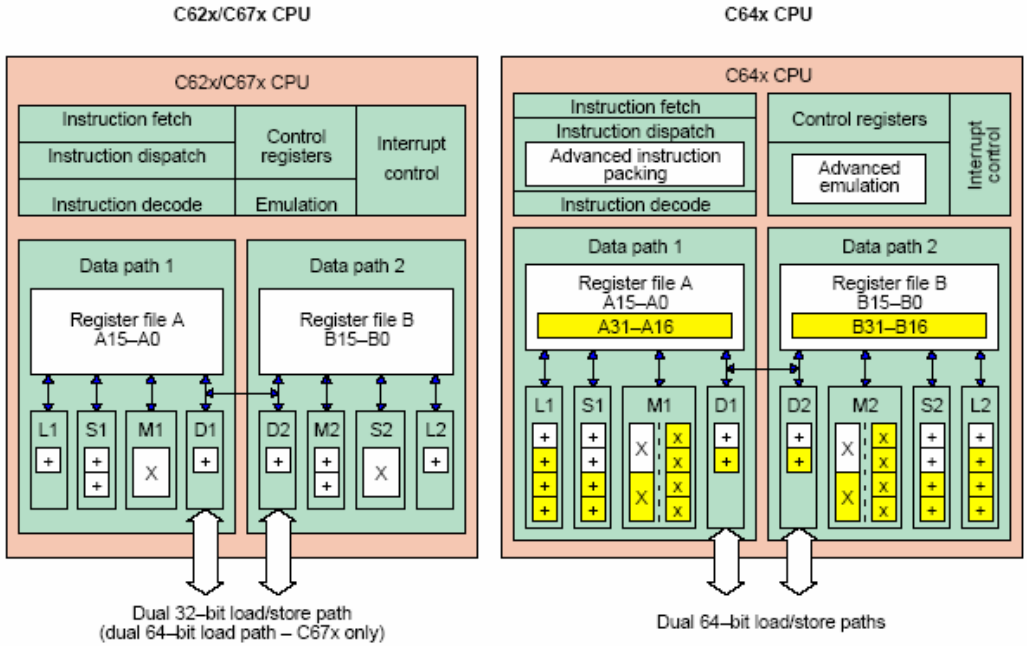


Figure 3.2: The TMS320C64x DSP Chip Architecture and Comparison with Ancient TMS320C62x/C67x Chip.

3.1.1 Central Processing Unit

Besides the eight independent functional units and sixty-four general purpose registers that has been mentioned before, the C64x CPU also consists of the program fetch unit, instruction dispatch unit (attached with advanced instruction packing), instruction decode unit, two data path (A and B, each with four functional units), test unit, emulation unit, interrupt logic, several control registers and two register files (A and B with respect to the two data paths). The architecture is illustrated in more detail in Fig .3.2 [12]. Compared with the other C6000 family DSP chip, the C64x DSP chip provides more available hardware resources. The additional features that are only available on C64x are:

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

The program fetch unit shown in the figure could fetch eight 32-bit instructions (which implies 256-bit wide program data bus) every single cycle, and the instruction dispatch and decode units could also decode and arrange the eight instructions to eight functional units. The eight functional units in the C64x architecture could be further divided into two data paths A and B as shown in Fig. 3.2. Each path has one unit for multiplication operations (.M), one for logical and arithmetic operations (.L), one for branch, bit manipulation, and arithmetic operations (.S), and one for loading/storing, address calculation and arithmetic operations (.D). The .S and .L units are for arithmetic,

logical, and branch instructions. All data transfers make use of the .D units. Two cross-paths (1x and 2x) allow functional units from one data path to access a 32-bit operand from the register file on the opposite side. There can be a maximum of two cross-path source reads per cycle. There are 32 general purpose registers, but some of them are reserved for specific addressing or are used for conditional instructions.

Most of the buses in the CPU support 32-bit operands, and some of them support 40-bit operands. Each functional unit has its own 32-bit write port into a general-purpose register file. All functional units which end in 1 (for example, .L1) write to register file A while all functional units which end in 2 (for example, .L2) write to register file B. There is an extra 8-bit wide port for 40-bit write as well as an extra 8-bit wide input port for 40-bit read in four specific units (.L1, .L2, .S1 and .S2). Since each unit has its own 32-bit write port, all eight functional units could be operated in parallel in every single cycle.

The program pipelining is also an important technique to make instructions execute in parallel and hence reduce the overall execution cycles. In order to make pipelining work properly, we should have knowledge of the pipeline stages and instruction execution phases. Since the program pipelining is highly related to the optimization of DSP program, we left it to be discussed in next chapter and not go into detail here.

3.1.2 Memory

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

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

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

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

3.1.3 Peripherals

In addition to the on-chip memory, the TMS320C64x DSP chips also contain peripherals for supporting with off-chip memory options, co-processors, host processors, and serial devices. The peripherals are direct memory access (DMA) controller, Host-Port Interface (HPI), EMIF, Timers and some other units.

The DMA controller transfers data between regions in the memory map without the intervention by CPU. It could move the data from internal memory to external memory or from internal peripherals to external devices. It is used for communication to other devices.

The Host-Port Interface (HPI) is a 16-bir wide parallel port through which a host processor could directly access the CPUs memory space. It is used for communication between the host PC and the target DSP.

The C64x has two 32-bit general-purpose timers that are used to time events, count events, generate pulses, interrupt the CPU and send synchronization events to the DMA controller. The timer has two signaling modes and could be clocked by an internal or an external source.

3.2 The DSP Baseboard

The Quixote DSP Baseboard card is shown in Fig. 3.3 and the architecture is shown in Fig. 3.4 [15]. Quixote consists of a TMS320C6416 600 MHz 32-bit fixed-point DSP chip and a Xilinx two- or six-million gate Virtex-II FPGA in a single board. Utilizing the signal processing technology to provide processing flexibility, efficiency and deliver high performance. Quixote has 32MBytes SDRAM for use by DSP and 4 or 8Mbytes zero bus turnaround (ZBT) SBSRAM for use by FPGA. Developers could build complicated signal processing systems by integrating these reusable logic designs with their specific application logic.

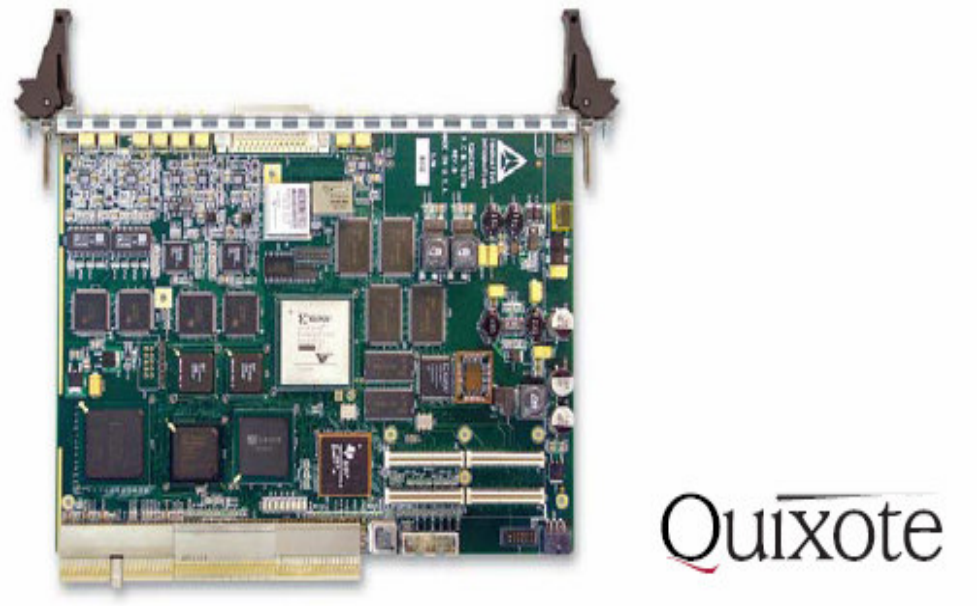


Figure 3.3: Innovative Integration's Quixote DSP Baseboard Card.

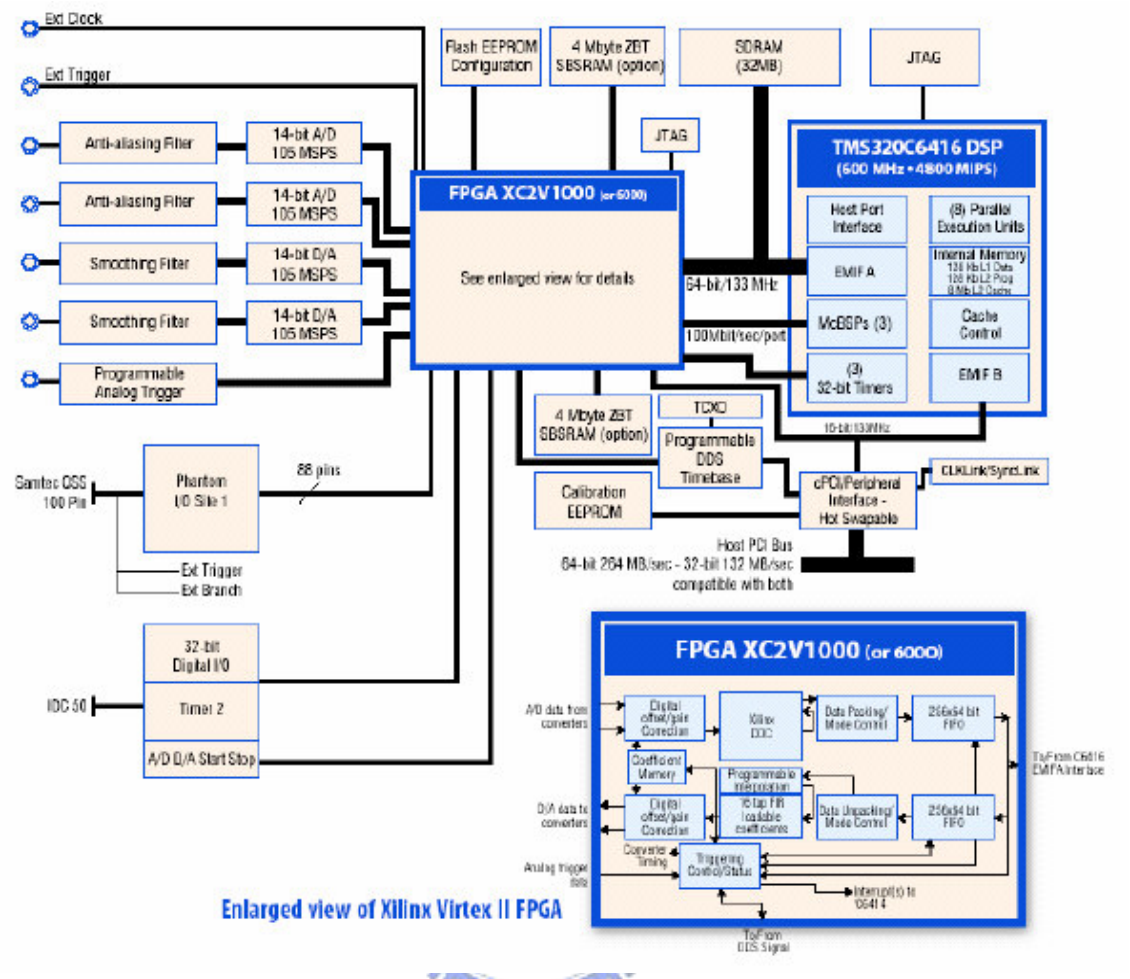


Figure 3.4: The Architecture of Quixote Baseboard.

3.3 Data Communication Mechanism

Many applications of the Quixote baseboards involve communication with the host CPU in some manner. All applications at a minimum must be reset and downloaded from the host, even if they are isolated from the host after that. The simplest communication method supported is a mapping of Standard C++ I/O to the Uniterminal applet that allows console-type I/O on the host. This allows simple data input and control and sending text strings to the user. The next level of support is provided by the Packetized Message Interface. This allows more complicated medium rate transfer of commands and information between the host and target. It requires more software support on the host than the Standard I/O. For full rate data transfers Quixote supports

the creation of data streaming to the host, for the maximum ability to move data between the target and host. On Quixote baseboards, a second type of busmaster communication between target and host is available for use, it is the CPU Busmaster interface.

The primary CPU busmaster interface is based on a streaming model, where logically data is an infinite stream between the source and destination. This model is more efficient because the signaling between the two parties in the transfer can be kept to a minimum and transfers can be buffered for maximum throughput. In addition, the Busmaster streaming interface is fully handshook, so that no data loss can occur in the process of streaming. For example, if the application cannot process blocks fast enough, the buffers will fill, then the busmaster region will fill, then busmastering will stop until the application resumes processing. When the busmaster stops, the DSP will no longer be able to add data to the PCI interface FIFO.

However, in the application of FEC encoder and decoder, the data sequence is first divided into RS blocks then performed encoding and decoding procedure. Hence the continuous streaming may not be suitable for FEC application. Alternatively, there is a data flow paradigm supported for non-continuous data sequence called block mode streaming. For very high rate applications, any processing done to each point may result in a reduction in the maximum data rate that can be achieved. Since block mode does no implicit processing on a point-by-point basis, the fastest data rates are achievable using this mode.

The DSP Streaming interface is bi-directional. Two streams can run simultaneously, one running from the analog peripherals through the DSP into the application. This is called the “Incoming Stream”. The other stream runs out to the analog peripherals. This is the “Outgoing Stream”. In both cases, the DSP needs to act as a mediator, since there is no direct access to analog peripherals from the host. The block diagram of the DSP streaming mode is shown in Fig. 3.5 [15].

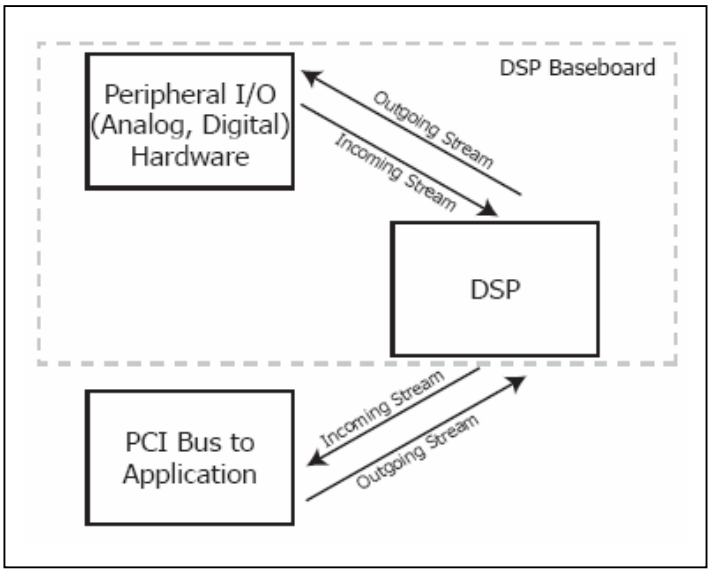


Figure 3.5: Block Diagram of DSP Streaming Mode.

DSP Streaming is initiated and started by the Host, using the Caliente component. On the target, the DSP interface uses a pair of DSP/BIOS Device Drivers, PciIn (on the Outgoing Stream) and PciOut (on the Incoming Stream), provided in the Pismo peripheral libraries for the DSP. They use burst-mode and are capable of copying blocks of data between target SDRAM and host bus-master memory via the PCI interface at instantaneous rates up 264 MBytes/sec.

In addition to the busmaster streaming interface, the DSP and the host also have a lower bandwidth communications link called packetized message interface for sending commands or side information between the host PC and the target DSP.

Chapter 4

Implementation and Optimization of 802.16a FEC Scheme on DSP Platform

As mentioned in last chapter, we adopt the Texas Instruments (TI) digital signal processor (DSP) for implementing the Forward Error Correction (FEC) scheme in the IEEE 802.16a wireless communication standard. In this chapter, we are going to discuss the main themes of this thesis – the implementation and optimization of the specified FEC scheme on the newly released II's Quixote DSP baseboard, which houses a TI TMS320C6416 DSP chip. We firstly briefly introduce the entire system structure of our FEC implementation and its communication mechanism. Secondly, we introduce some special features of TI C6000 family DSP that is helpful when doing compiler level optimization. Then, we proposed some simple and yet practically useful techniques for improving the computational speed of Reed-Solomon (RS) Code and Convolutional (CC) Code (mainly for decoder part) on TI C64 family DSP. Finally, we present the improvement after the efforts we made on the RS code and the CC code optimization by showing the simulation profile generated by the TI Code Composer Studio (CCS) built-in profiler.

4.1 System Structure of the FEC Implementation

As defined in the IEEE 802.16a standard, the FEC scheme, which consists of FEC encoder and FEC decoder, is located between the source encoder/decoder and the channel modulator/demodulator. Due to the features of the FEC scheme, it requires massive computation in the decoding procedure. Thus, for the purpose of achieving the real-time processing goal, it is necessary to assign a single DSP board for the FEC use only. In consequence, the FEC scheme, source coding scheme and channel modulation scheme each uses an individual DSP board and linked by the PCI port on the personal computer (PC), one thing to be noted here is the communication mechanism between the DSP board and the Host PC. It supposed to be the data streaming mode that has been described in Chapter 3, but due to the malfunction of the newly released II's Quixote DSP baseboard, up till now, the streaming mode on the DSP board is not yet work. In substitution, we use the standard I/O (fread and fwrite) to implement the DSP file I/O mechanism on the TMS320C6416 Simulator. The drawback of using the standard I/O is that it cannot proceed too many input data or the processor may crash during the I/O time and it takes extra cycles to perform the file I/O mechanism.

The system structure is shown in Fig. 4.1 and Fig. 4.2 for the transmitter side and the receiver side, respectively. At the transmitter side, the source coded data sequence is first multiplexed by an audio/video multiplexer then transmits to the randomizer of the FEC encoding scheme through the PCI interface of Host PC. Afterward, the sequence is processed by the randomizer, the RS encoder, the CC encoder and the block interleaver and then the interleaved coded sequence is transmitted to the channel modulator through the PCI interface. At the receiver side, the procedure is an reverse of that in the transmitter side. First the demodulator transmits the soft decision demodulated metric sequence to the FEC decoding scheme (again through the PCI interface). After FEC decoding, the decoded sequence is passed to the source decoder through the PCI interface and then the source decoding operation is performed.

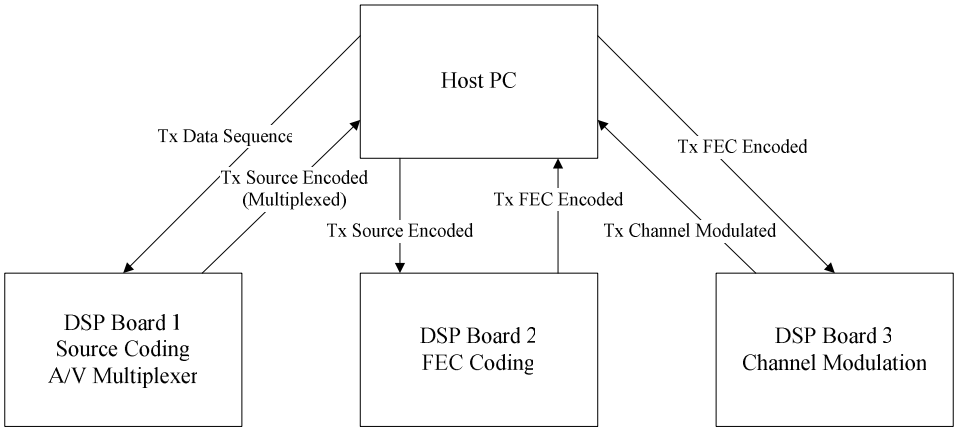


Figure 4.1: System Structure of Transmitter Side.

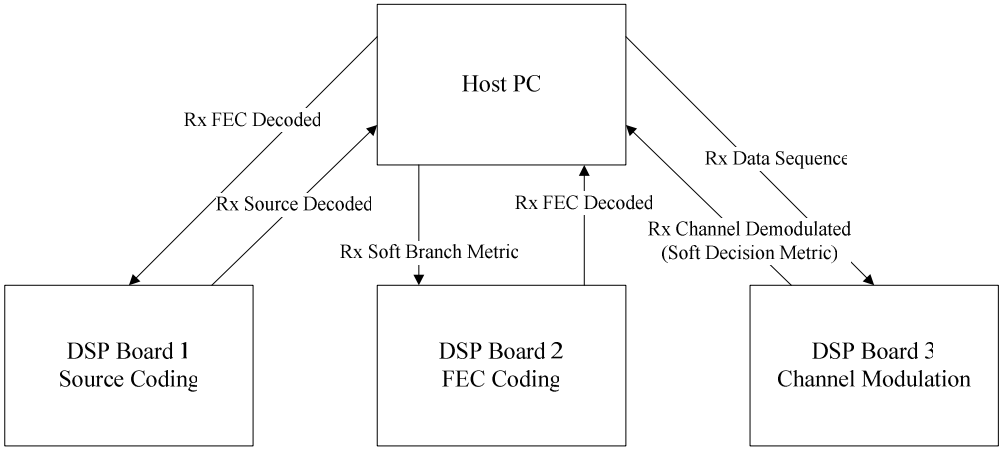


Figure 4.2: System Structure of Receiver Side.

4.2 Compiler Level Optimization Techniques

In this subsection, firstly the TI C6000 family pipeline structure is introduced for understanding how the processor arranges the pipeline stages and what instructions are more time consuming and shall be avoided if possible. Secondly, the code development flow is presented to show how to develop a DSP program efficiently and systematically. Thirdly, an important techniques used by the TI CCS compiler to improve the program speed, so-called “software pipelining”, is introduced and a simple example is given to

explain how we can improve the program efficiency by using the software pipelining technique.

4.2.1 Pipeline Structure of the TI C6000 Family

There are a few features regarding to the TI C6000 family's pipeline structure that can provide the advantages of good performance, low cost, and simple programming. The following are several useful features [11]:

- *Increased pipelining eliminates traditional architectural bottlenecks in program fetch, data access, and multiply operation.*
- *Pipeline control is simplified by eliminating pipeline locks.*
- *The pipeline can dispatch eight parallel instructions in every cycle.*
- *Parallel instructions proceed simultaneously through the same pipeline phase.*

The pipeline structure of the C6000 family consists of three basic pipeline stages, They are Fetch stage (F), Decode stage (D), and Execution stage (E). At the **F** stage, the CPU first generates an address, fetches the opcode of the specified instruction from memory, and then passes it to the program decoder. At the **D** stage, the program decoder efficiently routes the opcode to the specific functional unit determined by the type of instruction (LDW, ADD, SHR, MPY, etc). Once the instruction reaches the **E** stage, it is executed by its specified functional unit. Most instructions of the C6000 family fall in the Instruction-Single-Cycle (ISC) category, such as ADD, SHR, AND, OR, XOR, etc. However, the results of a few instructions are delayed. For example, the multiply instructions - MPY (and its varieties) requires a delay length equal to one cycle.

One cycle delay means that the execution result will not be available until one cycle later (i.e., not available for the next instruction to use). The results of a load instruction – LDW (and its varieties) are delayed for 4 cycles. Branches instructions

reach their target destination 5 cycles later. Store instructions are viewed as an ISC from the CPU's perspective because of the fact that there is no execution phase required for a store instruction but actually it still finishes in 2 cycles later. Since the maximum delay among all the available instructions is 5 cycles (6 execution cycles totally), it is intuitive to split the execution stage (E) into six phases as shown in Table 4.1.

Execution Phases (Completing Phase)	***Instructions' Category***
E1	Instruction single cycle
E2	Multiply and its varieties
E3	Store and its varieties
E4	
E5	Load and its varieties
E6	Branch to destination

Table 4.1: Completing Phase of Different Type Instructions.

4.2.2 Code Development Flow

Traditional development flows in DSP industry have involved validating a C model for correctness on a host PC or Unix workstation and then painstakingly porting that C code to hand-coded DSP assembly language. This is both time consuming and error prone. The recommended code development flow involves utilizing the C6000 code generation tools to help in optimization rather than forcing the programmer to code by hand in assembly. These advantages allow the compiler to do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation. Fig. 4.3 illustrates the three phases in the code development flow [13]. Because phase 3 is kind of 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 very poor or the resource allocation is very unbalanced.

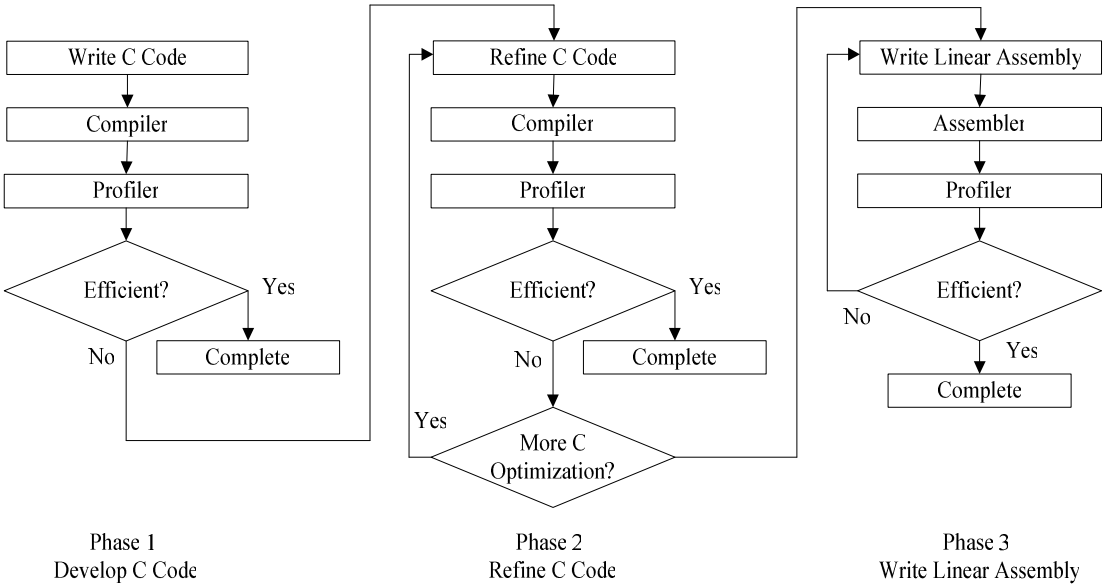


Figure 4.3: Code Development Flow.

4.2.3 Software Pipelining

Software pipelining is extensively used to exploit instruction level parallelism (ILP) in loops and TI CCS compiler is also capable of doing it. To say more clearly, it is a technique used to schedule instructions in a loop so that multiple iterations of the loop can execute in parallel, so the empty slots can be filled and the functional units can be used more efficiently. Overall it makes a loop to be a highly optimized loop code and hence accelerate the program execution speed significantly.

For the ease of understanding how software pipelining actually works, here we give an example to illustrate [16]. A simple *for* loop and its code after applying software pipelining are shown in Fig 4.4(a) and 4.4(b). The loop schedule length is reduced from four control steps to one control step in the software pipelined loop. However, the code size of software pipelined loop is three times longer than that of the original code in this

example. Fig. 4.5(a) and 4.5(b) show the execution records of the original loop and the software pipelined loop, respectively.

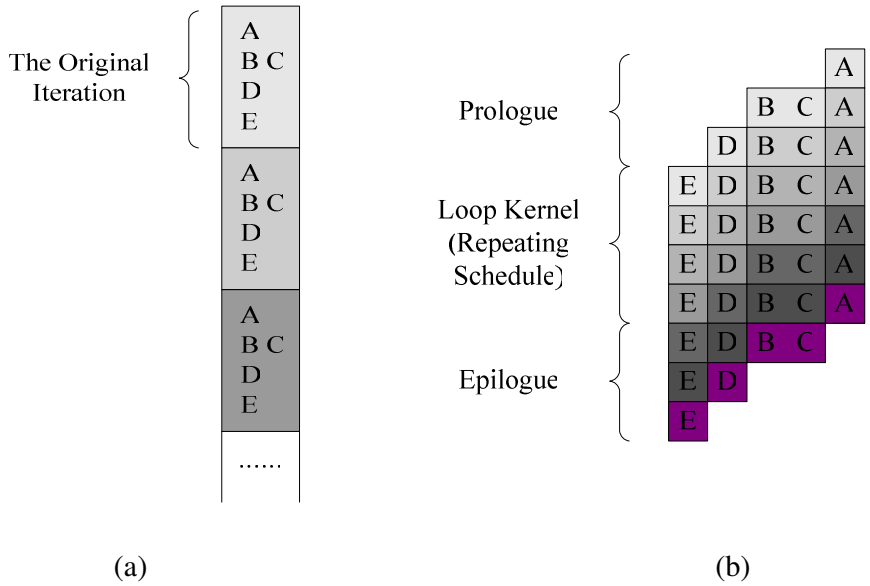
```
for i = 1 to n do
    A[i] = E[i-4] + 9;
    B[i] = A[i] * 5;
    C[i] = A[i] + B[i-2];
    D[i] = A[i] * C[i];
    E[i] = D[i] + 30;
end
```

(a)

```
A[1] = E[-3] + 9;
A[2] = E[-2] + 9;
B[1] = A[1] * 5'
C[1] = A[1] + B[-1];
A[3] = E[-1] + 9;
B[2] = A[2] * 5;
C[2] = A[2] + B[0];
D[1] = A[1] * C[1];
for i = 1 to n-3 do
    A[i+3] = E[i-1] + 9;
    B[i+2] = A[i+2] * 5;
    C[i+2] = A[i+2] + B[i];
    D[i+1] = A[i+1] * C[i+1];
    E[i] = D[i] + 30;
End
E[n] = D[n] +30;
D[n] = A[n] * C[n];
E[n-1] = D[n-1] + 30;
B[n] = A[n] * 5;
C[n] = A[n] + B[n-2];
D[n-1] = A[n-1] * C[n-1];
E[n-2] = D[n-2] + 30;
```

(b)

Figure 4.4: (a) The Original Loop. (b) The Loop After Applying Software Pipelining.



(a) (b)

Figure 4.5: (a) Execution Record of the Original Loop. (b) Execution Record of the Software Pipelined Loop.

In these figures, we can clearly observe that there are only two (B and C) of the five instructions – A,B,C,D,E executed in parallel in the original loop, while there are all five instructions executed in parallel in the software pipelined loop and hence the program efficiency is improved significantly. We can also notice that the pipelined code can be classified into three regions: prologue, loop kernel (repeating schedule) and epilogue. The prologue is the "setup" to the loop. Running the prologue code is often called "priming" the loop. The length of the prologue depends on the latency between the beginning and ending of the loop code; i.e., the number of instruction and their latency. The epilogue refers to the ending instructions, which must be completed at the end after the loop kernel; it is kind of similar to the prologue and is optional. If necessary, it can be rolled into the loop kernel. Prologue and epilogue of the software pipelined loop occupy a large part of the code size, so there may be a trade-off issue between the speed and memory size consideration that we have to take into account. But since the program memory of the Quixote DSP baseboard is quite large and the original FEC code size is quite small, it may not be a serious issue if we adopt software pipelining in our codes.

Concerning the implementation using the TI C6000 DSP family, the C code loop performance is greatly influenced by how well the CCS compiler can do the software pipelining on our loop. The compiler provides some feedback information to the programmers to fine-tune the loop structure. Understanding the feedback information, we can quickly tune our C codes to obtain the highest possible performance. The feedback is geared for explaining exactly what all the issues related to pipelining the loop are and what the results are. The compiler goes through three basic stages when compiling a loop, these stages are [13]：

1. Qualify the loop for software pipelining.
2. Collect loop resource and dependency graph information.
3. Software pipelining the loop.

In the first stage, the compiler tries to identify what the loop counter (named trip counter because of the number of trips through a loop) is and any information about the loop counter such as minimum value (known minimum trip count), and whether it is a multiple of something (has a known maximum trip count factor).

If the above information is known about a loop counter, the compiler can be more aggressive with performing packed data processing and loop unrolling optimizations. For example, if the exact value of a loop counter is not known but it is known that the value is a multiple of some number, the compiler may be able to unroll the loop to improve the performance.

There are several conditions that must be met before software pipelining is allowed, or legal, from the compiler's point of view. These conditions are：

- It cannot have too many instructions in the loop. Loops that are too big typically require more registers than that are available and they require a longer compilation time.
- It cannot call another function within the loop unless the called function is inlined. Any break in control flow makes it impossible to software pipeline as multiple iterations are executing in parallel.

If any of the conditions for software pipelining are not met, qualification of the pipeline will halt and a disqualification messages will appear. In this situation, software pipelining will not be applied to our loop program and hence the program operating speed will be quite slow.

In the second stage, the compiler is collecting loop resource and dependency graph information. It will derive the loop carried dependency bound, unpartitioned resource bound across all resources, partitioned resource bound across all resources based on our loop code. It shows the resource partition table, which summarizes how the instructions have been assigned to the various machine resources and how they have been partitioned between the A and B side, after it has the information about the three bounds.

In the third stage, the compiler attempts to software pipeline our loop based on the knowledge it collects from the previous two stages. The first thing the compiler attempts to do during this stage, is to schedule the loop at an iteration interval (ii) equal to the minimum value of the three bounds obtained in second stage. If the attempt was not successful, the compiler provides additional feedback message to help explain why it failed; i.e., register is live too long or did not find a schedule. The compiler will keep proceeding to ii = (previous failed ii + 1) till it find a valid schedule and then the software pipeline is done.

4.3 Optimization on Reed-Solomon Code

Follow the code development flow described in section 4.2.2, before actually revising our program code, we should first generate a profile by using the CCS built-in profiler to know the exact execution cycles. Then, we can identify which part of our program consumes the most execution time based on the profile data, and hence we can concentrate on this part to make the whole program faster. In the following subsections, the optimization of our RS code program on TI DSP platform is divided into the encoder and the decoder parts to be discussed.

4.3.1 Optimization on RS Encoder

4.3.1.1 Choose Appropriate Data Types

Table 4.2 shows the original (before optimization) profile of the RS encoder, the first shadowed function, GF_Multiply, is the galois field multiplier used by the RS_Encode function to encode the data sequence into RS blocks. The last two shadowed functions, Int_to_Vec and Vec_to_Int, are used by the GF_Multiply function to convert the integer containing 8-bit galois field element to a vector containing each

bit of the integer, and convert the vector back to the original integer, respectively. The TI Programmer's Guide [13] reminds us that the size of data type is different between the DSP platform and the PC platform. For example, the "int" and "long" data type is of the same size in PC platform which equal to 32-bit, but the "long" data type in DSP platform is of larger size, which equals to 40-bit. If we just port our C program to DSP using the CCS compiler without any checking on the data type, it may result in larger variables and data arrays in our program if our program assumed that "int" and "long" data types are the same size. Furthermore, using the "long" data type will result in a worse execution efficiency since it requires extra instructions to be generated and limits functional unit selection. The "long" data type value needs two registers – one 32-bit register plus 8-bit LSB in another specific 32-bit register. If the registers used in the program are full, we need to store the register contents into the stack and load them back after the "long" type data are computed completely. That is, we waste time and memory space because of the load/store operations, which are the most time consuming instructions.

After examining our program code carefully, we find it does not affect the correctness of our program if we replace the "long" data type by the "int" data type, and it will result in a significantly improvement on the program executing speed. Fig. 4.6 shows the pseudo assembly code for variable using "long" data type and "int" data type. Obviously, compared to the variable with "int" data type, the one with "long" data type needs extra assembly instructions to execute the same C instruction.

Areas	Code Size	Cycles	Percentage (%)	Processing Rate (Kbits/sec)
Main Function	1164	1433434	100	120
RS_Encode	356	1430005	99	
GF_Multiply	964	1348940	94	
Int_to_Vec	220	179728	13	
Vec_to_Int	224	91776	7	

Table 4.2: Original Profile of RS Encoder.

The profile of modified RS encoder is shown in Table 4.3. We can find that not only the processing rate, but also the code size are improved by this modification.

Areas	Code Size	Cycles	Percentage (%)	Processing Rate (Kbits/sec)
Main Function	1160	1265024	100	137
RS_Encode	236	1261595	99	
GF_Multiply	584	1210168	96	
Int_to_Vec	148	87952	7	
Vec_to_Int	156	76480	6	

Table 4.3: Profile of Revised RS Encoder (Data Type Modification).

```
/* before  modification */
;-----------------------------------------------
;long feedback;
;feedback = data[i] ^ bb[0];
;bb[j] = bb[j+1] ^ gf_mul_tab(Gg_poly
[no_p-1-j],feedback);
;-----------------------------------------------
       STW    .D2T1   A15,*SP--(40)
||     MV     .D1X    SP,A31

       STDW   .D1T1   A11:A10,*-A31(32)
||     STDW   .D2T2   B11:B10,*+SP(32)

       STDW   .D1T1   A13:A12,*-A31(24)
||     STW    .D2T2   B12,*+SP(28)

       STW    .D1T1   A14,*-A31(36)
||     STW    .D2T2   B3,*+SP(24)


       MV     .D1     A6,A14
||     MV     .S1X    B4,A10
```

```
/* after  modification */
;-----------------------------------------------
;int  feedback;
;feedback = data[i] ^ bb[0];
;bb[j] = bb[j+1] ^ gf_mul_tab(Gg_poly
[no_p-1-j],feedback);
;-----------------------------------------------
       STW    .D2T2   B3,*SP--(8)

       MV     .D1X    B4,A7
||     MV     .S1     A6,A8
```

Figure 4.6: Pseudo Code for Variable Using Long and Int Data Type.

4.3.1.2 Galois Field Multiplication

Refer to Table 4.3 shows the profile of our RS encoder program after the data type modification. As expected in Chapter 2, we observe that 96% execution time is consumed by galois field multiplication. Also as described in Chapter 2, there are

already plenty of methods proposed [4], [5], [6] to accelerate the galois field multiplication for either hardware or software implementation. In order to find an appropriate method for our DSP implementation, we do some evaluations on these three proposed methods. They are Mastrovito multiplier method, serial multiplier method and logarithmic table lookup method.

As proposed in [4], the Mastrovito algorithm is used to perform multiplication in the ground field GF(2^m) (in our case, m = 8). Before going through the algorithm, we first introduce the polynomial notation for galois field multiplication equation

$$A(y)B(y) = C(y) \bmod Q(y)$$

All elements in GF(2^8) are polynomials of degree less than 8 with binary coefficients：

$$c_7 y^7 + c_6 y^6 ... + c_1 + c_0 =$$
$$(a_7 y^7 + a_6 y^6 ... + a_1 + a_0)(b_7 y^7 + b_6 y^6 ... + b_1 + b_0) \bmod Q(y)$$

If we implement the galois field multiplication described above by directly performing the degree 8 polynomial multiplication and modulo operation on DSP, which is the case of original version galois field multiplier, it will result in a very slow galois field multiplier. This is because other than the 8 by 8 bit multiplication, it also requires plenty of branch instructions every time when doing the modulo operation, and from the pipeline structure we described above, branch instructions requires 6 execution phases to destination instruction. Hence, it is more time consuming comparatively.

Alternatively, Mastrovito has proposed an algorithm to speed up the galois field multiplication. First, the GF(2^8) elements B(y) and C(y) can be represented as column vectors consisting of the binary polynomial coefficients. By introducing a "*product matrix*" $\mathbf{Z} = f(A(y),Q(y))$, the galois field multiplication can be described as

$$C = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_7 \end{pmatrix} = ZB = \begin{pmatrix} f_{0,0} & \cdots & f_{0,7} \\ \vdots & \ddots & \vdots \\ f_{7,0} & \cdots & f_{7,7} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_7 \end{pmatrix}$$

The coefficients $f_{i,i} \in GF(2)$ of the product matrix depend recursively on the coefficients a_i of polynomial A(y) and q_i of the matrix **Q** which is derived by the binary field polynomial Q(y)

$$f_{i,j} = \begin{cases} a_i & ; j = 0 \quad ; i = 0,...,7 \\ u(i-j)a_{i-j} + \sum_{t=0}^{j-1} q_{j-1-t,i} a_{7-t} ; j = 1,...,7 ; i = 0,...,7 \end{cases}$$

where the elements $q_{i,j}$ of the matrix **Q** is defined by

$$q_{i,j} = \begin{cases} q_{i-1,7} & ; i = 1,...,6; j = 0 \\ q_{i-1,j-1} + q_{i-1,7} q_{0,j} ; i = 1,...,6; j = 1,...,7 \end{cases}$$

This method eliminates the modulo operation, which is required in the original version multiplier, and hence should be faster than the original one.

The second method proposed [5] is known as the serial multiplier which is originally designed for the implementation of the public-key cryptosystems that requires programmable multipliers in large galois fields. The algorithm of this multiplier is also derived from the basic galois field multiplication equation

$$A(y)B(y) = C(y) \bmod Q(y).$$

From this equation, we know that $GF(2^8)$ multiplication can be carried out by multiplying A(y) and B(y) and then performing the reduction modulo Q(y). But there is also an alternative way to do the same thing – By interleaving multiplication and reduction according to the equation

$$C'^{(i)} = xC'^{(i-1)} \bmod Q(y) + a_{8-i}B(y) \quad for\ i = 1,2,...,8; C'^{(0)} = 0; C(y) = C'^{(8)}$$

In this equation, $C'^{(i)}$ represent the partial results generated at step i of the recursion. The $a_0 \sim a_7$ are the binary coefficients of A(y). And the products $xW'^{(i-1)}$ are polynomials of degree k, which must be reduced modulo Q(y).

These reductions are done using the following identity.

$$x^8 = Q_7 x^7 + Q_6 x^6 \cdots + Q_1 x + Q_0 \bmod Q(y)$$

Each galois field multiplication using this algorithm requires a total of 64 bit multiplications and 7 polynomial reductions. The $GF(2^8)$ serial multiplier, sometimes

referred to as "MSB-First multiplier," is a polynomial basis multiplier that use 8 slices and computes GF(2^8) in 8 cycles. It is based on the algorithm described in the previous paragraph. To make it more clearly, the algorithm is rewritten and shown in Fig. 4.7, and its hardware realization is given in Fig. 4.8 for a reference purpose. This serial multiplier method is attractive for VLSI implementation, but for DSP implementation, it also has some advantages because it provides a parallel architecture for the galois field multiplier. It eliminates the complex branch instruction which is required in the original galois field multiplier in doing reduction modulo. Hence, the CCS compiler can perform the software pipelining more efficiently.

```
Algorithm Begin
C[−1] = 0
C[0 to 7] = 0
For i = 7 down to 0 Do
    Parallel For j = 0 to 7 Do
        C[j] = C[j − 1] + (A[i] * B[j]) + (C[7] * Q[j])
    End Parallel For
End For
Algorithm End
```

Figure 4.7: Algorithm for Serial Multiplier.

The third method proposed [6] is the logarithmic table lookup method. It is a well-known method for computing GF(2^n) arithmetic (both multiplication, squaring and inversion) for small values of n. In our case, the galois field is GF(2^8). So, a primitive element $g \in GF(2^8)$ is selected to serve as the generator of the field GF(2^8). Thus, an element A(y) in this field can be written as a power of g, that is $A(y) = g^i$, where

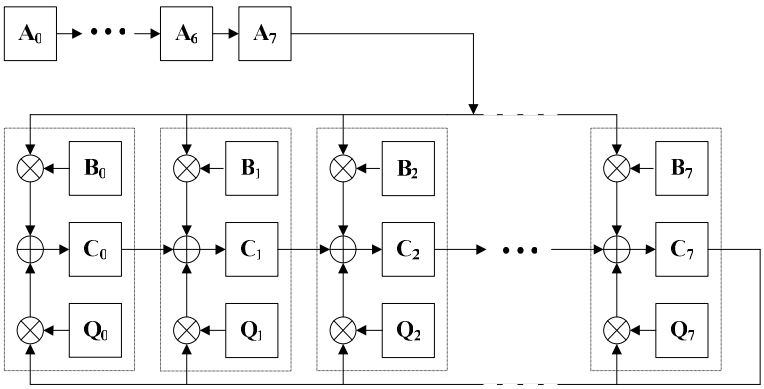


Figure 4.8: Serial Multiplier in GF(2^8).

$0 \le i \le 255$. Then, we compute the powers of the primitive element, g^i for i = 0,1,…,255, and then obtain 256 pairs of the form (A(y),i). Afterward, we can construct two tables that sorting these 256 pairs in two different ways: **the log table** is sorted with respect to A(y) and **the alog table** is sorted with respect to i. For example, for i = 26 and A(y) = g^{26}, we have log[A(y)] = 26 and alog[26] = A(y). These tables are then stored in the DSP internal memory and they are accessed when performing the field multiplication, the squaring, and the inversion operations. Given two elements $A(y), B(y) \in GF(2^8)$, we perform the multiplication C(y) = A(y)B(y) mod Q(y) as follows：

1. a := log[A(y)]
2. b := log[B(y)]
3. c := a + b (mod 255)
4. C(y) := alog[c]

This is due to the fact that C(y) = A(y) x B(y) = $g^i g^j = g^{i+j \bmod 255}$. The ground field multiplication requires three memory access, a single modular addition operation with modulus 255. The squaring of an element A(y) is slightly easier: only two memory access operations are required for computing C(y) = $A(y)^2$, as illustrated below

1. a := log[A(y)]

2. c := 2*a (mod 255)

3. C(y) := alog[c]

Similarly, the inversion of an element A(y) is computed using the property $C(y) = A(y)^{-1} = g^{-i} = g^{255-i}$, which requires two memory access operations

1. a := log[A(y)]

2. c := 255 – i

3. C(y) := alog[a]

This method has the advantage of low computation complexity. It only requires integer addition and a modulo 255 operation to perform either galois field multiplication or inversion or squaring. The cost of this method is that it requires more memory accessing compared to the previous two methods and it occupies larger and larger memory space when the values of n grows up. But since in our application, the value of n is only 8, the memory space shall not be a serious problem here.

Besides the above three methods, we also try to find some useful instructions or special architecture for $GF(2^8)$ multiplication that can well fit into the DSP hardware. Fortunately, we find that the C64x series DSP chips provide a special intrinsic function to perform the $GF(2^8)$ (and for $GF(2^8)$ only!) multiplication. The intrinsic function format is (unsigned int) _gmpy4(unsigned int A, unsigned int B). This function is capable of doing four $GF(2^8)$ multiplication simultaneously, but before performing the four simultaneous multiplication, we have to packet the four 8-bit galois field elements into a 32-bit register, and the packaging operation also consume execution time. Overall, it does not provide benefit if we need to packet the 8-bit galois field elements into a 32-bit register then perform one _gmpy4 intrinsic instruction. Therefore, we decide to perform only one galois field multiplication each time we call this intrinsic function, not four simultaneous multiplication.

Table 4.4 shows the simulation results of the original galois field multiplier, the three proposed galois field multiplier and the intrinsic galois field multiply instruction, which are generated by the CCS built-in profiler. The multiplier A refers to the original galois field multiplier; the multiplier B refers to the Mastrovito multiplier; the multiplier

C refers to the serial multiplier; the multiplier D refers to the logarithmic table lookup multiplier, and the multiplier E refers to the intrinsic galois field multiplier provided by the TI C64x series DSP chips. The notation "one mult" in the cycles column denotes that it is the cycle count for performing one galois field multiplication. From this table, we can observe that the multiplier B has the largest code size. We think it is due to the build-up of the "product matrix", but actually, the most memory space consuming multiplier is the multiplier D because it has to store two tables each contains 256 elements. The code size (4204) denotes the total size of the two tables plus the multiplier. The most efficient multiplier is the multiplier E. The C64x series DSP chips may contain an application-specified hardware structure for computing galois field multiplication. This conjecture is based on the evidence that the CCS compiler directly translate the _gmpy4 intrinsic function to an assembly instruction named GMPY4. Therefore, there is an assembly instruction used for the galois field multiplication. Likely, there is a specific hardware to perform this task.

To make our program platform independent, our attempt is to seek for an appropriate algorithm among B, C and D. As one may expect, the performance of any of these three multipliers shall not exceed the TI's intrinsic multiplier since it is accelerated by TI's hardware. We find that the logarithmic table lookup multiplier performance is still pretty good even compared with the intrinsic one. It means that a software-oriented algorithm is more appropriate for DSP implementation than a hardware-oriented algorithm. However, if we implement the hardware-oriented algorith-

Multiplier Type	Code Size	Cycles (One Mult)
GF_Multiplier A	584	292
GF_Multiplier B	1080	167
GF_Multiplier C	456	189
GF_Multiplier D	88 (4204)	22
GF_Multiplier E	12	6

Table 4.4: Comparison of the Five Different Galois Field Multiplier.

m on the built-in FPGA of Quixote DSP baseboard, we may have a totally different conclusion. The profile of simulation results for each revision step will be shown in section 4.5.

4.3.1.3 Compiler Level Improvements

In the last part of optimizing the RS encoder, we try to improve the speed of our program by tuning the CCS compiler's setting. Fig. 4.9 shows the compiler's feedback for the loop in the RS_Encode function. The compiler tells us that the loop contains a call, so it can not qualify this loop and hence the software pipeline is disabled. From Fig. 4.10 which shows the pseudo code of RS_Encode, we know that it is due to the calling of the galois field multiplier function "gmpy( )".

```
;*----------------------------------------------------------------------------*
;*      SOFTWARE PIPELINE INFORMATION
;*           Disqualified loop: Loop contains a call
;*----------------------------------------------------------------------------*
```

Figure 4.9: Compiler's Feedback for RS_Encode Loop.

```
for (i = 0; i < 239; i++) {
                feedback = data[i] ^ bb[0];
                for (j = 0; j < no_p-1; j++)
                        bb[j] = bb[j+1] ^ gmpy(Gg_poly[15-j],feedback);
                bb[15] = gmpy(Gg_poly[0],feedback);
        }
```

Figure 4.10: Pseudo Code for RS_Encode Loop.

In order to make the compiler work more efficiently, we set the "Opt. Level" option to "File" level. It makes the compiler capable of obtaining information of the entire program. As a result of this setting, now the compiler can deal with the function calls inside a loop. Fig. 4.11 shows the compiler's feedback after we set the "Opt. Level" to "File" level. We observe that now the compiler attempts scheduling both the

outer and inner loops, which has a function call inside, into a software pipelined loop. Compared to the original setting, the compiler now can software pipeline the key loop of our syndrome calculator and make 2 loops run in parallel with each loop completed in 18 cycles. Moreover, the “File” level optimization also inline the callee functions which locates inside a loop and hence it also reduces the overhead of calling the functions. When the functions inlined are frequently used, it can save a lot of execution time spent on function calling.

The reason we do not set the “Opt. Level” to “File” level initially is because the cycle count recorded by the profiler may be wrong for individual function if we set it at “File” level. This is due to the fact that the file level optimization usually schedules the instructions across functions and hence when we select the profile area, it may not be the original instruction there. We will later give a final profile based on the file optimization level setting in section 4.5.

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count          : 239
;*      Known Maximum Trip Count          : 239
;*      Known Max Trip Count Factor       : 239
;*      Loop Carried Dependency Bound(^)  : 12
;*      Unpartitioned Resource Bound      : 17
;*      Partitioned Resource Bound(*)     : 18
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1
;*      .D units                    16       18*
;*      .M units                    11        5
;*      .X cross paths               2        9
;*      .T address paths            17       17
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical    ops (.LS)         0        0      (.L or .S unit)
;*      Addition ops (.LSD)          8        8      (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1
;*      Bound(.L .S .D .LS .LSD)     8        9
;*
;*      Searching for software pipeline schedule at ...
;*           ii = 18 Schedule found with 2 iterations in parallel
```

Figure 4.11: Compiler’s Feedback for RS_Encode Loop

(After Build Option Change).

4.3.2 Optimization on RS Decoder

4.3.2.1 Galois Field Inversion

In addition to the galois field multiplication which is discussed in previous section, the galois field inversion is also required in the RS decoder. According to Fermat's algorithm, we know that $A(y)^{-1} = A(y)^{254} \bmod Q$ in $GF(2^8)$. The equation can be then computed in this way： $A^{254} = A^2 \cdot A^4 \cdot A^8 \cdots A^{128}$ [17]. In other words, it requires 13 multiplications to complete an inversion operation, which result in a huge computational complexity. In order to reduce the high complexity on doing galois field inversion, we shall find an appropriate inverter architecture for our RS decoder. Similar to the case in galois field multiplication, we employ the original $GF(2^8)$ multiplier, the three proposed $GF(2^8)$ multiplier and the intrinsic $GF(2^8)$ multiplier to do the multiplication operation in the inverter. The profile of simulation results is shown in Table 4.5, the notation "one inv" denotes the cycle count is obtained from performing one inversion and the notation A to E are the same as in the comparison of multiplier. Inverter A refers to the inverter using the original multiplier to perform the multiplication operation; inverter B refers to the case of using the Mastrovito multiplier; inverter C refers to the case of using the serial multiplier; inverter D refers to the case of using the logarithmic table lookup multiplier, and inverter E refers to the case of using the intrinsic multiplier. From the profile data, we find something different from the previous comparison between the multipliers. In the comparison for multipliers, the intrinsic multiplier is the fastest one among all the candidates, but in the case of inverter, the inverter employing logarithmic table lookup multiplier is about 15 times faster than the one using the intrinsic multiplier. We conclude that it is because only the logarithmic table lookup method can perform the inversion by simple integer subtraction not by the complex chain multiplication. The optimization discussed below

will use the logarithmic table lookup method to handle both the multiplication and inversion in the RS decoder.

Inverter Type	Code Size	Cycles (One Inv)
GF_Inverter A	300	4584
GF_Inverter B	360	2508
GF_Inverter C	300	2572
GF_Inverter D	44 (4160)	13
GF_Inverter E	304	196

Table 4.5: Comparison of the Five Different Galois Field Inverter.

4.3.2.2 Data Type Modification

Similar to the RS encoder, there are also several variables, which we originally declare as "long" data type in the RS decoder. At the beginning of optimization on the RS decoder, we first modify the variable declared as "long" data type to "int" data type. Table 4.6 shows the profile of our RS decoder before data type modification and Table 4.7 shows the profile after data type modification. In these tables, the RS_Syndrome function is used to calculate the syndrome of the received data, and the RS_Decode function is used to correct the errors and erasures of the received data based on the syndrome obtained in RS_Syndrome. The last three functions – Berlekamp,

Areas	Code Size	Cycles	Percentage (%)	Processing Rate (Kbits/sec)
Main Function	840	579249	100	298
RS_Syndrome	424	177400	30	
RS_Decode	928	396611	68	
Berlekamp	2176	53234	13	
Chien_Search	720	295723	75	
Forney	1308	46445	12	

Table 4.6: Original Profile of RS Decoder.

Areas	Code Size	Cycles	Percentage (%)	Processing Rate (Kbits/sec)
Main Function	832	369637	100	467
RS_Syndrome	312	144212	39	
RS_Decode	656	220247	60	
Berlekamp	1992	30036	14	
Chien_Search	652	172730	78	
Forney	760	16819	8	

Table 4.7: Profile of Revised RS Decoder (Data Type Modification).

Chein_Search and Forney are called by RS_Decode function and used to calculate the errata locator polynomial coefficients, search the roots of the errata locator polynomial and compute the magnitude of the errors and erasures, respectively. Similarly, the code size and execution cycles are improved significantly by this revision of data type.

4.3.2.3 Chien Search Improvement - I

Refer to Table 4.6, we observe that 60% of the total execution time is spent on the RS_Decode function. The Chien_Search function is a part of the RS_Decode function and it uses 78% of the RS_Decode execution time. So we first examine the Chien_Search function and see what we can do to reduce its computational complexity. Fortunately, there is a special feature in our RS code structure that we can exploit to simplify the Chien search procedure. As discussed in Chapter 2, we know that the Chien search is used to find the roots of the errata locator polynomial, and the method it used to find the roots is by exhaustively substitute all the field elements of GF(2^8) into the errata locator polynomial. When the sum equals zero, it means this field element is one of the polynomial's roots. In other words, it means we have to substitute 255 elements into the polynomial first, then we are sure that all the possible roots of this polynomial are found. To make the operation simpler, we employ a technique named "early termination", which adds a termination criterion in the substitution loop. The criterion

we set is the degree of the errata locator polynomial since the degree of a polynomial equals to the total number of roots it has. After the number of roots equal to the degree of the errata polynomial has been found, we can confirm that we have already found all the roots of the polynomial. Thus, we do not have to do field element substitutions anymore and can exit the substitution loop. Figs. 4.12(a) and 4.12 (b) show the pseudo code of the (48,36,6) RS code as an example for the Chien search function without early termination and with early termination, respectively. In the case of (48,36,6) code, the last 4 position of the codeword are always the roots of the errata locator polynomial since the last 4 codewords are marked as erasures. Hence, we only have to find (deg_lambda – 4) roots. Then, we can declare all roots have been found. Fig. 4.13 shows the flowchart that illustrates how the early termination is performed. The solid arrows denote the flow of early terminated Chien search while the dashed arrows denote the flow of original Chien search.

```
for (i = 1; i <= gf_nn_max; i++) {
        q = 1;
        for (j = deg_lambda; j > 0; j--)
                if (reg[j] != A0) {
                        reg[j] = (reg[j] + j)%nn;
                        q ^= Alpha_to[reg[j]];
                }
        if (!q) {
                root[count] = i;
                loc[count] = gf_nn_max - i;
                count++;
        }
}
```

(a)

```
for (i = 1; i <= gf_nn_max; i++) {
        q = 1;
        for (j = deg_lambda; j > 0; j--)
                if (reg[j] != A0) {
                        reg[j] = (reg[j] + j)%nn;
                        q ^= Alpha_to[reg[j]];
                }
        if (!q) {
                root[count] = i;
                loc[count] = gf_nn_max - i;
                count++;
        }
        if(count == deg_lambda - 4) break;
}
loc[count+1] = 255 - 252;
loc[count+2] = 255 - 253;
loc[count+3] = 255 - 254;
loc[count+4] = 255 - 255;
count = count + 4;
```

(b)

Figure 4.12: Pseudo Code for Chien Search (a) w/o Criterion. (b) w/ Criterion.

Table 4.8 shows the profile of the early terminated Chien search. The “worst case” denotes the case of not doing early termination and the “best case” denotes the case of doing early termination and the errors happen to be in the first symbol of the codeword. From this table we know that the operating speed of the best case is about 2.2 times faster than the original one (the worst case). For the average case, the operating speed is about 1.37 times faster than the original one.

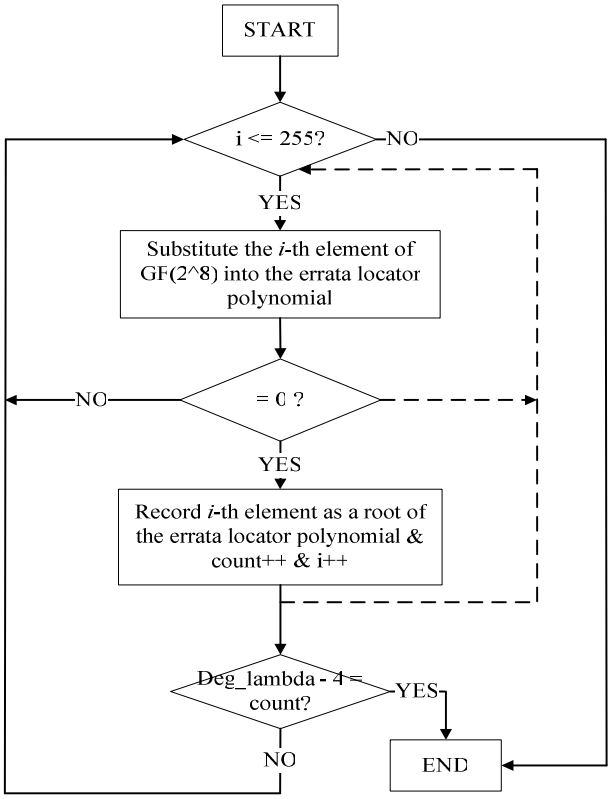


Figure 4.13: Flowchart of Early Terminated Chien Search.

Type	Cycles
Chien Search (Worst Case)	171526
Chien Search (Best Case)	78247
Average	124886

Table 4.8: Profile of the Worst Case and Best Case of Early Terminated Chien Search.

4.3.2.4 Chien Search Improvement - II

After adding the early termination into the Chien search, we find that even in the best case, it still requires 57645 cycles to find the root (only one root!). We are confused at first because if the error occurs in the first symbol of the codeword, the Chien search should find the root at the first attempt and should not spend so much execution time. After we carefully examine our program code, we finally find the reason. It is due to the fact that the RS code we use is a shortened RS code. Take the (48,36,6) RS code for example, when doing the RS decoding, 203 bytes of zero are inserted to the (48,36,6) codeword. That is to say, the first 203 symbols in the core (255,239,8) code will never be distorted by the channel noise. But if we still use the Chien search module which is originally designed for the (255,239,8) code, it always start finding the roots from the first symbol of the (255,239,8) code, which is never an error position. To reduce the redundant calculations due to this shortened code, we modify the original Chien search module to start searching roots from the first valid symbol, i.e., the 221st symbol for the (24,18,3) RS code, the 213th symbol for the (30,26,2) RS code, the 203rd symbol for the (48,36,8) RS code, and the 185th symbol for the (60,54,3) RS code. After the modification, the new profile of the Chien search module is shown in Table 4.9. As expected, the cycle count is greatly reduced compared with the original Chien search.

Type	Cycles
Chien Search (Worst Case)	34558
Chien Search (Best Case)	1387
Average	17972

Table 4.9: Profile of the Worst Case and Best Case of Early Terminated Chien Search (Modified).

4.3.2.5 Inverse-Free Berlekamp Massey Algorithm

The iterative algorithm discovered by Berlekamp and Massey (BM algorithm) for decoding RS Code is a well-known technique for finding the errata locator polynomial. Other decoding algorithms such as the Euclidean and continued fraction algorithm have a slightly higher complexity when compared with BM algorithm. However, the inversion of discrepancy needed in the computation of the original BM algorithm is complex and time-consuming due to the requirement of chain multiplications. Fortunately, there is a new inverse-free BM algorithm proposed in [18]. We are not going to describe the new algorithm in detail but simply explain the operation of this algorithm and then compare it to the original algorithm.

The inverse-free algorithm is formulated as follows (Based on the modified inverse-free BM algorithm proposed in [19], which eliminates the need for pre-computing the Forney syndrome and post-computing the errata locator polynomial in [18]).

1. Initially define $\mu^{(0)}(x) = \Lambda(x), \lambda^{(0)}(x) = \Lambda(x), l^{(0)} = 0, k = 0$ *and* $\gamma^{(0)} = 1$.
2. Set $k = k + 1$. If k >= 16 – s, stop. Otherwise, compute

$$\delta^{(k)} = \sum_{j}^{l^{(k-1)}} \mu_j^{(k-1)} S_{k-j}$$

$$\mu^{(k)}(x) = \gamma^{(k-1)} \mu^{(k-1)}(x) - \delta^{(k)} \lambda^{(k-1)}(x) x$$

$$\lambda^{(k)}(x) = \begin{cases} x \cdot \lambda^{(k-1)}(x), & \text{if } \delta^{(k)} = 0 \text{ or if } 2l^{(k-1)} > k-1 \\ \mu^{(k-1)}(x), & \text{if } \delta^{(k)} \neq 0 \text{ and } 2l^{(k-1)} \leq k-1 \end{cases}$$

$$l^{(k)} = \begin{cases} l^{(k-1)}, & \text{if } \delta^{(k)} = 0 \text{ and } 2l^{(k-1)} > k-1 \\ k - l^{(k-1)}, & \text{if } \delta^{(k)} \neq 0 \text{ and } 2l^{(k-1)} \leq k-1 \end{cases}$$

$$\gamma^{(k)} = \begin{cases} \gamma^{(k-1)}, & \text{if } \delta^{(k)} = 0 \text{ and } 2l^{(k-1)} > k-1 \\ \delta^{(k)}, & \text{if } \delta^{(k)} \neq 0 \text{ and } 2l^{(k-1)} \leq k-1 \end{cases}$$

3. Return to step 2.

Finally, the errata locator polynomial is computed as $\tau(x) = \mu^{16-s}(x)$, whereas the s denotes the number of erasures.

Compared to the original BM algorithm, it eliminates the use of inversion in the BM algorithm. To make sure this algorithm meets our needs, we perform an evaluation. Table 4.10 shows the profile of the comparison between the original BM algorithm and the new inverse-free BM algorithm, whereas the notation A~E are the five galois field multiplier described previously. As we expected, the inverse-free BM algorithm is faster than the original one since it eliminates the need for calculating inversion of discrepancy. For the first three case, due to the high computational complexity on galois field multiplication, the inverse-free BM algorithm outperforms the original one significantly (around 5 times faster). However in the last two cases, the advantage of the inverse-free algorithm has decayed greatly. It is due to that the inversion for the last two cases is based on table lookup, which is very simple and does not drastically affect the total execution time.

Type	Cycles (Original BM)	Cycles (Inverse-Free BM)	Improvement (%)
A	1185892	210616	463
B	555215	105753	425
C	628873	114714	448
D	30036	25175	19
E	16163	10913	48

Table 4.10: Comparison between the Original and the Inverse-Free BM Algorithm.

4.3.2.6 Compiler Level Improvements

Refer to the profile data given in Table 4.6, our next target in optimizing the RS decoder is focused on the syndrome calculator because 39% of the execution time is spent on it. As described in Chapter 2, the syndrome calculation structure is not complex. The cause that results in the slow processing speed is because of the massive substitutions of the field elements ($\alpha, \alpha^2 \cdots \alpha^{16}$) into the polynomial, which is composed of the received data sequence as its coefficients. The compiler's feedback of

the original syndrome calculator is shown in Fig. 4.14. Similar to the case in encoder part, we notice that in the last line it reports “Disqualified loop: Loop contains a call”. From the pseudo code of the syndrome calculator loop shown in Fig. 4.15, we know that it is due to the calling of galois field multiplier “gmpy( )”. As described above, we also turn on the “File” level optimization to allow the compiler to deal with the function calls inside a loop. Fig. 4.16 shows the compiler’s feedback after we set the “Opt. Level” to “File” level. It finally schedules the loop into a software pipelined loop, which runs 2 iterations in parallel with each iteration completed in 13 cycles.

```
;*--------------------------------------------------------------------------*
;*      SOFTWARE PIPELINE INFORMATION
;*           Disqualified loop: Loop contains a call
;*--------------------------------------------------------------------------*
```

Figure 4.14: Compiler’s Feedback for Syndrome Calculator Loop.

```
for (i = 0; i < nn; i++) {
        for (j = 1; j <= no_p; j++) {
                product = gmpy(Alpha_to[B0-1+j],s[j]);
                s[j] = product ^ data[i];
        }
}
```

Figure 4.15: Pseudo Code for Syndrome Calculator.

The second compiler level improvement we have done is on the Chien search function. Refer to Fig. 4.12 (b), we find that the trip count of the first inner loop depends on the deg_lambda, which is calculated by the BM algorithm and thus cannot be determined in the compiling stage. As a result of it, the compiler cannot decide how many times the loop will be executed and hence the flexibility for the compiler to arrange the resources is limited. To break this limit, we carefully examine the loop parameter, i.e., the maximum trip count, the minimum trip count and the maximum trip count factor, and find if any implicit information we can provide for the compiler. Fortunately, we find that the deg_lambda is always greater than the number of erasures since we employ the erasure locator polynomial as the initial status in the BM algorithm.

By using the pragma “MUST_ITERATE(min. trip count)”, we can send the loop information to the compiler for the purpose of giving the compiler more information available in compile stage and thus the efficiency of the Chien search is improved.

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count          : 255
;*      Known Maximum Trip Count          : 255
;*      Known Max Trip Count Factor       : 255
;*      Loop Carried Dependency Bound(^)  : 11
;*      Unpartitioned Resource Bound      : 10
;*      Partitioned Resource Bound(*)     : 10
;*      Resource Partition:
;*                                    A-side   B-side
;*      .L units                        0        0
;*      .S units                        1        0
;*      .D units                        9        10*
;*      .M units                        8        8
;*      .X cross paths                  0        8
;*      .T address paths                9        10*
;*      Long read paths                 0        0
;*      Long write paths                0        0
;*      Logical    ops (.LS)            0        0        (.L or .S unit)
;*      Addition ops (.LSD)             8        8        (.L or .S or .D unit)
;*      Bound(.L .S .LS)                1        0
;*      Bound(.L .S .D .LS .LSD)        6        6
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 11 Did not find schedule
;*         ii = 12 Did not find schedule
;*         ii = 13 Schedule found with 2 iterations in parallel
```

Figure 4.16: Compiler’s Feedback for Syndrome Calculator Loop

(After Build Option Change).

4.4 Optimization on Convolutional Code

Similar to the case for the RS code, we start the optimization from the convolutional encoder part, but soon we find that the original speed of the convolutional encoder is sufficient (around 11Mbits/sec). The structure of the encoder is simple, so we skip optimizing the convolutional encoder but do the optimization on the Viterbi decoder.

4.4.1 Optimization on Viterbi Decoder

4.4.1.1 Choose Appropriate Data Types for Branch Metric

The same case as in optimization for the RS codes, we first use the CCS built-in profiler to analyze the Viterbi decoder. Table 4.11 shows the original profile of the Viterbi decoder. This profile is obtained from the Viterbi decoder using the floating-point values for branch metric, which is mainly for the purpose of soft-decision decoding.

Areas	Code Size	Cycles	Percentage (%)	Processing Rate (Kbits/sec)
Main Function	1536	8393747	100	27
Initialize_State	892	335	0	
VD_Decode	960	8336355	100	

Table 4.11: Original Profile of Viterbi Decoder.

However, the speed performance is awful if we still use the floating-point numbers for branch metrics on DSP, since our DSP is a fixed-point processor. To do floating-point arithmetics, it uses multiple fixed-point instructions to simulate a floating-point operation. So in the first step on Viterbi decoder optimization, we try to convert the floating-point values to fixed-point values. Of course, this conversion does cause loss in preciseness, but since what we really care is the relative values of the branch metrics, not the absolute value of each branch metric, the conversion does not hurt the performance strongly. The conversion we have done is simply multiply the original floating-point value by 1000 then round it to integer. The profile of the Viterbi decoder using fixed-point values as branch metric is shown in Table 4.12. As expected, we observe a significant improvement on speed by avoiding using floating-point values.

Areas	Code Size	Cycles	Percentage (%)	Processing Rate (Kbits/sec)
Main Function	932	616349	100	374
Initialize_State	892	196	0	
VD_Decode	944	559879	100	

Table 4.12: Profile of Viterbi Decoder Using Fixed Point Value As Branch Metric.

4.4.1.2 Modified Path Recording - I

Table 4.13 shows the profile of the VD_Decode function in Table 4.12, we thus know that the ACS (Add-Compare-Select) computation takes 79% of the total execution time. To increase the Viterbi decoder efficiency, we concentrate on the ACS computation. As described in Chapter 2, the ACS consists of 32 butterfly structures; each butterfly can produces 2 state metrics, so totally we can obtain the 64 state metrics that we used to expand the trellis. However, unlike the case of RS decoder, a single ACS computation is simple and regular; the major reason that leads to slow operating speed is due to the massive computations, i.e., we have to do 64 ACS computation for only 1 output bit. Thus the key to accelerate the Viterbi decoder on the DSP platform is to make the program code well match the features of the CCS compiler. In other words, instead of trying to explore a new algorithm to perform the convolutional decoding, we choose to refine the C code to make it well software-pipelined by the CCS compiler, or

Areas	Code Size	Cycles	Percentage (%)
ACS	320	401665	79
Others (Metric Setup, Traceback)	712	105986	21

Table 4.13: Profile of VD_Decode Function.

if possible we avoid using the instructions which require long time to complete such as loading, storing or branching.

Firstly, reference to [7], we know that the register file is limited so we are forced to store the state metric in the internal data memory. However, for recording the optimal path, we only need to write down "0" or "1" to represent the results in our path tracking procedure since we exploit the butterfly structure in the ACS calculation. If we just record the information at bit level, we can store it in an internal register first and then put it in the internal data memory when the register is full. Hence, we can reduce the frequency on memory access. The revision is shown as the pseudo code in Fig. 4.17. We see that the path is recorded in the pointer that points to the address in the internal

```
/* Internal Data Memory */
  unsigned char  *pp = decoding_pathA;
  …
if(m1 > m0)
{
            nmetric[i] = m1;
            *pp |= mask;
}
else
{
            *pp -= ((*pp)&mask);
}
```

```
/* Internal Register */
unsigned char  dec_mask = 0;
  …
if(m1 > m0)
{
            metric[i] = m1;
            dec_mask |= mask;
}
else
{
        dec_mask -= ((dec_mask)&mask);
}
```

Figure 4.17: Pseudo Code for Recording Path in Internal Data Memory and Register.

data memory as shown on the left side of Fig. 4.17 while the path is recorded in the internal register as shown on the right side of Fig. 4.17. According to Table 4.1, the store instruction is in the three-cycle instruction category, so we can expected that the speed will be improved by using the register to temporarily store the path information.

4.4.1.3 Modified Path Recording - II

After analyzing the pseudo code shown in Fig 4.17, we can find that the frequency of memory accessing is reduced from 1 time per bit to 1 time per 8 bits. That is, we

need to do memory accessing once for every 8 bits (path information) are recorded. In order to improve the speed further, we want to reduce the memory access time as much as possible. In the previous section, we use the register declared as “unsigned char” to record the path information, the size of the “unsigned char” is 8 bit. So every 8 bits (path information) we need to do memory accessing or the old path information will lose due to overflow. In this section, we further declare the register used to record the path information as “unsigned int”, which is the largest data type that can store bit information and do not use extra instructions to load/store the register. That is, the size of “long” is larger than “int” but requires extra instructions to load/store value from it. By using the “unsigned int” register, the frequency of memory accessing is reduced from 1 time per 8 bit to 1 time per 32 bit and thus the speed is improved further. The profile data of these two optimization are shown later in this chapter.

4.4.1.4 Counter Splitting

Refer to the CCS compiler’s feedback which is shown in Fig. 4.18, we notice that the core loop of the ACS computation is not software-pipelined well since it requires 17 cycles per iteration. It is too slow as a key loop of our Viterbi decoder. By carefully examining the feedback information, we find the problem is due to the strong dependency between the instructions inside the loop and the dependency mainly comes from the counter i and i/2 used in the butterfly structure for ACS computation. To break the dependency between these two counters, we exploit the relationship between these two counters which is shown in Fig. 4.19 that each time when i is increased by 2, i/2 is increased by 1. That is to say, we can declare a new counter named j, which initializes as zero and is increased by 1 at the end of each iteration. This counter j is actually equivalent to i/2 but do not have the dependency with counter i since we do not declare j = i/2. The compiler’s feedback after counter splitting is shown in Fig. 4.20, from these figures we know that the loop is software-pipelined better after we split the counter manually. Finally, the loop is software-pipelined as 3 iterations run in parallel and each

is completed in 7 cycles, while the original one is only software-pipelined as 2 iterations run in parallel and each is completed in 17 cycles.

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count             : 32
;*      Known Maximum Trip Count             : 32
;*      Known Max Trip Count Factor          : 32
;*      Loop Carried Dependency Bound(^)     : 15
;*      Unpartitioned Resource Bound         : 8
;*      Partitioned Resource Bound(*)        : 14
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        3
;*      .S units                     1        2
;*      .D units                     0       11
;*      .M units                     0        0
;*      .X cross paths               0        0
;*      .T address paths             0       11
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical    ops (.LS)         0        0     (.L or .S unit)
;*      Addition ops (.LSD)          2       26     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        3
;*      Bound(.L .S .D .LS .LSD)     1       14*
;*
;*      Searching for software pipeline schedule at ...
;*          ii = 15 Did not find schedule
;*          ii = 16 Did not find schedule
;*          ii = 17 Schedule found with 2 iterations in parallel
```

Figure 4.18: Compiler's Feedback for ACS Loop.

```
/* Dependent Counters */
for(i=0;i< 64; i+=2)
{

  b1 = mets[Syms[i]];

  nmetric[i] = m0 = cmetric[i/2] + b1;

  b2 = mets[Syms[i+1]];

  b1 -= b2;

  m1 = cmetric[(i/2) + 32] + b2;
...
}
```

```
/*  Independent Counters */
j=0    /* j = i/2 */;
for(i=0;i< 64; i+=2)
{
b1 = mets[Syms[i]];

nmetric[i] = m0 = cmetric[j] + b1;

b2 = mets[Syms[i+1]];

b1 -= b2;

m1 = cmetric[j + 32] + b2;
...
j++
}
```

Figure 4.19: Pseudo Code for Counter Splitting.

```
;*     SOFTWARE PIPELINE INFORMATION
;*
;*       Known Minimum Trip Count                : 32
;*       Known Maximum Trip Count                : 32
;*       Known Max Trip Count Factor             : 32
;*       Loop Carried Dependency Bound(^)        : 6
;*       Unpartitioned Resource Bound            : 6
;*       Partitioned Resource Bound(*)           : 7
;*       Resource Partition:
;*                                    A-side     B-side
;*       .L units                        2          0
;*       .S units                        1          1
;*       .D units                        6          5
;*       .M units                        0          0
;*       .X cross paths                  2          2
;*       .T address paths                6          5
;*       Long read paths                 0          0
;*       Long write paths                0          0
;*       Logical    ops (.LS)            0          0        (.L or .S unit)
;*       Addition ops (.LSD)             6         15        (.L or .S or .D unit)
;*       Bound(.L .S .LS)                2          1
;*       Bound(.L .S .D .LS .LSD)        5          7*
;*
;*       Searching for software pipeline schedule at ...
;*          ii = 7    Schedule found with 3 iterations in parallel
```

Figure 4.20: Compiler's Feedback for ACS Loop (After Counter Splitting).

4.4.1.5 Removal of Replicated Metrics

In this subsection, we remove the redundancy in the ACS loop to make it faster. Refer to the pseudo code shown on the left side of Fig. 4.21, which shows the operation for updating the state metric value. This operation must be performed after finishing 64 ACS computations for updating the old state metric with the newly computed state metric. This code can be eliminated by manually unrolling the ACS loop into 2 sub-loops as shown on the right side pseudo code of Fig. 4.21. These two sub-loops executed in turns is equivalent to the operation of updating state metrics. In addition, by manually unroll the ACS loop, the instruction level parallelism is also increased and thus the compiler can pipeline the ACS loop even better than the original program.

```
/* Metrics Replica */
for(…)
{
  …
  nmetric[I] = cmetric[I] + metric;
  ….
}
for( i=0; i< 64; i++)
        cmetric[i] = nmetric[i];
```

```
/* Removing Replica by Loop Unrolling */
For odd iterations
{
  …
  nmetric[i] = cmetric[i] +metric;
  …
}
For even iterations
{
  …
  cmetric[i] = nmetric[i] +metric;
  …
}
```

Figure 4.21: Pseudo Code For Removing Replicated Metrics.

4.5 Simulation Results

In this section, we present some simulation profiles generated by the CCS built-in profiler for the FEC scheme in IEEE 802.16a. The results of each optimization step described formerly are also shown in the these simulation profiles. So, we can understand how much improvement can be obtained by our optimizations. At the end, the overall profiles of the FEC encoder and FEC decoder for the four required coding scheme defined in the IEEE 802.16a standard are also shown for evaluating the processing rate of our improved FEC programs.

4.5.1 Simulation Profile for RS Encoder

Table 4.14 shows the simulation profile of our RS encoder for encoding 36 bytes data. The profile can be categorized into 5 areas. The first one is the simulation result for original program. The second one is the result for modification on data type. The third one is the result for three galois field multiplier (Mastrovito, Serial, and Logarithmic Table Lookup), which is marked as shadow area surrounded with double-line. The fourth one is the result for employing intrinsic galois field multiplier to

perform galois field multiplication in the RS encoding procedure. The fifth one is the result for modification on build option. We set it to the “File” level optimization and we can also inline the galois field multiplier function code inside the encoding function to reduce the huge overhead for calling it frequently. The label “I/O Included” in the table means the execution time spent on I/O operation using fread( ) and fwrite( ) is included in the cycle count. If the streaming mechanism of the Quixote baseboard functions correctly, the execution time spent on I/O operation should be different from using fread( ) and fwrite( ) (hopefully better than using fread ( ) and fwrite( )). For reference, the profile which excludes the execution time spent on I/O operations is shown in

Optimization Step	Code Size	Cycles	Processing Rate (Kbits/sec)	Improvement (%)
Original	2928	1433434	120	N/A
Data Type Modification	2284	1265024	136	13
Mastrovito Multiplier	3056	678174	254	86
Serial Multiplier	2100	766120	225	65
Logarithmic Table Lookup Multiplier	5692	137394	1257	824
Intrinsic Multiplier	1464	77799	2221	76
Compiler Level Optimization	1848	8036	21503	868

Table 4.14: Profile of Reed-Solomon Encoder (I/O Included).

Table 4.15. From the simulation profile, we know that the most efficient modification on the RS encoder is the replacement of original galois field multiplier by TI C64x intrinsic galois field multiplier. But if we implement the RS encoder on other DSP board which does not have an intrinsic galois field multiplier, then the most efficient modification shall be the replacement of original galois field multiplier by the

logarithmic table lookup multiplier. The improved processing rate of our RS encoder is about 21Mbits/sec with I/O included or 37Mbits/sec with I/O excluded.

Optimization Step	Code Size	Cycles	Processing Rate (Kbits/sec)	Improvement (%)
Original	2928	1430005	120	N/A
Data Type Modification	2284	1261595	136	13
Mastrovito Multiplier	3056	674745	256	88
Serial Multiplier	2100	763689	226	66
Logarithmic Table Lookup Multiplier	5692	133954	1289	847
Intrinsic Multiplier	1464	74371	2323	80
Compiler Level Optimization	1848	4607	37508	1514

Table 4.15: Profile of Reed-Solomon Encoder (I/O Excluded).

4.5.2 Simulation Profile for RS Decoder

Tables 4.16 and 4.17 show the simulation profile of our RS decoder with I/O included and with I/O excluded, respectively. This profile is generated when decoding 48 bytes received data. Besides the intrinsic multiplier, the second best improvement is with the Chien search modification. In the last step, we do further improvements based on compiler level tuning. It includes setting the “Opt. Level” in CCS build option from “Function” to “File”, using pragma MUST_ITERATE to provide the minimum trip count information for compiler and inline the functions for the frequently used functions such as galois field multiplier and inverter to reduce the overhead on calling these functions. Finally we can achieve a processing rate of 7.9Mbits/sec with I/O included or 10.4Mbits/sec with I/O excluded.

Optimization Step	Code Size	Cycles	Processing Rate (Kbits/sec)	Improvement (%)
Original	10688	579249	298	N/A
Data Type Modification	9464	369637	467	57
Chien Search - I	9496	354090	488	5
Chien Search - II	9316	217101	796	63
Inverse-Free BM	9608	212270	814	2
Intrinsic Multiplier	9368	115721	1493	83
Compiler Level Optimization	9912	21699	7964	433

Table 4.16: Profile of Reed-Solomon Decoder (I/O Included).

Optimization Step	Code Size	Cycles	Processing Rate (Kbits/sec)	Improvement (%)
Original	10688	574011	301	N/A
Data Type Modification	9464	364459	474	57
Chien Search - I	9496	348912	495	5
Chien Search - II	9316	211923	815	65
Inverse-Free BM	9608	206869	834	2
Intrinsic Multiplier	9368	110543	1563	87
Compiler Level Optimization	9912	16521	10459	569

Table 4.17: Profile of Reed-Solomon Decoder (I/O Excluded).

4.5.3 Simulation Profile for CC Encoder

Optimization Step	Code Size	Cycles	Processing Rate (Kbits/sec)	Improvement (%)
Original	768	20596	11186	N/A

Table 4.18: Profile of Convolutional Encoder (I/O Included).

Table 4.18 shows the simulation profile of our convolutional encoder for encoding 48 bytes data, the original processing rate is about 11Mbits/sec. Table 4.19 shows the simulation profile which excludes the execution time spent on I/O operation, the processing rate is about 15Mbits/sec. This processing speed satisfies our requirements. There is no need for further improvement currently.

Optimization Step	Code Size	Cycles	Processing Rate (Kbits/sec)	Improvement (%)
Original	768	15325	15034	N/A

Table 4.19: Profile of Convolutional Encoder (I/O Excluded).

4.5.4 Simulation Profile for CC Decoder

Table 4.20 shows the simulation profile of our soft decision decoding Viterbi decoder for decoding 72 bytes received data (4608 input bytes actually, since each received data bit is represented by two integer (32-bit) branch metrics for soft decision decoding). Similar to the case of RS code, the profile which excludes the execution time spent on I/O operation is shown in Table 4.21 for a reference. From these two tables, we can see that the most significant improvement is on the fixed point modification because our DSP is designed for fixed-point calculation.

Optimization Step	Code Size	Cycles	Processing Rate (Kbits/sec)	Improvement (%)
Original	3388	8393747	27	N/A
Fixed Point	2856	616349	374	1285
Path Recording I	2828	446506	516	38
Path Recording II	3020	355547	648	26
Counter Splitting	3284	237030	972	50
Removal of Metric Replica	3896	209261	1101	13

Table 4.20: Profile of Soft Decision Decoding Viterbi Decoder (I/O Included).

The second best improvement is on the counter splitting because it enables the software-pipeline working much better on the key loop (ACS loop) of our Viterbi decoder. Finally, we can achieve a processing rate of 1101 Kbits/sec with I/O included or 1514 Kbits/sec with I/O excluded.

Optimization Step	Code Size	Cycles	Processing Rate (Kbits/sec)	Improvement (%)
Original	3388	8336690	28	N/A
Fixed Point	2856	560080	411	1367
Path Recording I	2828	392896	586	43
Path Recording II	3020	298492	772	32
Counter Splitting	3284	179976	1280	66
Removal of Metric Replica	3896	152206	1514	18

Table 4.21: Profile of Soft Decision Decoding Viterbi Decoder (I/O Excluded).

4.5.5 Simulation Profile for FEC Encoder

In this subsection, we show the improved profile of our FEC encoder, which concatenates the RS encoder and the convolutional encoder. After encoding around 100 bytes data (108 bytes for scheme 1, 3, 4; 104 bytes for scheme 2 to meet the code specification), Table 4.22 shows the average improved profile for the four required

Modulation	RS Code	CC Code Rate	Code Size	Cycles		Processing Rate (Kbits/sec)	
				W/	W/O	W/	W/O
QPSK	(24,18,3)	2/3	2384	86762	77938	5975	6651
QPSK	(30,26,2)	5/6	2464	56312	48758	8865	10238
16-QAM	(48,36,6)	2/3	2388	61677	54044	8405	9592
16-QAM	(60,54,3)	5/6	2476	59638	52854	8692	9808

Table 4.22: Profile of Forward Error Correction Encoder.

coding scheme defined in the IEEE 802.16a standard, whereas the "W/" and "W/O" represent "with I/O" and "without I/O", respectively. Finally, the average processing rate of the FEC encoder can reach 7984 Kbits/sec with I/O included or 9072 Kbits/sec with I/O excluded after improvement.

4.5.6 Simulation Profile for FEC Decoder

Table 4.23 shows the average improved profile obtained in decoding the coded data generated in the above simulation. We find that the average processing rate of the FEC decoder can achieve 750 Kbits/sec with I/O included or 960 Kbits/sec with I/O excluded after improvement.

Modulation	RS Code	CC Code Rate	Code Size	Cycles		Processing Rate (Kbits/sec)	
				W/	W/O	W/	W/O
QPSK	(24,18,3)	2/3	12836	815093	650539	636	797
QPSK	(30,26,2)	5/6	12960	673042	535694	742	932
16-QAM	(48,36,6)	2/3	12652	696082	533861	745	971
16-QAM	(60,54,3)	5/6	12936	591806	456128	876	1137

Table 4.23: Profile of Forward Error Correction Decoder.

Chapter 5

ACS Unit Acceleration by Employing Xilinx FPGA as an Assistant

Based on the simulation results discussed in Chapter 4, we know the speed bottleneck of our FEC program is the soft decision decoding Viterbi decoder. Furthermore, we know that the most time consuming kernel in the Viterbi decoder is the Add Compare Select (ACS) unit. In order to speed up the ACS unit, we have done some optimization on DSP platform. However, the final speed is still slower than we wish. The reason for the slow operating speed of the ACS unit is the massive sequential computations required to obtain a single output bit; i.e., it requires 64 ACS computations to obtain a single bit output. We notice that the ACS unit is suitable for the FPGA implementation, on which we can design and allocate as many functional units as we want as long as it does not exceed the area limit of the FPGA. Clearly, we can accelerate the ACS unit based on the Xilinx FPGA XC2V2000, which is embedded on the Quixote DSP baseboard, by simply placing 64 ACS units and make them operated in parallel on FPGA. And then integrated it with the original DSP program to make the overall speed performance of our FEC decoder faster. In this chapter, we test two ACS design on FPGA and evaluate how much improvement we may achieved with the assistance of FPGA. Similar to the case in DSP implementation, the Xilinx FPGA on Quixote board must be controlled by the DSP program. However,

the communication mechanism of the Quixote board does not work. Thus, the simulations shown in following sections are obtained from the Debussy's nWave tools.

5.1 ACS Design - I

5.1.1 Original ACS Structure

The original ACS structure we designed is shown in Fig. 5.1, where SM1 and SM2 denote the upper state metric and lower state metric of the ACS butterfly structure shown in Chapter 2. And BM1 and BM2 denote the upper branch metric and lower branch metric, respectively, CTL_IN and CTL_OUT denote the input control signal and output control signal, SEL denotes the path record information and N_SM denotes the next state metric after ACS computation. This structure can operate

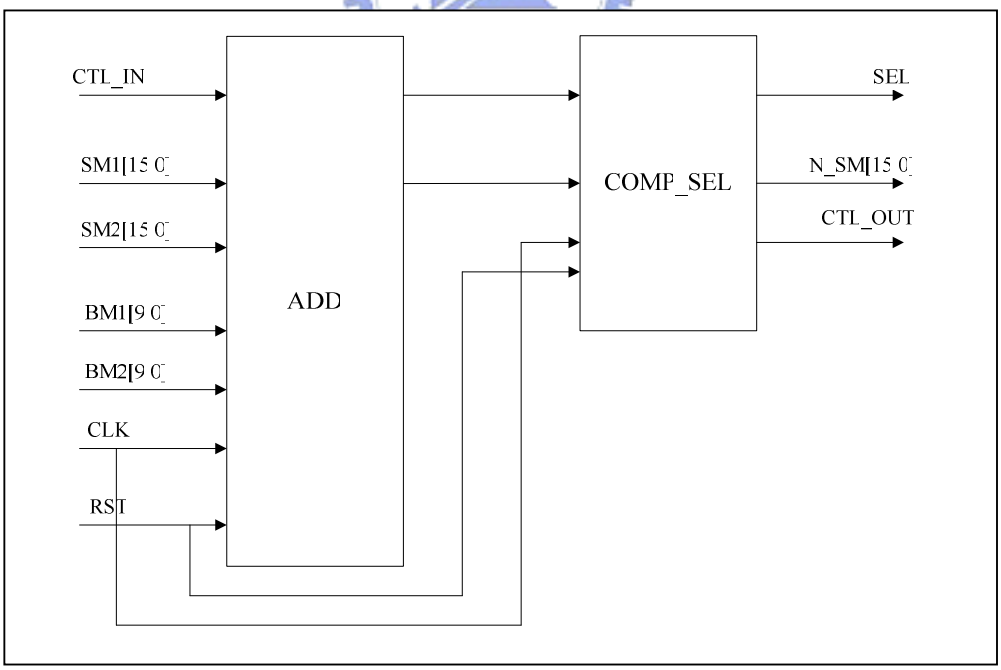


Figure 5.1: Block Diagram of Original ACS Design.

at around 100MHz, which can be translated to a processing rate of 12.5M (64 states/sec). The unit "64 states/sec" represents how many 64 state metrics can be computed per second, since the Viterbi algorithm has to compute 64 state metrics to produce 1 decoded output bit. The ACS module implemented on FPGA is much faster than on DSP. The DSP version only achieves 2M (64 states/sec). However, we find a physical limit after we finish the original design. It is the transmission bandwidth limit on our implementation platform. Based on the architecture of Quixote DSP baseboard which has been discussed in Chapter 3, we know that the communication between DSP and FPGA must go through the EMIF (External Memory InterFace) A, and the bandwidth of the EMIF A is 64-bit/133MHz, or 8512Mbps. Although the bandwidth is wide enough for most applications, it is still not sufficient for the ACS module we designed originally. According to Fig. 5.2, which shows the synthesis report generated by Xilinx ISE6.1 for our original design, the ACS module requires 690 bits data transmission for the ACS computation of 16 state metrics. Equivalently, it means 690*4 bits data transmission for decoding 1 bit. Use the notation of the EMIF A, we can translate it to 8512/(690*4) = 3.08M (64 states/sec). It means that the bandwidth of EMIF A can only support the processing rate of our original ACS module up to 3.08M (64 states/sec). Thus, if we do not do any modification, the processing rate of 12.5M (64 states/sec) is meaningless when we actually integrate the FPGA ACS module to the residual DSP program.

Device utilization summary:

Selected Device : 2v6000ff1152-6

Number of Slices:	1122	out of	33792	3%
Number of Slice Flip Flops:	1104	out of	67584	1%
Number of 4 input LUTs:	1312	out of	67584	1%
Number of bonded IOBs:	690	out of	824	83%

Figure 5.2: FPGA Synthesis Report for Original ACS Design.

Obviously, the processing rate of our design is limited by the data transmission rate that EMIF A can support. There are two possible solutions to solve this problem. The first one is to find a faster communicatoin interface between the DSP and FPGA, but it seems hard to do so. So we consider another approach: reduce the pin used in our design. The first improvement we have done on reducing the number of used pin is to avoid inputting the state metric values to FPGA since the state metric value is usually large and require more bits to represent. This can be achieved by storing the state metric values inside the FPGA. Since the initial value of the state metrics are known to be zero, we only have to reset the state metrics to zero at the beginning of decoding and then keep the updated state metrics in the registers inside FPGA for next time stage computation. Another possibility to reduce the pins is to reduce the input data size before they are sent to the FPGA device. Thus, we can represent the branch metric or state metric with fewer bits. After surveying several papers and textbooks, we find that a quantization level of 8 on branch metric is reasonable, according to the study by [20]; It results in a slight decrease on coding gain if the number of quantization level is greater than 8. Together, the 8-level quantization solution with the elimination of state metric transmission can provide a processing rate up to 32 M (64 states/sec), which is much better compared with the original 3.08M (64 states/sec).

5.1.2 Improved ACS Structure

The modified ACS structure is shown in Fig. 5.3. The core module is the ACS64 module, which consists of 64 ACS units for computing the 64 state metric for the next stage every two cycles in parallel. IN_BUF and OUT_BUF denote the input buffer and output buffer respectively, and they are used to link with the DSP device; i.e., this two buffers are used to temporarily store the data transmitted from the DSP side and store the data to be transmitted to the DSP, respectively. The newly added COMP module is a comparator that compares 64 resulting state metrics and find the minimum of the

metrics for the purpose of eliminating state metric input and output. It is because if we do not send back the state metric to the DSP side, we must implement the comparator on FPGA to select the best terminating state each time the time stage equals to the truncation length of our Viterbi decoder. To make the comparator operate more efficiently, we manually partition a comparator into five stages, and the pipeline stage is controlled by the FSM module. That is to say, the comparator will output the state with the smallest state metric value 5 cycles later after it receives the END_REQ control signal transmitted by the DSP side program.

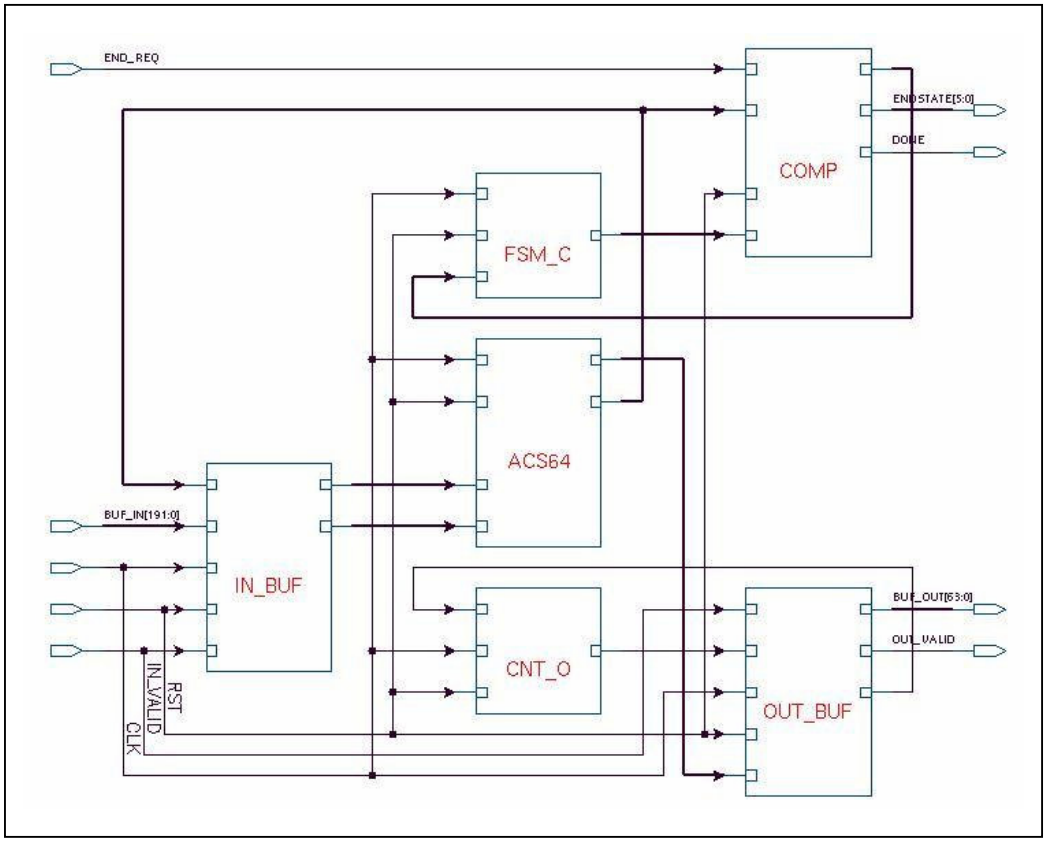


Figure 5.3: Schematic of Modified ACS Design.

According to the synthesize report for our modified ACS design, which is shown in Fig. 5.4, we can find that the slices used in our new (quantized) ACS design is greater than the original design. It is mainly due to the addition of comparator. As expected, the IOBs used in the new design are greatly reduced because of the use of quantization on

branch metric and the avoidance for outputting intermediate state metric values. Thus the processing rate constraint can be greatly reduced. From the synthesis report, the estimated clock rate is around 161Mhz. From the P&R report, which is shown in Fig. 5.5, the post P&R clock rate of our design is 1/11.089ns = 90Mhz, due to the fact that the output value is valid every 2 cycles, it can translate to a processing rate of 45M (64 states/sec). The waveform of the modified ACS design is shown in Fig. 5.6, the control signal IN_VALID is transmitted by DSP to inform FPGA that the data in input buffer is valid and the control signal OUT_VALID is transmitted by FPGA to inform DSP that the output data in the output buffer is valid, the control signal END_REQ is sent by DSP based on the truncation length of our Viterbi decoder to inform FPGA that DSP now requires the number of best state to perform the Traceback operation, as described above, the result of comparison is valid in 5 cycles later and the FPGA will send a

Device utilization summary:

Selected Device : 2v2000ff896-6

Number of Slices:	3785	out of	10752	35%
Number of Slice Flip Flops:	3049	out of	21504	14%
Number of 4 input LUTs:	6131	out of	21504	28%
Number of bonded IOBs:	266	out of	624	42%

Timing Summary:

Speed Grade: -6

Minimum period: 6.175ns (Maximum Frequency: 161.956MHz)
Minimum input arrival time before clock: 4.349ns
Maximum output required time after clock: 4.812ns
Maximum combinational path delay: No path found

Figure 5.4: FPGA Synthesis Report for Modified ACS Design.

control signal DONE to inform DSP that the result of comparison is valid now. One thing to be noticed is that the time interval between two consecutive IN_VALID signal cannot be less than 2 clock cycle time since the ACS unit need 2 cycles to complete the

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0	
The AVERAGE CONNECTION DELAY for this design is:	1.082
The MAXIMUM PIN DELAY IS:	11.089
The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:	5.369

Figure 5.5: Place and Route Report for Modified ACS Design.

computation and raise the OUT_VALID signal.

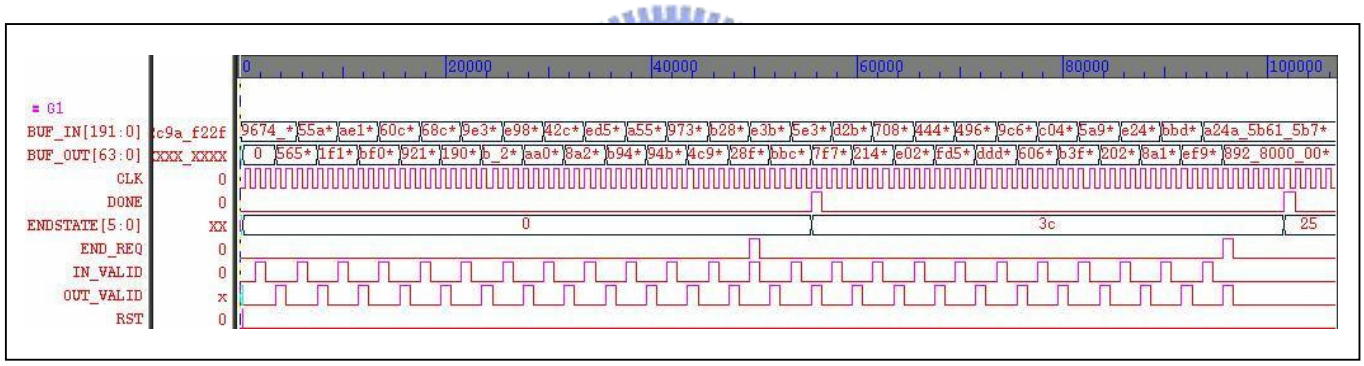


Figure 5.6: Waveform of Modified ACS Design.

5.2 ACS Design - II

Although the processing rate of the previous design, which is 45M (64 states/sec), already exceeds the physical limitation, which is 32M (64 states/sec) of the transmission bandwidth. For research and evaluation, we try to further improve the ACS design. According to [21], a new architecture of the ACS unit is proposed to accelerate the ACS unit by increasing the states of trellis. By reformulating the Viterbi algorithm, the proposed architecture provides an alternative approach to the high-throughput design.

The new proposed architecture is called “double state” architecture, which refers to the fact that it requires twice more states than the original architecture. By having double states, the serial operation of ACS unit can be transformed into parallel operation.

The basic concept of this new architecture is briefly introduced here. For simplicity, we take the convolutional code with generator polynomial X+1 as an example. By extending the original generator polynomial of the convolutional code one degree and set the coefficient of the highest degree to zero, one can obtain an equivalent trellis as illustrated in Fig. 5.7, where X+1 is the original generator polynomial and $0 \cdot X^2 + X + 1$ is the extended generator polynomial.

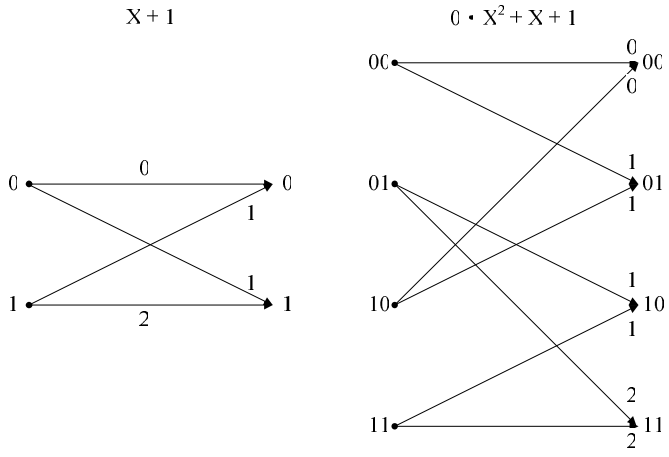


Figure 5.7: Two Equivalent Trellises.

In the double state trellis, the branch metrics (BMs) ending at the same state are all equal. That is to say, when comparing the state metrics for the next stage, we can select the minimum of the two previous stage state metrics without waiting for an addition of the BMs. Fig. 5.8 (a) shows the original ACS architecture while Fig. 5.8 (b) shows the modified “double state” ACS architecture, where n denotes the current stage, n+1 denotes next stage, $BM_{i,k}$ denotes the branch metric that start from state i and end at state k, and SM_i denotes the state metric of state i. In the original ACS architecture, we have to wait for the addition of BMs done, then we can compare the SMs for next stage. But in the new architecture, since the BMs ending at the same state are equal, we do not

have to wait for the addition of BMs and can just compare the current two state metrics directly. Equivalently, we perform the

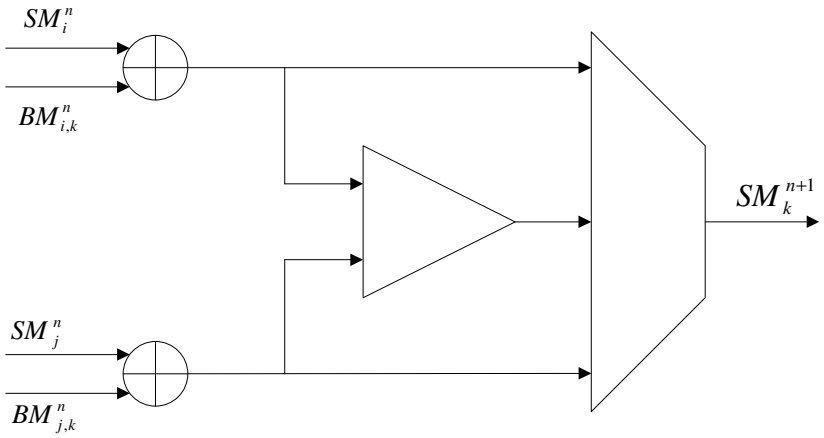


Fig. 5.8: (a) Original ACS Architecture.

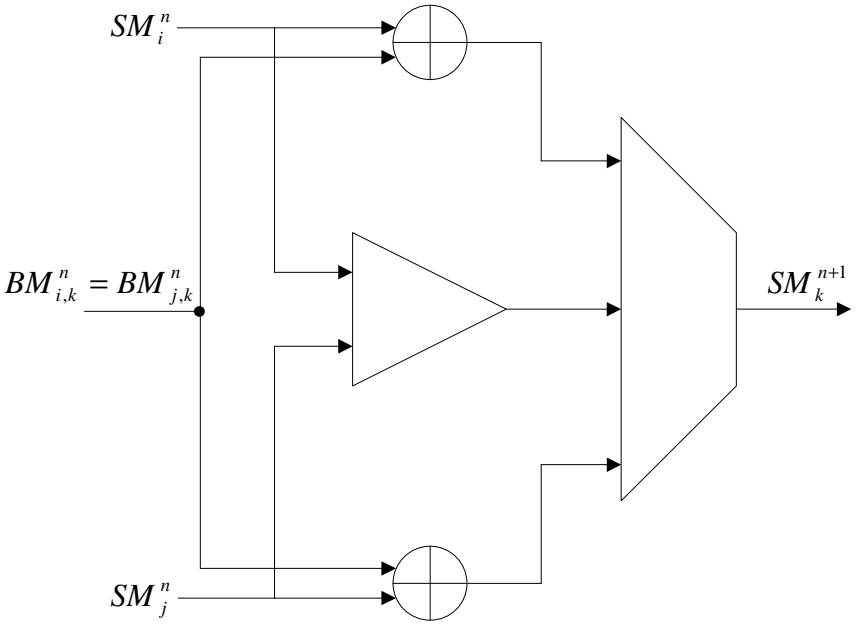


Fig. 5.8: (b) ACS Architecture Based on Double State Trellis.

following recursion: $SM_k^{n+1} = \min_i(SM_i^n) + BM_k^n$. Therefore, we can make the addition of BMs and the comparison of the SMs operate in parallel while the original ACS architecture must operate sequentially. Moreover, another hardware savings is also proposed based on the double state architecture. Looking at the next states 10 and 11 in Fig. 5.7, they share the same pair of the current states 01 and 11. Hence, if the next state 11 choose the path from the current state 01 over one from the current state 11, then the same decision is made at the "Select" operation for the next state 11. Therefore, every two states in the "double state" architecture can share the same decision-making unit in the "Compare" operation.

The schematic of the new ACS design based on double state trellis architecture is almost the same as before, the only difference is the ACS64 unit should be extend to ACS128 to deal with the double state ACS, the synthesis report for this new ACS design is shown in Fig. 5.9. From the synthesis report, the estimated clock rate of our

Device utilization summary:

Selected Device : 2v2000ff896-6

Number of Slices:	6782	out of	10752	63%
Number of Slice Flip Flops:	5594	out of	21504	26%
Number of 4 input LUTs:	6131	out of	21504	48%
Number of bonded IOBs:	266	out of	624	42%

Timing Summary:

Speed Grade: -6

Minimum period: 4.347ns (Maximum Frequency: 230.053MHz)
Minimum input arrival time before clock: 4.349ns
Maximum output required time after clock: 4.812ns
Maximum combinational path delay: No path found

Figure 5.9: FPGA Synthesis Report for Double State ACS Design.

new double state design is about 230MHz (1.43 times faster than the previous design), but the area increased is about 1.8 times. Though in our case, the area is not the major consideration, in actual ASIC design, it may not be a good solution considering both area and speed. From the P&R report shown in Fig. 5.10, the actual clock rate on FPGA is greatly decreased due to the routing factor, the post P&R clock rate is 1/10.214ns = 98MHz, which can be translated to a bitrate of 49M (64 states/sec).

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0	
The AVERAGE CONNECTION DELAY for this design is:	1.185
The MAXIMUM PIN DELAY IS:	10.214
The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:	6.541

Figure 5.10: Place and Route Report for Double State ACS Design.

Macro Statistics	(a)
Registers	135
1-bit register	67
13-bit register	64
832-bit register	1
192-bit register	1
3-bit register	1
64-bit register	1
Multiplexers	64
2-to-1 multiplexer	64
Adders/Subtractors	128
13-bit adder	128
Comparators	127
13-bit comparator less	63
13-bit comparator less equal	64

Macro Statistics	(b)
Registers	199
1-bit register	67
13-bit register	128
1664-bit register	1
192-bit register	1
3-bit register	1
64-bit register	1
Multiplexers	128
2-to-1 multiplexer	128
Adders/Subtractors	256
13-bit adder	256
Comparators	191
13-bit comparator less	127
13-bit comparator less equal	64

Figure 5.11 Macro Statistics of the (a) Original ACS Design. (b) Double State ACS Design.

Fig. 5.11 (a) shows the macro statistics of the original ACS design and (b) the double state ACS design generated by Xilinx ISE 6.1. Compared with the original design, the double state design requires twice more adders, SM registers (the 13-bit register), multiplexers, and 64 more comparators to compare 128 resulting SMs for traceback.

Figure 5.12 shows the waveform of the double state ACS design. It is similar to the original ACS design. The only difference is that the endstate of the double state design may be the same state of the original design or it may be the state which only differs in the MSB from the endstate of the original design.

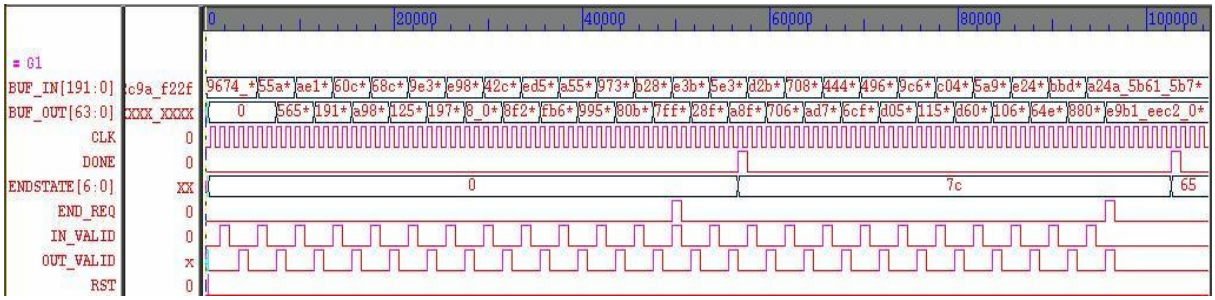


Figure 5.12: Waveform of Double State ACS Design.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, our original target is to implement the FEC scheme of IEEE 802.16a wireless communication standard on the Innovative Integration's Quixote DSP baseboard, which houses the Texas Instruments TMS320C6416 DSP chip. We optimize the FEC scheme to achieve real-time operation. However, due to the malfunction of the Quixote board, we are not able to actually transmit data between the DSP board and the host PC. We thus can only simulate the modified FEC schemes on the TI DSP simulator, which is included in the TI code composer studio. Therefore, the simulation profile shown in this thesis may not match the exact profile obtained from the DSP emulator operated on Quixote board. However, we believe that the algorithm/program modifications we have done based on the DSP simulator are also valid on the actual Quixote board, and the improving rate should be similar.

In the previous chapters, we first introduce the FEC standard of IEEE 802.16a briefly. We then describe how we implement it and explain the algorithms we use. Next, we introduce our implementation environment to show the available hardware resources. Afterward, we apply some useful techniques to speed up the software. They are either tuning the program to make the compiler work more efficiently or modifying the programs based on the algorithms either proposed by ourselves or proposed by the other

researchers to improve the processing rate of the RS encoder and decoder, and the convolutional decoder (Viterbi decoder). With DSP optimization, we have achieved an average processing rate of 7984 Kbits/sec on the FEC encoder with I/O included or 9072 Kbits/sec without I/O and an average processing rate of 750 Kbits/sec on the FEC decoder with I/O included or 960Kbits/sec without I/O. (Once again, the actual processing rate on the DSP emulator operated on the Quixote board may be different from these results.)

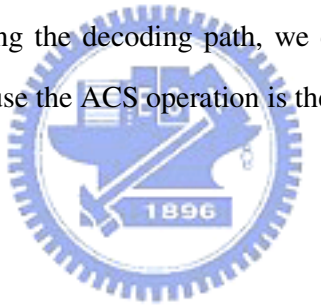
To further improve the processing rate of the FEC decoder, we need to accelerate the ACS operation in the Viterbi decoder, which is the bottleneck of our implementation. We put the ACS operation on the Xilinx FPGA, which is located on the Quixote board as an assistant hardware resource. We have done two simulations to evaluate the processing rate of the ACS operation implemented on Xilinx FPGA. The first ACS design that simply makes the ACS units operate in parallel, can achieve a processing rate of 45M (64 states/sec). It is much faster than the original speed of 2M (64 states/sec) on DSP. The second ACS design based on double state trellis architecture can achieve a processing rate of 49M (64 states/sec). But the area it consumed is 1.8 times more than that of the first ACS design. In consideration of the limit on physical transmission bandwidth on Quixote board, the processing rate of ACS units beyond 32M (64 states/sec) does not result in actual data processing speed improvement. Therefore, if we want to implement the ACS units on Xilinx FPGA, the first ACS design can be accepted to achieve a good balance of area and speed.

6.2 Future Work

As mentioned above, the communication mechanism of the Quixote DSP baseboard does not work properly. Hence, our simulations are all performed on the TI C64xx DSP simulator on PC. Therefore, when the communication mechanism works,

the streaming mechanism shall be concatenated to our FEC program to actually test the exact processing rate on the Quixote board.

As for the optimization tasks, there are still some functions can be further improved to accelerate the overall processing rate of our FEC program. For the RS decoder, the Chien search still consumes a lot of execution time if the data errors happen to occur in the last symbol of the codeword. If we can find a fast algorithm for Chien search, whose computational complexity is independent to the position of the errors, then the processing rate of the RS decoder can be much improved. For the Viterbi decoder, the ACS operation consumes a lot of execution time due to the need of expanding the entire trellis structure. But in fact only a small portion of the trellis structure is used to generate the decoding path. So, if we can find a modified Viterbi algorithm that does not expand the entire trellis structure, but it only expands a part of the trellis needed for generating the decoding path, we can accelerate the whole FEC decoder by a large factor because the ACS operation is the bottleneck of our program.



Bibliography

[1] IEEE Standard for local and metropolitan area networks, Part 16, Amendment 2.

[2] D. Wilson, "An Efficient Viterbi Decoder Implementation for the ZSP500 DSP Core," *Adv. DSP Dev.,* LSI Logic.

[3] I. S. Reed and X.-M. Chen, *Error-Control Coding for Data Networks*. Kluwer Academic Publishers, Dordrecht, 1999.

[4] C. Paar, "A new architecture for a parallel finite field multiplier with low complexity based on composite fields," *IEEE Trans. on Comp.*, vol. 45, pp. 856-861, Jul. 1996.

[5] C. Paar and G. Orlando, "A Super-Serial Galois Fields Multiplier for FPGAs and its Application to Public-Key Algorithms," *Seventh Annual IEEE Symp. on Field-Programmable Custom Computing Machines, FCCM '99*, pp. 232-239, Apr. 1999.

[6] E. Savas and C,. K. Koc, "Efficient Methods for Composite Field Arithmetic," *Elect. and Comp. Eng.*, Oregon State University, Dec. 1999.

[7] J.-S. Lin, "DSP Implementation and error performance study on speech source/channel coding," M.S. thesis, National Chiao Tung University, *Dep. of Elect. Eng.*, Hsinchu, Taiwan R.O.C., Jun. 2002.

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

[9] F. Tosato and P. Bisaglia, "Simplified Soft-Output Demapper for Binary Interleaved COFDM with Application to HIPERLAN/2," *IEEE Int. Conf. Commun. Conf. Rec., vol. 2, pp. 664-668, 2002.*

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

[11] Texas Instruments, TMS320C6000 CPU and Instruction Set Reference Guide. Literature Number: SPRU189F, Oct. 2000.

[12] Texas Instruments, TMS320C64x Technical Overview. Literature Number: SPRU396B, Jan. 2001.

[13] Texas Instruments, TMS320C6000 Programmer's Guide. Literature Number: SPRU198G, Aug. 2002.

[14] Innovative Integration, Quixote User's Manual. Jul. 2003.

[15] Innovative Integration, Quixote Architecture.

[16] Q. Zhuge, B. Xiao, and E. H.-M. Sha, "Code Size Reduction Technique and Implementation for Software-Pipelined DSP Applications," *ACM Tran. on Embedded Computing Systems*, vol. 2, pp. 590-613, Nov. 2003.

[17] J. Guajardo, "Itoh-Tsujii Inversion Algorithm," Ruhr-University, *Commun. Security Group*, Bochum, Germany, Jun. 2003.

[18] I. S. Reed, M. T. Shih, and T. K. Truong, "VLSI design of inverse-free Berlekamp-Massey algorithm," *Proc. Inst. Elect. Eng.*, vol. 138, pt. E, pp. 295-298, Sep. 1991.

[19] J. H. Jeng and T. K. Truong, "On Decoding of Both Errors and Erasures of a Reed-Solomon Code Using an Inverse-Free Berlekamp-Massey Algorithm," *IEEE Tran. on Commun.*, vol. 47, pp. 1488-1494, Oct. 1999.

[20] J. A. Heller and I. M. Jacobs, "Viterbi Decoding for Satellite and Space Communication," *IEEE Tran. on Commun. Tech.*, vol. 19, pp. 835-848, Oct. 1971.

[21] I. Lee and J. L. Sonntag, "A New Architecture for the Fast Viterbi Algorithm," *IEEE Tran. on Commun.*, vol. 51, pp. 1624-1628, Oct. 2003.

作者簡歷

李仰哲，西元 1980 年 8 月 26 日生於台北市。2002 年畢業於國立交通大學電子工程學系。同年進入國立交通大學電機資訊學院電子工程研究所電路與系統組碩士班，從事通道編碼系統之相關研究。2004 年 6 月取得碩士學位，論文題目為「IEEE 802.16a 標準之前向誤差改正編碼於數位訊號處理器平台上之實現與最佳化」。研究範圍與興趣包括：通道編碼、通訊系統。