國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

AMR 編碼及 IEEE 802.16a 標準之 Reed-Solomon 解碼器於數位訊號處理器之實現

DSP Implementation of AMR Speech Coding and the Reed-Solomon Decoder in IEEE 802.16a Standard

研 究 生：陳志楹

指導教授：杭學鳴 博士

中 華 民 國 九 十 四 年 六 月

AMR 編碼及 IEEE 802.16a 標準之 Reed-Solomon 解碼器於數位訊號處理器之實現

研究生: 陳志楹　　　　　　　　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

近年來，多媒體與無線通訊已成為市場上非常重要的發展趨勢，IEEE 802.16a 通訊標準主要在於實現無線網路上能夠傳輸高品質的多媒體的目標，在本篇論文中，我們將會實現語音與 Reed-Solomon 編碼機制於 TI DSP 平台上。

本篇論文的重點之一，在於多媒體編碼的部分，我們將討論第三代無線通訊系統中所採用的語音標準「適應性多速率編碼(AMR)」，它提供了多樣的編碼模式來因應各種通道所產生的影響；另一個重點為 IEEE 802.16a 無線通訊標準中前向誤差改正編碼機制的部分，由於 Reed-Solomon 編碼高度的修正能力，因而被 IEEE 802.16a 採用於前向誤差改正編碼的程序之一。

在論文中，首先我們將簡單描述 AMR 語音標準與 IEEE 802.16a FEC 部分的演算法與架構，並且針對數位訊號處理器(DSP)平台的特性，改善 AMR 語音編碼與 Reed-Solomon 解碼器的執行效率，進而實現於 DSP 平台上。我們的實現平台核心為德州儀器公司所發展的數位訊號處理器，程式經過改進後，AMR 語音編碼器在 DSP 平台上可以達到每秒 22.78K 位元的處理速率，解碼器則可達到每秒 31.84K 位元，而在 IEEE 802.16a 中 Reed-Solomon 解碼器的部分，在 DSP 平台上甚至可以達到每秒 176.4K 位元的處理速度，但這些測試數據都包括電腦與 DSP 之間資料傳輸所花費的時間，若扣除後將會更加快速。此外，我們也對原先的程式加以比

較，在 AMR 編碼方面進步了 65.94%，在 Reed-Solomon 解碼器方面也比原先實現的版本進步了 96.44%。

DSP Implementation of AMR Speech Coding and the Reed-Solomon Decoder in IEEE 802.16a Standard

Student: Chih-Ying Chen   *Advisor: Dr. Hsueh-Ming Hang*

Department of Electronics & Institute of Electronics
National Chiao Tung University

Abstract

Multimedia and wireless communication have been two very important trends in the recent years. Transmitting high quality multimedia data over wireless channel is the target of the IEEE 802.16a standard. In this thesis, we will implement a speech coding scheme and a Reed-Solomon coding scheme on TI DSP.

One focus of this thesis is Adaptive Multi Rate (AMR), the speech coding standard of 3GPP. It provides various coding modes match the channel error rates. Another focus of this thesis is the Forward Error Correction (FEC) scheme of the IEEE 802.16a wireless communication standard. The Reed-Solomon coding is adopted by the IEEE 802.16a because of its high capability of correcting errors.

We first describe the basic structure and algorithm of the AMR speech coding and the FEC in IEEE 802.16a. Then, we adopt and modify fast scheme to accelerate the programs of the AMR speech codec and Reed-Solomon decoder to match the architecture of the DSP baseboard. We further implement them on the DSP platform, which contains the Texas Instruments (TI) TMS320C6416 digital signal processor

(DSP). The processing rate of the AMR codec on the DSP platform reaches 22.78 Kbytes/sec for the encoder and 31.84 Kbytes/sec for the decoder. And the Reed-Solomon decoder reaches up to 176.4 Kbytes/sec. Moreover, those processing rates includes of the data transfer time between the host and the DSP board. It can be much faster if the data transfer time is excluded. In addition, the AMR speech codec after our improvement is 65.94% faster for the encoder and 61.31% faster for the decoder than the original one. The Reed-Solomon decoder is 96.44% faster than the original one.

誌謝

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,使我能夠順利完成這篇論文。在研究的過程中，有停滯不前的時候也有迷惘的時候，老師總是以關心和體諒代替苛責，適時的給予指導，促使我能夠克服在研究中所遇到的瓶頸；而在有所突破時，也不忘給予勉勵。除了與研究相關的課題，老師也不斷地鼓勵我們涉獵其它相關領域，厚實未來作進一步研究的基礎。除此之外，老師也總是能夠關心並體諒我們在生活上的種種問題,使我能夠在研究與生活中取得良好的平衡。此外，還要感謝張錫嘉老師，在研究上給予許多的協助並引領我正確的研究方向，讓我受益良多，同時也使我的研究得以順利進行，在此特地感謝老師如此耐心的指教。

在這裡也要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，讓我在研究中不虞匱乏。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，讓彼此能夠分享研究生活的點點滴滴、歡樂與苦澀。感謝楊政翰與陳繼大學長，在研究的過程不吝提供經驗與鼓勵，也感謝蔡家揚學長，適時提供技術上的支援，解決了許多我在研究上遇到的困難，也讓我學到解決各類問題的正確方式，另外還要感謝王盈閔、董景中、陳昱昇與洪朝雄等同學百忙之中提供研究工作與課業上的協助，使得論文能夠順利的進行。

最後，要感謝的是我的家人，不論在生活上或求學上都給了我最大的鼓勵與支持，讓我能夠心無旁騖的從事研究工作，遇到挫折時更讓我能夠有勇氣去面對。沒有家人在背後的付出，也就沒有今天的我，在此，謹獻上最高的謝意與歉意。

謝謝所有陪我走過這一段歲月的師長、同儕與家人，謝謝！

Content

1 Introduction 1

2 Adaptive Multi-Rate of Speech Coding 4

2.1 The Overview of AMR....4
2.2 Principles of the Encoder....6
2.2.1 Pre-processing....7
2.2.2 Linear Prediction....7
2.2.2.1 Windowing and auto-correlation....7
2.2.2.2 Levinson-Durbin algorithm....10
2.2.2.3 LP to LSP Conversion....10
2.2.2.4 Monitoring resonance in the LPC spectrum....12
2.2.3 Open-loop pitch analysis....13
2.2.4 Impulse response computation (all modes)....13
2.2.5 Target signal computation (all modes)....14
2.2.6 Adaptive codebook....14
2.2.6.1 Adaptive codebook search....14
2.2.6.2 Adaptive codebook gain control (all modes)....16
2.2.7 Algebraic codebook....16
2.2.7.1 Algebraic codebook structure....17
2.2.7.2 Algebraic codebook search....17
2.2.8 Quantization of adaptive and fixed codebook gains....19
2.2.8.1 Adaptive codebook gain limitation....19
2.2.8.2 Quantization of codebook gains....19
2.2.9 Memory update (all modes)....21
2.3 Functional description of the decoder....22
2.3.1 Decoding and speech synthesis....22
2.3.2 Post-processing....25
2.3.2.1 Adaptive post-filtering (all modes)....25
2.3.2.2 High-pass filtering and up-scaling....26
2.4 Bit Allocation....27

3 Overview of IEEE 802.16a FEC Scheme 29
3.1 Introduction to IEEE 802.16a Standard 29
3.2 IEEE 802.16a FEC Specifications 30
3.2.1 Randomizer 31
3.2.2 Forward Error Correction Coding 32
3.2.2.1 Reed-Solomon Code Specification 34
3.2.2.2 Convolutional Code Specification 34
3.2.2.3 Interleaver 36
3.3 Implementation Issues of the FEC Scheme 37
3.3.1 Reed-Solomon Code 37
3.3.1.1 Encoding of Shortened and Punctured Reed-Solomon Codes 37
3.3.1.2 Decoding of Shortened and Punctured Reed-Solomon Codes 40
3.3.2 Convolutional Code 43
3.3.2.1 Encoding of Punctured Convolutional Code 43
3.3.2.2 Viterbi Decoding of Punctured Convolutional Code 44
3.3.2.3 Bit Interleaved Soft Decision Viterbi Decoding 48
3.3.2.4 Viterbi Decoding of Tail-Biting Convolutional Code 50
3.3.2.5 The Butterfly Structure in the Trellis Diagram 50

4 DSP Implementation Environment 52
4.1 The DSP Chip 52
4.1.1 Central Processing Unit 55
4.1.2 Memory 56
4.1.3 Peripherals 57
4.2 The DSP Baseboard 58
4.3 DSP Transmission Mechanism 59
4.4 Features of TI TMSC6000 Family DSP for Optimization 62
4.4.1 Code Development Flow 62
4.4.2 Pipeline Structure of the TI TMSC6000 Family 63
4.4.3 Software Pipelining 65
4.4.4 Program-Level Optimization 68

5 Implementation and Acceleration of AMR Speech Coding on TI DSP Platform 70
5.1 AMR Codec Acceleration 71
5.1.1 AMR Code Profile 71
5.1.2 Acceleration by Using the Intrinsics 75
5.1.3 Compiler Level Improvement 80
5.2 AMR Codec on C64x DSP Platform 82

5.2.1 Structure of AMR Implementation 82
5.2.2 Execution Flow of AMR Implementation 83
5.2.3 Performance Analysis 88
5.2.3.1 AMR Encoder Performance Analysis 89
5.2.3.2 AMR Decoder Performance Analysis 91

6 Implementation and Acceleration of 802.16a Reed-Solomon Decoder on TI DSP Platform 94

6.1 Acceleration on Reed-Solomon Decoder 95
6.1.1 Profiling the Original RS Decoder 95
6.1.2 Modifications of RS Decoder 97
6.1.2.1 Syndrome Computation Improvement 97
6.1.2.2 Chien Search Improvement 99
6.1.3 Performance Analysis 101
6.2 Remainder Decoding Algorithm for RS Decoder 104
6.2.1 Remainder Decoding Algorithm 105
6.2.2 Program Flow and Performance Analysis 107
6.3 DSP Implementation of Reed-Solomon Decoder and Viterbi Decoder 112
6.3.1 Structure of RS Decoder and Viterbi Decoder Implementation 112
6.3.2 Execution Flow of RS Decoder and Viterbi Decoder 112
6.3.2.1 DSP Program Flow for RS Decoder 112
6.3.2.2 DSP Program Flow for Viterbi Decoder 115
6.3.3 Performance Analysis 115

7 Conclusions and Future Work 117

7.1 Conclusion 117
7.2 Future Work 118

Bibliography 120

List of Figures

Figure 2.1 Simplified block diagram of the CELP speech synthesis model 3
Figure 2.2 Simplified block diagram of the adaptive multi-rate encoder.........................6
Figure 2.3 LP analysis windows 9
Figure 2.4 Simplified block diagram of the adaptive multi-rate decoder.......................23

Figure 3.1 IEEE local and metropolitan area networks standards family 30
Figure 3.2 Channel coding structure in transmitter side (top) and receiver side (bottom) 31
Figure 3.3 PRBS for Data Randomization 31
Figure 3.4 Creation of OFDMA randomizer initialization vector.................................32
Figure 3.5 Forward Error Correction structure in transmitter side (left) and receiver side (right) 33
Figure 3.6 Convolutional Encoder of Rate 1/2..35
Figure 3.7 Block Diagram of the RS Encoder Program 39
Figure 3.8 The Linear Feedback Shift Register Structure of RS Encoder 39
Figure 3.9 Block Diagram of a Conventional RS Encoder 40
Figure 3.10 Block Diagram of the RS Decoder Program...42
Figure 3.11 Syndrome Computation Circuit 42
Figure 3.12 Block Diagram of the Convolutional Encoder Program 44
Figure 3.13 State Transition Diagram Example 45
Figure 3.14 Trellis Diagram Example for a Viterbi Decoder 46
Figure 3.15 Survivor path of the Trellis Diagram 47
Figure 3.16 Block Diagram of the Viterbi Decoder Program..47
Figure 3.17 Structure of the Viterbi Algorithm 47
Figure 3.18 Partition of the 16-QAM Constellation...49
Figure 3.19 Block Diagram of the Suboptimal Tail-Biting Viterbi Decoder..................50
Figure 3.20 Butterfly Structure Showing Branch Cost Symmetry 51

Figure 4.1 The Block Diagram of TMS320C6x DSP Chip..54
Figure 4.2 The TMS320C64x DSP Chip Architecture and Comparison with Ancient TMS320C62x/C67x Chip..54
Figure 4.3 Innovative Integration's Quixote DSP Baseboard Card..............................58
Figure 4.4 The Architecture of Quixote Baseboard..59
Figure 4.5 Block Diagram of DSP Streaming Mode...61
Figure 4.6 Code Development Flow 63
Figure 4.7 (a) The Original Loop. (b) The Loop After Applying Software Pipelining..65
Figure 4.8 (a) Execution Record of the Original Loop. (b) Execution Record of the Software Pipelined Loop 66

Figure 5.1 Structure of AMR Speech Codes Implementation on the Host and DSP 83
Figure 5.2 (a) Graphical Interface of the AMR Encoder Implementation. (b) A Snapshot of Running the Program...85
Figure 5.3 (a) Graphical Interface of the AMR Decoder Implementation. (b) A Snapshot of Running the Program...86
Figure 5.4 the Flowchart of the AMR Encoder Implementation.................................87

Figure 6.1 the C Code of the Syndrome Computation in the Lee Decoder....................98
Figure 6.2 the Plot of the Decoding Cycle versus SNR 103
Figure 6.3 the Plot of the Correct Decoding Ratio versus SNR.................................104
Figure 6.4 Implementation of LFSR with the Intrinsics...110
Figure 6.5 the Interface of our RS Decoder implementation 113
Figure 6.6 the Flowchart of our RS Decoder Implementation 114
Figure 6.7 the Interface of the Viterbi Decoder Implementation................................115

List of Table

Table 2.1 Bit allocation of the AMR coding algorithm for 20ms frame 28

Table 3.1 Mandatory Channel Coding per Modulation..34
Table 3.2 The Inner Convolutional Code with Puncturing Configuration 35
Table 3.3 Bit Interleaved Block Sizes and Modulo...36

Table 4.1 Completing Phase of Different Type Instructions 64

Table 5.1 Profile of AMR Encoder Provided by 3GPP 73
Table 5.2 Profile of the Top Ten Encoder Functions Called Most (Except for the Functions Containing Value Assignment Only) 74
Table 5.3 Profile of AMR Codec Arithmetic Functions (Not Counted are Value Assignments or Function Calling Only)...76
Table 5.4 Profile of AMR Arithmetic Functions Listed in Table 5.3 after Acceleration 79
Table 5.5 Profile of Different Improved Versions of AMR Encoder.............................80
Table 5.6 Profile of Different Improved Versions of AMR Decoder.............................81
Table 5.7 Code Size of the AMR Encoder for Different Acceleration Level 88
Table 5.8 Code Size of the AMR Decoder for Different Acceleration Level.................88
Table 5.9 Execution Time of the DSP Implementation under Different Source Rate for Each Test Sequence 89
Table 5.10 Execution Time of the DSP Implementation under Different Source Rate for Each Test Sequence (ms/frame: the Processing Time for one frame, %: Improvement Percentage)..90
Table 5.11 Execution Time of the DSP Implementation under Different Source Rate for Each Test Sequence (the List Representation is the Same as Table 5.10)90
Table 5.12 Execution Time of the DSP Implementation under Different Source Rate for Each Test Sequence...91
Table 5.13 Execution Time of the DSP Implementation under Different Source Rate

for Each Test Sequence (ms/frame: the Processing Time for one frame, %: Improvement Percentage) 92
Table 5.14 Execution Time of the DSP Implementation under Different Source Rate for Each Test Sequence (the List Representation is the Same as Table 5.13) 92

Table 6.1 Profile of the Lee RS Decoder 96
Table 6.2 Improvement of Syndrome Somputation 99
Table 6.3 Profile Chien Search without the Intrinsics and Compiler Optimization 101
Table 6.4 Profile Chien Search with _gmpy4 and file-Level Optimization 101
Table 6.5 Simulation Profile for RS Decoder 102
Table 6.6 the Decoding Ratio and Cycle under the Channel with Different SNR 103
Table 6.7 Comparison of the Remainder Decoding Algorithm and the Lee Decoder (without the Intrinsics) 107
Table 6.8 Profile of the Improved Remainder Decoding Algorithm 108
Table 6.9 Profile of our Implementation for RS Decoder and Viterbi Decoder 115

Chapter 1

Introduction

Digital wireless transmission of multimedia contents is one of the important trends in the consumer electronics field in the present. Due to the demand for wireless communication of multimedia contents, the high compression ratio with high quality is an important issue for multimedia transmission. Multimedia service contains many different types of contents such as data, audio, video, image, and the traditional speech. These services would have poor quality if they are overly compressed with non-efficient source coding or cannot be recovered from the errors introduced by the noisy channel. According to channel condition, it is desirable to adjust the source and channel coding rate to provide a better overall performance.

The international organization of 3GPP has adopted the concept above into its standard. For the traditional speech coding, it defines a set of technical specifications, which include the codecs of G.723.1 and AMR (Adaptive Multi Rate). Both G.723.1 and AMR are CELP based coders. However, AMR has a better speech quality than G.723.1 at about similar data rate. AMR also offers multiple modes for joint source/channel coding, providing flexibility for different QoS(Quality of Service).

For the efficient channel coding, 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 technique. 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 transmission lines. However, one major problem is that the transmission channel is not noisefree. The transmission signals are easily interfered and distorted by several different types of noise sources such as the crowd traffic, bad weather, the obstacle of buildings, etc. To improve the robustness of the wireless communication against the noisy channel condition, the FEC (Forward–Error-Correcting Coding) and FED (Forward–Error-Correcting Decoding) mechanism is necessary to reduce channel errors and is adopted by almost every commercial communication standards, including the IEEE 802.16a. Our study focuses on the Reed-Solomon coding included in the FEC/FED of the IEEE 802.16a standard, which specifies the air interface of fixed broadband wireless access systems for providing multiple accesses. The Reed-Solomon coding adds the resistance directly to the front end multimedia from the channel efforts. It has been wildly used and investigated because of its high capability of correcting both the random and burst errors and its efficient decoding algorithm of existence.

In this thesis, we implement the AMR speech codec and the Reed-Solomon coding scheme of IEEE 802.16a standard on II Quixote DSP/FPGA board. We first review the algorithm of the AMR codec and the whole FEC/FED scheme of IEEE 802.16a in detail. Then, we simulate their procedure by the C codes to accelerate their execution efficiency. Finally, we implement the AMR codec and the Reed-Solomon coding algorithm on our DSP platform. The AMR encoder can reach a processing rate of 14.05 ms/frame, and the AMR decoder can reach a processing rate of 2.43 ms/frame. The Reed-Solomon decoder even achieves a processing rate of 176.4 Kbytes/s after our improvement and implementation.

In Chapter 2, the concept and the major algorithm blocks of AMR are introduced. Due to the limited space, we only present the issues that are important for comprehending the structure of speech compression, such as ACELP model, LSP, and codebook formation.

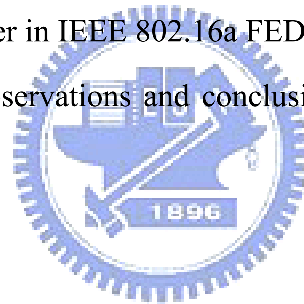
In Chapter 3, we briefly introduce the forward error correction scheme of the IEEE 802.16a standard. Furthermore, we also describe the algorithm to be implemented.

In Chapter 4, we give a brief description of our implementation environment; it includes both the II's Quixote DSP baseboard, its transmission mechanism between host PC and target DSP, and the techniques used to accelerate the programs.

In Chapter 5, we profile and accelerate the AMR codec program before implementing on the TI C6x DSP. We first describe the technique used to accelerate our C code step by step. Then the structure and the execution flow of its DSP implementation shall be introduced in detail.

In Chapter 6, we first discuss the original Reed-Solomon program required for speeding up. Secondly, the acceleration steps we have done on the Reed-Solomon decoder are discussed in detail. Finally, the DSP implementation of the improved program and the Viterbi decoder in IEEE 802.16a FED scheme is also described.

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



Chapter 2

Adaptive Multi-Rate of Speech Coding

2.1 Overview of AMR

AMR (Adaptive Multi-Rate) is a new concept for achieving a high speech quality while maintaining an efficient spectrum usage. A trade-off between speech quality and system capacity can be achieved for a variety of radio channel and operating conditions. It is a successful joint source/channel combined codec standard. The system allows channel mode (HR or FR) and codec mode (combination of speech and channel bit-rates) to vary in order to suit traffic and channel conditions. The channel mode consists of two different transmission bit rate: 22.8 kbit/s (Full rate) and 11.4 kbit/s (Half rate) and can be switched in order to increase channel capacity, replacing for example one full-rate channel with two half-rate channels, while maintaining a certain lower limit for the speech quality. These AMR handovers occur much less frequently than the codec mode changes, probably a few times per minutes [1].

For each channel mode (HR or FR), the codec mode, i.e. bit partitioning between speech and channel bit-rates, can be varied rapidly to track the channel error rate or the channel's C/I. The changes must occur quite immediately (several times a second), with no perceptible speech degradation. The process is equivalent to Link Adaptation. Besides the basic source and channel codec for speech signal payload, the AMR system

concept further includes channel state tracking and in-band transmission of adaptation data.

The AMR coder consists of eight source codecs with bit-rates of 12.2, 10.2, 7.95, 7.40, 6.70, 5.90, 5.15 and 4.75 kbit/s. The codec is based on the code-excited linear predictive (CELP) coding model. In this model, the excitation signal at the input of the short-term LP synthesis filter is constructed by adding two excitation vectors from adaptive and fixed (innovative) codebooks. The speech is synthesized by feeding the two properly chosen vectors from these codebooks through the short-term synthesis filter. The optimum excitation sequence in a codebook is chosen using an analysis-by-synthesis search procedure in which the error between the original and synthesized speech is minimized according to a perceptually weighted distortion measure. The structure of the CELP speech synthesis model is shown in figure 2.1 [2][3]. For details, more information can be obtained in [14][15][16].

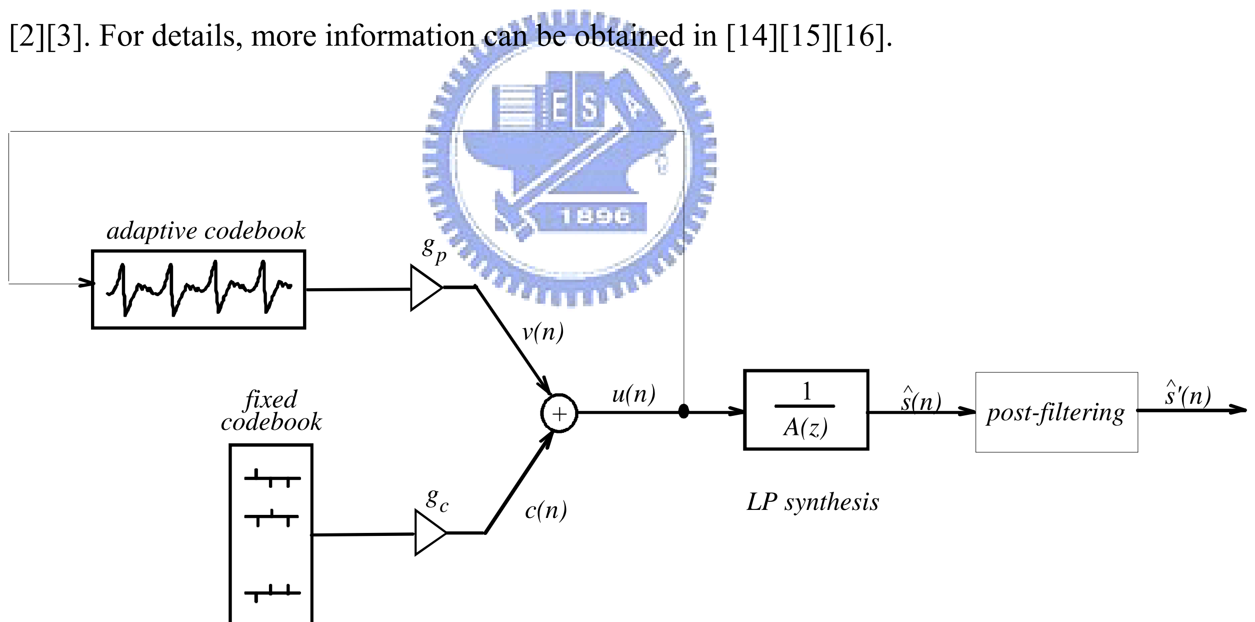


Figure 2.1: Simplified block diagram of the CELP speech synthesis model.

2.2 Principles of the Encoder

The AMR coder operates on speech frames of 20ms corresponding to 160 samples at the sampling frequency of 8000 sample/s. At each 160 speech samples, the speech signal is analyzed to extract the parameters of the CELP model (LP filter coefficients, adaptive and fixed codebooks' indices and gains). A 10th order linear prediction (LP), or short-term, synthesis filter is used which is given by [3]:

$$H(z) = \frac{1}{\hat{A}(z)} = \frac{1}{1 + \sum_{i=1}^{m} \hat{a}_i^{-i}} \tag{2.1}$$

where $\hat{a}_i, i = 1,...,m,$ are the (quantified) linear prediction (LP) parameters, and m=10 is the predictor order. The long term, or pitch, synthesis filter is given by:

$$\frac{1}{B(z)} = \frac{1}{1 - g_p z^{-T}}, \tag{2.2}$$

where T is the pitch delay and g_p is the pitch gain. The pitch synthesis filter is implemented using the so-called adaptive codebook approach. Then the following operations are repeated for each sub-frame:

The target signal x(n) is computed by filtering the LP residual through the weighted synthesis filter W(z)H(z) with the initial states of the filters having been updated by filtering the error between LP residual and excitation. The impulse response, h(n) of the weighted synthesis filter is then computed.

Closed-loop pitch analysis is then performed (to find the pitch lag and gain), using the target x(n) and impulse response h(n), by searching around the open-loop pitch lag. Fractional pitch with 1/6th or 1/3rd of a sample resolution (depending on the mode) is used. The target signal x(n) is updated by removing the adaptive codebook contribution (filtered adaptive codevector), and this new target, $x_2(n)$, is used in the fixed algebraic codebook search (to find the optimum innovation).

The gains of the adaptive and fixed codebook are scalar quantified with 4 and 5 bits respectively or vector quantified with 6-7 bits (with moving average (MA)

prediction applied to the fixed codebook gain). The different functions of the encoder is presented in figure 2.2.

2.2.1 Pre-processing

Two pre-processing functions are applied prior to the encoding process: high-pass filtering and signal down-scaling. Down-scaling consists of dividing the input by a factor of 2 to reduce the possibility of overflows in the fixed-point implementation. The high-pass filter serves as a precaution against undesired low frequency components with a cut off frequency of 80Hz.

2.2.2 Linear Prediction

The LP analysis and quantization for the 12.2 kbit/s mode follows that of the GSM EFR coder, i.e. two LP filters are computed for each frame. These filters are jointly quantized with split matrix quantization (SMQ) of 1st order MA-prediction LSF residuals. For all the other modes, one LP filter is estimated per frame. Split VQ (SVQ) of 1st order MA-prediction LSF residuals are performed with 3 subvectors of dimension 3, 3, and 4.

2.2.2.1 Windowing and auto-correlation

For 12.2 kbit/s, LP analysis is performed twice per frame using two different 30ms asymmetric windows. Asymmetric windows have been proved to own better quality-delay performance then symmetric window [4]. The first window has its weight concentrated at the second subframe and it consists of two halves of Hamming windows with different size.

Figure 2.2: Simplified block diagram of the adaptive multi-rate encoder

On the other hand, the second window has its weight concentrated at the fourth subframe and it consists of two parts: the first part is half a Hamming window and the second part is a quarter of a cosine function cycle [5]. No samples from future frames are used (no lookahead). A diagram of the two LP analysis windows is depicted in figure 2.3.

The auto-correlations of the windowed speech $s'(n), n = 0,...,239,$ are computed by:

$$r_{ac}(k) = \sum_{n=k}^{239} s'(n)s'(n-k), \quad k = 0,...,10, \tag{2.3}$$

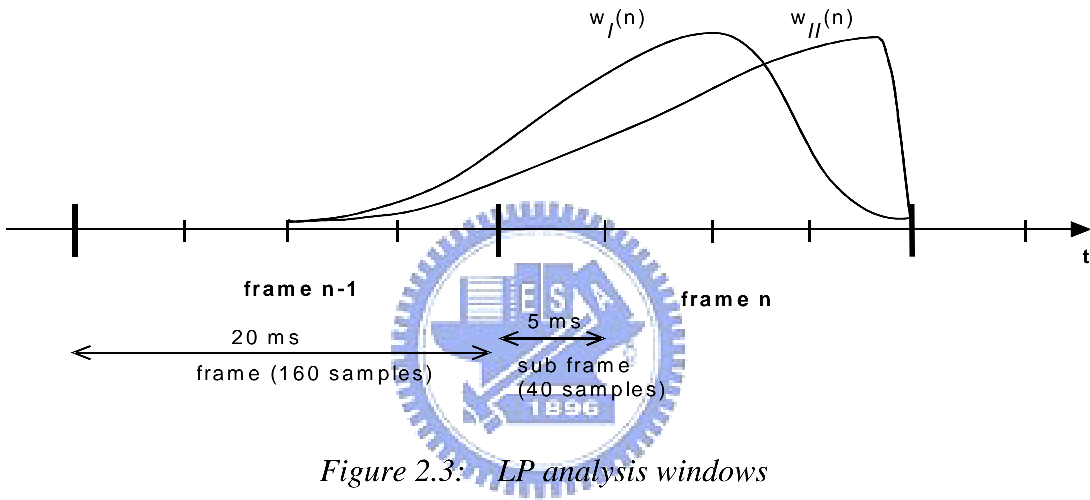


Figure 2.3: LP analysis windows

and a 60 Hz bandwidth expansion is used by lag windowing the auto-correlations using the window:

$$w_{lag}(i) = \exp\left[-\frac{1}{2}\left(\frac{2\pi f_0 i}{f_s}\right)^2\right], i = 1,...10, \tag{2.4}$$

where $f_0 = 60\ Hz$ is the bandwidth expansion. The expansion on the autocorrelation coefficients reduces the possibility of ill-condition in the Levinson algorithm (especially in fixed point). It also reduces the underestimation of the formant bandwidth, which could create undesirably sharp resonances. Further, $r_{ac}(0)$ is multiplied by the white noise correction factor 1.0001 which is equivalent to adding a noise floor at –40 dB. The operation reduces the possibility of ill-condition due to bandpass filtering of the input [6].

2.2.2.2 Levinson-Durbin algorithm

The modified auto-correlations are used to obtain the direct form LP filter coefficients $a_k, k = 1,...,10,$ by solving the set of equations.

$$\sum_{k=1}^{10} a_k r_{ac}^{'}(|i-k|) = -r_{ac}^{'}(i), \quad i = 1,...,10. \tag{2.5}$$

The set of equations is solved using the Levinson-Durbin algorithm.

$$
\begin{array}{l}
E_{LD}(0) = r_{ac}'(0) \\
\text{for } i = 1 \text{ to } 10 \text{ do} \\
\quad a_0^{(i-1)} = 1 \\
\quad k_i = -\left[\sum_{j=0}^{i-1} a_j^{(i-1)} r_{ac}'(i-j)\right] / E_{LD}(i-1) \\
\quad a_i^{(i)} = k_i \\
\quad \text{for } j = 1 \text{ to } i-1 \text{ do} \\
\quad \quad a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)} \\
\quad \text{end} \\
\quad E_{LD}(i) = (1 - k_i^2) E_{LD}(i-1) \\
\text{end}
\end{array}
$$

The final solution is given as $a_j = a_j^{(10)}, j = 1,...,10$. The LP filter coefficients are converted to the line spectral pair (LSP) representation for quantization and interpolation purposes.

2.2.2.3 LP to LSP Conversion

LP is not conducive to efficient quantization, because it has relatively high spectral sensitivity. On the other hand, LSP has intimate relationship with the formant frequencies. Also LSP's can be quantized taking into account spectral features known to be important in perceiving speech signals.

For the 10th order LP filter, the LSPs are defined as the roots of the sum and difference polynomials [3]:

$$F_1'(z) = A(z) + z^{-11} A(z^{-1}) \tag{2.6}$$

and

$$F_2'(z) = A(z) - z^{-11}A\left(z^{-1}\right) \tag{2.7}$$

respectively. It can be proven that all roots of these polynomials are on the unit circle and they alternate each other. $F_1'(z)$ has a root $z = -1\ (\omega = \pi)$ and $F_2'(z)$ has a root $z = 1\ (\omega = 0)$. To eliminate these two roots, we define the new polynomials:

$$F_1(z) = F_1'(z)/\left(1 + z^{-1}\right) = \prod_{i=1,3,\ldots,9}(1 - 2q_i z^{-1} + z^{-2}) \tag{2.8}$$

and

$$F_2(z) = F_2'(z)/\left(1 - z^{-1}\right) = \prod_{i=2,4,\ldots,10}(1 - 2q_i z^{-1} + z^{-2}) \tag{2.9}$$

where $q_i = \cos(\omega_i)$ with ω_i being the line spectral frequencies (LSP) and they satisfy the ordering property $0 < \omega_1 < \omega_2 < \ldots < \omega_{10} < \pi$. We refer to q_i as the LSPs in the cosine domain. Since both polynomials $F_1(z)$ and $F_2(z)$ are symmetrical, it means only the first 5 coefficients of each need to be computed. The coefficients of these polynomials are found by the recursive relation(for I=0 to 4):

$$\begin{aligned} f_1(i+1) &= a_{i+1} + a_{m-i} - f_1(i) \\ f_2(i+1) &= a_{i+1} - a_{m-i} + f_2(i) \end{aligned} \tag{2.10}$$

where m=10 is the predictor order. The LSPs are found by evaluating the polynomials $F_1(z)$ and $F_2(z)$ at 60 points equally spaced between 0 and π and checking for sign changes. A sign change signifies the existence of a root and the sign change interval is then divided 4 times to better track the root.

The Chebyshev polynomials are used to evaluate $F_1(z)$ and $F_2(z)$ [8]. In this method the roots are found directly in the cosine domain $\{q_i\}$. The polynomials $F_1(z)$ and $F_2(z)$ evaluated at $z = e^{j\omega}$ can be written as:

$$F(\omega) = 2e^{-j5\omega}C(x),$$

with

$$C(x) = T_5(x) + f(1)T_4(x) + f(2)T_3(x) + f(3)T_2(x) + f(4)T_1(x) + f(5)/2, \quad (2.11)$$

where $T_m(x) = \cos(m\omega)$ is the mth order Chebyshev polynomials, and $f(i), i = 1,...5$ are the coefficients of either $F_1(z)$ or $F_2(z)$. The polynomial $C(x)$ is evaluated at a certain value of $x = \cos(\omega)$ using the recurrence relation:

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x) \quad for\ k = 2,3,..., \quad (2.12)$$

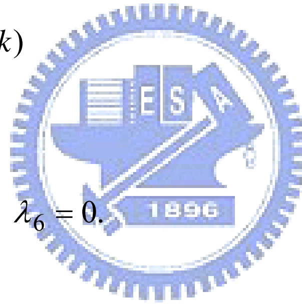
and trigonometric representation on [-1, 1]

$$T_N(x) = \cos(N \arccos(x)) \quad for \quad -1 \le x \le 1. \quad (2.13)$$

and then we obtain the following recursive relation:

for $k = 4$ down to 1
$\lambda_k = 2x\lambda_{k+1} - \lambda_{k+2} + f(5-k)$
end
$C(x) = x\lambda_1 - \lambda_2 + f(5)/2,$

with initial values $\lambda_5 = 1$ and $\lambda_6 = 0$.



2.2.2.4 Monitoring resonance in LPC spectrum (all modes)

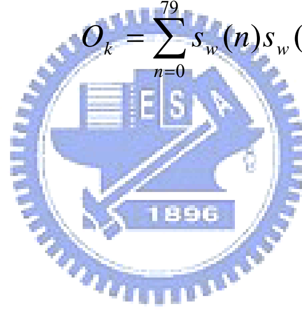
Resonances in the LPC filter are monitored to detect possible problem areas where divergence between the adaptive codebook memories in the encoder and the decoder could cause unstable filters in areas with highly correlated continuous signals. Typically, this divergence is due to channel errors. The monitoring of resonance signals is performed using unquantized LSPs $q_i, i = 1,...,10$. The LSPs are available after the LP to LSP conversion. The algorithm utilizes the fact that LSPs are closely located at a peak in the spectrum. First, two distances, $dist_1$ and $dist_2$, are calculated in two different regions, defined as $dist_1 = \min(q_i - q_{i+1}), i = 4,...8,$ and another as $dist_2 = \min(q_i - q_{i+1}), i = 2,3.$ Either of these two minimum distance conditions must be fulfilled to classify the frame as a resonance as a resonance frame and increase the

resonance counter. 12 consecutive resonance frames are needed to indicate possible problem condition, otherwise the LSP_flag is cleared.

2.2.3 Open-loop pitch analysis

Open-loop pitch analysis is performed in order to simplify the pitch analysis and confine the closed-loop pitch search to a small number of lags around the open-loop estimated lags. Open-loop pitch estimation is based on the weighted speech signal $s_w(n)$ which is obtained by filtering the input speech signal through the weighting filter $W(z) = A(z/\gamma_1)/A(z/\gamma_2)$. Open-loop pitch analysis is performed as follows. In the first step, 3 maxima of the correlation:

$$O_k = \sum_{n=0}^{79} s_w(n) s_w(n-k) \tag{2.14}$$



are found in the three ranges:

$i=3$: 18,...,35,

$i=2$: 36,...,71,

$i=1$: 72,...,143.

The retained maxima $O_{t_i}, i = 1,...3,$ are normalized by dividing by $\sqrt{\sum_{nw} s_w^2(n-t_i)}, i = 1,...3,$ respectively. The normalized maxima and corresponding delays are denoted by $(M_i, t_i), i = 1,...,3.$ The winner, T_{op} , among the three normalized correlations is selected by favouring the delays with the values in the lower range. This is performed by weighting the normalized correlations corresponding to the longer delays. This procedure of dividing the delay range into 3 clauses and favouring the lower clauses is used to avoid choosing pitch multiples.

2.2.4 Impulse response computation (all modes)

The impulse response, $h(n)$, of the weighted synthesis filter $H(z)W(z) = A(z/\gamma_1)/\left[\hat{A}(z)A(z/\gamma_2)\right]$ is computed each subframe. This impulse response is needed for the search of adaptive and fixed codebooks. The use of unquantized coefficients gives a weighting filter that matches better the original spectrum. The values of γ_1 and γ_2 modify the frequency response of the filter $W(z)$, and thereby the amount of noise weighting. It also deemphasizes the error at the formant regions of speech spectrum.

2.2.5 Target signal computation (all modes)

The target signal for adaptive codebook search is usually computed by subtracting the zero input response of the weighted synthesis filter $H(z)W(z)$ from the weighted speech signal $s_w(n)$. This is performed on a subframe basis. An equivalent procedure for computing the target signal is filtering of the LP residual signal $res_{LP}(n)$ through the combination of synthesis filter $1/\hat{A}(z)$ and the weighting filter $A(z/\gamma_1)/A(z/\gamma_2)$. After determining the excitation for the subframe, the initial states of these filters are updated by filtering the difference between the LP residual and excitation. The residual signal $res_{LP}(n)$ which is needed for finding the target vector is also used in the adaptive codebook search to extend the past excitation buffer. This simplifies the adaptive codebook search procedure for delays less than the subframe size of 40.

2.2.6 Adaptive codebook

2.2.6.1 Adaptive codebook search

Adaptive codebook search is performed on a subframe basis. It consists of performing closed-loop pitch search, and then computing the adaptive codevector by

interpolating the past excitation at the selected fractional pitch lag. The adaptive codebook parameters (or pitch parameters) are the delay and gain of the pitch filter. In the adaptive codebook approach for implementing the pitch filter, the excitation is repeated for delays less then the subframe length. In the search stage, the excitation is extended by the LP residual to simplify the closed-loop search.

Closed-loop pitch analysis is performed around the open-loop pitch estimates on a subframe basis. In the first (and third) subframe the range $T_{op} \pm 3$ is searched. For the other subframes, closed-loop pitch analysis is performed around the integer pitch selected in the previous subframes. The closed-loop pitch search is performed by minimizing the mean-square weighted error between the original and synthesized speech. This is achieved by maximizing the term [9]:

$$R(k) = \frac{\sum_{n=0}^{39} x(n) y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n) y_k(n)}}, \tag{2.15}$$

where $x(n)$ is the target signal and $y_k(n)$ is the past filtered excitation at delay k(past excitation with $h(n)$. Note that the search range is limited around the open-loop pitch. The convolution $y_k(n)$ is computed for the first delay $t_{\min}$ in the searched range, and for the other delays in the search range $k = t_{\min} + 1, \ldots, t_{\max}$, it is updated using the recursive relation:

$$y_k(n) = y_{k-1}(n-1) + u(-k)h(n), \tag{2.16}$$

where $u(n),\ n = -(143+11), \ldots, 39,$ is the excitation buffer. Note that in search stage, the samples $u(n),\ n = 0, \ldots, 39$, are not known, and they are needed for pitch delays less then 40. To simplify the search, the LP residual is copied to $u(n)$ in order to make the relation in equation (38) valid for all delays.

Once the optimum integer pitch delay is determined, the fractions with a step of 1/6 (or 1/3) around that integer are tested [10]. The fractional pitch search is performed by interpolating the normalized correlation in equation (37) and searching for its

maximum. The interpolation is performed using an FIR filter based on a Hamming windowed $\sin(x)/x$ function truncated and padded with zero. The filter has its cut-off frequencies (-3 dB) at 3600 Hz in the over-sampled domain.

Once the fractional pitch lag is determined, the adaptive codebook vector $v(n)$ is computed by interpolating the past excitation signal $u(n)$ at the given integer delay k and phase (fraction) t. The interpolation filter is also based on a Hamming windowed $\sin(x)/x$ function truncated and padded with zero. The filter has a cut-off frequency (-3dB) at 3600 Hz in the over-sampled domain.

The adaptive codebook gain is then found by:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)}, \quad \text{bounded by} \quad 0 \le g_p \le 1.2 \tag{2.17}$$

where $y(n) = v(n) * h(n)$ is the filtered adaptive codebook vector (zero state response of $H(z)W(z)$ to $v(n)$). The computed adaptive codebook gain is quantified using non-uniform scalar quantization in the range [0.0, 1.2].

2.2.6.2 Adaptive codebook gain control (all modes)

The average adaptive codebook gain is calculated if the LSP_flag is set and the unquantized adaptive codebook gain exceeds the gain threshold $GP_{th} = 0.95$. The average gain is calculated from the present unquantized gain and the quantized gains of the seven previous subframes. That is, $GP_{ave} = mean\{g_p(n), \hat{g}_p(n-1), \ldots \hat{g}_p(n-7)\}$, where n is the current subframe. If the average adaptive codebook gain exceeds the GP_{th}, the unquantized gain is limited to the threshold value and the GpC_flag is set to indicate the limitation.

2.2.7 Algebraic codebook

The algebraic codebook (innovation codebook) is for the secondary excitation computation. The vectors contained in the excitation forms a very important part in the CELP coding algorithm. They serve two main purposes: first, they provide the start-up information to the LTP memory, and this includes any sudden changes in the speech not adequately tracked by the LTP. Second, they supply the 'filling in' information that the LTP omitted. This is especially the case during unvoiced region. In the figure shows the general framework for innovation codebook driven by algebraic codes. Shaping function F can be fixed or changed dynamically as illustrated.

2.2.7.1 Algebraic codebook structure

The algebraic codebook structure is based on interleaved single-pulse permutation (ISPP) design. In this codebook, the innovation vector contains some non-zero pulses. All pulses can have the amplitudes +1 or –1. The 40 positions in a subframe are divided into a few tracks, where each track contains one or two pulses. Each pulse position in one track is encoded with some bits and the sign of the first pulse in the track is encoded with one bit. For two pulses located in the same track, only one sign bit is needed. This sign bit indicates the sign of the first pulse. The sign of the second pulse depends on its position relative to the first pulse. If the position of the second pulse is smaller, then it has opposite sign, otherwise it has the same sign then in the first pulse.

2.2.7.2 Algebraic codebook search

The algebraic codebook is searched by minimizing the mean square error between the weighted input speech and the weighted synthesized speech. The target signal used in the closed-loop pitch search is updated by subtracting the adaptive codebook contribution. That is,

$$x_2(n) = x(n) - \hat{g}_p y(n), n = 0,\ldots,39 \qquad (2.18)$$

where $y(n) = v(n) * h(n)$ is the filtered adaptive codebook vector and $\hat{g}_p$ is quantified adaptive codebook gain. If $\mathbf{c}_k$ is the algebraic codevector at index k, then the algebraic codebook is searched by maximizing the term :

$$A_k = \frac{(C_k)^2}{E_{Dk}} = \frac{(\mathbf{d}^t \mathbf{c}_k)^2}{\mathbf{c}_k^t \Phi \mathbf{c}_k}, \tag{2.19}$$

where $\mathbf{d} = \mathbf{H}^t \mathbf{x}_2$ is the correlation between the target signal $x_2(n)$ and the impulse response $h(n)$, H is a lower triangular Toepliz convolution matrix with diagonal $h(0)$ and lower diagonals $h(1),...,h(39)$, and $\Phi = \mathbf{H}^t \mathbf{H}$ is the matrix of correlations of $h(n)$. The vector d (backward filtered target) and the matrix Φ are computed prior to the codebook search. To simplify the search procedure, the pulse amplitudes are preset by the mere quantization of an appropriate signal $b(n)$. This is simply done by setting the amplitude of a pulse at a certain position equal to the sign of $b(n)$ at that position. $b(n)$ is the correlated signal corresponding to the $d(n)$.

Having preset the pulse amplitudes, the optimal pulse positions are determined using an efficient non-exhaustive analysis-by-synthesis search technique. In this technique, the term in equation (43) is tested for a small percentage of position combination. During iterations, at least one pulse is located in a position corresponding to the global maximum and one pulse is located in a position corresponding to one of the 4 local maxima.

A special feature incorporated in the codebook is that the selected codevector is filtered through an adaptive pre-filter $F_E(z)$ which enhances special spectral components in order to improve the synthesized speech quality. Here the filter $F_E(z) = 1/(1 - \beta z^{-T})$ is used, where T is the nearest integer pitch lag to the closed-loop fractional pitch lag of the subframe, and β is a pitch gain. β is given by the quantified pitch gain bounded by [0.0, 1.0]. Note that prior to the codebook search, the impulse response $h(n)$ must include the pre-filter $F_E(z)$. That is, $h(n) = h(n) - \beta h(n - T), \quad n = T,...,39.$ The fixed codebook gain is then found by:

$$g_c = \frac{\mathbf{x}_2^t \mathbf{z}}{\mathbf{z}^t \mathbf{z}} \tag{2.20}$$

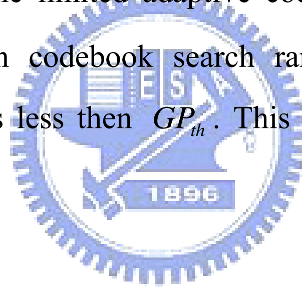
where $\mathbf{x}_2$ is the target vector for fixed codebook search and $\mathbf{z}$ is the fixed codebook vector convolved with $h(n)$,

$$z(n) = \sum_{i=0}^{n} c(i) h(n-i), n = 0, \ldots, 39. \tag{2.21}$$

2.2.8 Quantization of adaptive and fixed codebook gains

2.2.8.1 Adaptive codebook gain limitation

If the GpC_flag is set, the limited adaptive codebook gain is used in the gain quantization. The quantization codebook search range is limited to only include adaptive codebook gain values less then GP_{th}. This is performed in the quantization search for all modes.



2.2.8.2 Quantization of codebook gains

The fixed codebook gain quantization is performed using MA prediction with fixed coefficients. The 4th order MA prediction is performed on the innovation energy. Let $E(n)$ be the mean-removed innovation energy (in dB) at subframe n, and given by:

$$E(n) = 10\log\left(\frac{1}{N} g_c^2 \sum_{i=0}^{N-1} c^2(i)\right) - \overline{E}, \tag{2.22}$$

where $N = 40$ is the subframe size, and $c(i)$ is the fixed codebook excitation. $\overline{E}$ (in dB) is the mean of the innovation energy and a pre-defined value. The predicted energy is given by:

$$\tilde{E}(n) = \sum_{i=1}^{4} b_i \hat{R}(n-i) \tag{2.23}$$

where b_i are the MA prediction coefficients, and $\hat{R}(k)$ is the quantified prediction error at subframe k. The predicted energy is used to compute a predicted fixed codebook gain g'_c (by substituting $E(n)$ by $\tilde{E}(n)$ and g_c by g'_c). First, the mean innovation energy is found by:

$$E_I = 10\log\left(\frac{1}{N}\sum_{j=0}^{N-1} c^2(j)\right) \tag{2.24}$$

and then the predicted gain is found by:

$$g'_c = 10^{0.05\left(\tilde{E}(n)+\bar{E}-E_I\right)}. \tag{2.25}$$

A correction factor between the gain g_c and the estimated g'_c is given by:

$$\gamma_{gc} = g_c / g'_c. \tag{2.26}$$

Note that the prediction error is given by:

$$R(n) = E(n) - \tilde{E}(n) = 20\log(\gamma_{gc}). \tag{2.27}$$

The correction factor γ_{gc} is computed using a mean energy value $\bar{E}$. The correction vector γ_{gc} is quantified using an individual codebook or jointly vector quantized with adaptive codebook gain. If the correction factor γ_{gc} is quantized individually, the quantization table search is performed by minimizing the error

$$E_Q = \left(g_c - \hat{\gamma}_{gc} g'_c\right)^2. \tag{2.28}$$

Otherwise, The gain codebook search is performed by minimizing the square of the weighted error between original and reconstructed speech which is given by:

$$E = \left\| \mathbf{x} - g_p \mathbf{y} - g_c \mathbf{z} \right\|^2. \tag{2.29}$$

An adaptor based on the coding gain in the adaptive codebook decides if the coding gain is low. If this is the case, the correction factor codebook is searched once more minimizing a modified criterion in order to find a new quantized fixed codebook gain. The modified criterion is given by:

$$E_{\text{mod}} = (1-\alpha)\cdot\|\mathbf{c}\|^2\cdot\left(g_c - \hat{\gamma}_{gc}\cdot g_c'\right)^2 + \alpha\cdot\left(\sqrt{E_{res}} - \sqrt{E_{exc}}\right)^2 \tag{2.30}$$

where E_{res} and E_{exc} are the energy (the squared norm) of the LP residual and the total excitation, respectively. The criterion is searched with the already quantized adaptive codebook gain and the correction factor $\hat{\gamma}_{gc}$ that minimizes (60) is selected.

The balance α decides the amount of energy matching in the modified criterion. This factor is adaptively decided based on the coding gain in the adaptive codebook as computed by:

$$ag = 10\cdot\log_{10}\frac{\|\mathbf{res}_{LP}\|^2}{\|\mathbf{res}_{LP} - \mathbf{v}\|^2} \tag{2.31}$$

if the coding gain ag is less then 1 dB, the modified criterion is employed, except when an onset is detected. An onset is said to be detected if the fixed codebook gain in the current subframe is more then twice the value of the fixed codebook gain in the previous subframe. A hangover of 8 subframes is used in the onset detection so that the modified criterion is not used for the next subframes either if an onset is detected. The balance factor α is computed from the median filtered adaptive coding gain. The current and the ag-values for the previous 4 subframes are median filtered to get ag_m.

The α-factor is computed by:

$$\alpha = \begin{cases} 0 & ag_m > 2 \\ 0.5\cdot\left(1 - 0.5\cdot ag_m\right) & 0 < ag_m < 2 \\ 0.5 & ag_m < 0 \end{cases} \tag{2.32}$$

2.2.9 Memory update (all modes)

An update of the states of the synthesis and weighting filters is needed in order to compute the target signal in the next subframe. After the two gains are quantified, the excitation signal, $u(n)$, in the present subframe is found by:

$$u(n) = \hat{g}_p v(n) + \hat{g}_c c(n), n = 0,..,39. \tag{2.33}$$

The states of the filters can be updated by filtering the signal $res_{LP}(n) - u(n)$ (difference between residual and excitation) through the filters $1/\hat{A}(z)$ and $A(z/\gamma_1)/A(z/\gamma_2)$ for the 40-sample subframe and saving the states of the filters). A simpler approach which requires only one filtering is as follows. The output of the filter $1/\hat{A}(z)$ due to the input $res_{LP}(n) - u(n)$ is equivalent to $e(n) = s(n) - \hat{s}(n)$. So the states of the synthesis filter are given by $e(n), n = 30,...,39$. Updating the states of the filter $A(z/\gamma_1)/A(z/\gamma_2)$ can be done by filtering the error signal $e(n)$ through this filter to find the perceptually weighted error $e_w(n) = x(n) - \hat{g}_p y(n) - \hat{g}_c z(n)$. Since the signals $x(n)$, $y(n)$, and $z(n)$ are available, the states of the weighting filter are updated by computing $e_w(n)$ for $n = 30,...,39$.

2.3 Functional description of the decoder

The function of the decoder consists of decoding the transmitted parameters (LP parameters, adaptive codebook vector, adaptive codebook gain, fixed codebook vector, fixed codebook gain) and performing synthesis to obtain the reconstructed speech. The reconstructed speech is then post-filtered and upscaled. The signal flow at the decoder is shown in figure 2.5.

2.3.1 Decoding and speech synthesis

The received indices of LSP quantization are used to reconstruct the quantified LSP vectors. The interpolation is performed to obtain 4 interpolated LSP vectors (corresponding to 4 subframes). For each subframe, the interpolated LSP vector is converted to LP filter coefficient domain a_k, which is used for synthesizing the reconstructed speech in the subframe.

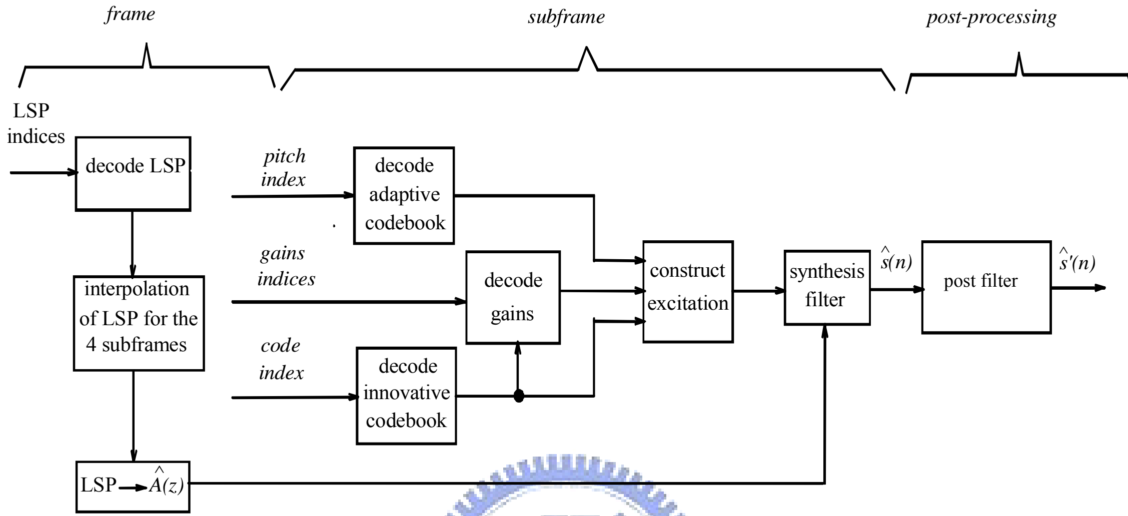


Figure 2.4: Simplified block diagram of the adaptive multi-rate decoder

The following steps are repeated for each subframe:

1. Decoding of the adaptive codebook vector: The received pitch index (adaptive codebook index) is used to find the integer and fractional parts of the pitch lag. The adaptive codebook vector $v(n)$ is found by interpolating the past excitation $u(n)$ (at the pitch delay) using the FIR filter.

2. Decoding of the innovative codebook vector: The received algebraic codebook index is used to extract the position and amplitudes (signs) of the excitation pulses and to find the algebraic codebook codevector $c(n)$. If the integer part of the pitch lag, T, is less than the subframe size 40, the pitch sharpening procedure is applied which translates into modifying $c(n)$ by $c(n) = c(n) + \beta c(n-T)$, where β is the decoded pitch gain, $\hat{g}_p$, bounded by [0.0,1.0] or [0.0,0.8], depending on mode.

3. Decoding of the adaptive and fixed codebook gains: In case of scalar quantization of the gains the received indices are used to readily find the quantified adaptive codebook gain, $\hat{g}_p$, and the quantified fixed codebook gain correction factor, $\hat{\gamma}_{gc}$, from the corresponding quantization tables. In case of vector quantization of the gains, the received index gives both the quantified adaptive gains, $\hat{g}_p$, and the quantified fixed codebook gain correction factor, $\hat{\gamma}_{gc}$.

4. Smoothing of the fixed codebook gain: An adaptive smoothing of the fixed codebook gain is performed to avoid unnatural fluctuations in the energy contour. The smoothing is based on a measure of the stationary of the short-term spectrum in the q domain.

5. Anti-sparseness processing: An adaptive anti-sparseness post-processing procedure is applied to the fixed codebook vector $c(n)$ in order to reduce perceptual artifacts arising from the sparseness of the algebraic fixed codebook vectors with only a few non-zero samples per subframe. The anti-sparseness processing consists of circular convolution of the fixed codebook vector with an impulse response. The selection of the impulse response is performed adaptively from the adaptive and fixed codebook gains [3].

6. Computing the reconstructed speech: Before the speech synthesis, a post-processing of excitation elements is performed. This means that the total excitation is modified by emphasizing the contribution of the adaptive codebook vector. Adaptive gain control (AGC) is used to compensate for the gain difference between the non-emphasized excitation $u(n)$ and emphasized excitation $\hat{u}(n)$.

7. Additional instability protection: An additional instability protection is implemented in the speech decoder which is monitoring overflows in the synthesis filter. If an overflow has occurred in the synthesis part, the whole adaptive codebook memory, $v(n), \quad n = -(143+11),...,39$ is scaled down by a factor of 4, and the synthesis filtering is repeated using this down-scaled memory.

2.3.2 Post-processing

2.3.2.1 Adaptive post-filtering (all modes)

As the encoding rate goes down, the SNR drops and the noise floor of this white coding noise is elevated to such an extent that it is very difficult to keep it below the threshold of audibility. In speech perception, the formants of speech are perceptually much more important then spectral valley regions. A good strategy is to sacrifice valley regions and preserve the formants. An important feature of the frequency response of the adaptive post-filter is that the spectral envelope peaks corresponding to the formants have roughly the same height. This feature ensures that the relative intensity of the formants will remain roughly unchanged after post-filtering [12].

The adaptive post-filter is the cascade of two filters: a formant post-filter, and a tilt compensation filter. The post-filter is updated every subframe of 5ms.
The formant post-filter is given by:

$$H_f(z) = \frac{\hat{A}(z/\gamma_n)}{\hat{A}(z/\gamma_d)} \tag{2.34}$$

where $\hat{A}(z)$ is the received quantified (and interpolated) LP inverse filter (LP analysis is not performed at the decoder), and the factors γ_n and γ_d control the amount of the formant post-filtering.
To further reduce the low-pass effect, we added a first-order filter with a transfer function $H_t(z)$ to compensate for the tilt in the formant post-filter $H_f(z)$ and is given by:

$$H_t(z) = 1 - \mu z^{-1} \tag{2.35}$$

where $\mu = \gamma_t k_1^{'}$ is a tilt factor, with $k_1^{'}$ being the first reflection coefficient calculated on the truncated ($L_h = 22$) impulse response, $h_f(n)$, of the filter $\hat{A}(z/\gamma_n)/\hat{A}(z/\gamma_d)$. $k_1^{'}$ is given by:

$$k_1^{'} = \frac{r_h(1)}{r_h(0)}; \quad r_h(i) = \sum_{j=0}^{L_h - i - 1} h_f(j) h_f(j+i) \tag{2.36}$$

Adaptive gain control (AGC) is used to compensate for the gain difference between the synthesized speech signal $\hat{s}(n)$ and the post-filtered signal $\hat{s}_f(n)$. The gain scaling factor γ_{sc} for the present subframe is computed by:

$$\gamma_{sc} = \sqrt{\frac{\sum_{n=0}^{39} \hat{s}^2(n)}{\sum_{n=0}^{39} \hat{s}_f^2(n)}} \tag{2.37}$$

The gain-scaled post-filtered signal $\hat{s}^{'}(n)$ is given by:

$$\hat{s}^{'}(n) = \beta_{sc}(n) \hat{s}_f(n) \tag{2.38}$$

where $\beta_{sc}(n)$ is updated in sample-by-sample basis and given by:

$$\beta_{sc}(n) = \alpha \beta_{sc}(n-1) + (1-\alpha)\gamma_{sc} \tag{2.39}$$

where α is AGC factor.

2.3.2.2 High-pass filtering and up-scaling

The high-pass filter serves as a precaution against undesired low frequency components. A filter cut-off frequency of 60 Hz is used. Up-scaling consists of multiplying the post-filtered speech by a factor of 2 to compensate for the down-scaling by 2 which is applied to the input signal.

2.4 Bit Allocation

The bit allocation of the AMR codec modes is shown in Table 2.1. In each 20ms speech frame, 95, 103, 118, 134, 148, 159, 204 or 244 bits are produced, corresponding to a bit-rate of 4.75, 5.15, 5.90, 6.70, 7.40, 7.95, 10.2 or 12.2 kbit/s. Note that the most significant bits (MSB) are always sent first [3].

Mode	Parameter	1st subframe	2nd subframe	3rd subframe	4th subframe	total per frame
12.2 kbit/s (GSM EFR)	2 LSP sets					38
	Pitch delay	9	6	9	6	30
	Pitch gain	4	4	4	4	16
	Algebraic code	35	35	35	35	140
	Codebook gain	5	5	5	5	20
	Total					244
10.2 kbit/s	LSP set					26
	Pitch delay	8	5	8	5	26
	Algebraic code	31	31	31	31	124
	Gains	7	7	7	7	28
	Total					204
7.95 kbit/s	LSP sets					27
	Pitch delay	8	6	8	6	28
	Pitch gain	4	4	4	4	16
	Algebraic code	17	17	17	17	68
	Codebook gain	5	5	5	5	20
	Total					159
7.40 kbit/s (TDMA EFR)	LSP set					26
	Pitch delay	8	5	8	5	26
	Algebraic code	17	17	17	17	68
	Gains	7	7	7	7	28
	Total					148
6.70 kbit/s (PDC EFR)	LSP set					26
	Pitch delay	8	4	8	4	24
	Algebraic code	14	14	14	14	56
	Gains	7	7	7	7	28
	Total					134
5.90 kbit/s	LSP set					26
	Pitch delay	8	4	8	4	24
	Algebraic code	11	11	11	11	44
	Gains	6	6	6	6	24
	Total					118
5.15 kbit/s	LSP set					23
	Pitch delay	8	4	4	4	20
	Algebraic code	9	9	9	9	36
	Gains	6	6	6	6	24
	Total					103
4.75 kbit/s	LSP set					23
	Pitch delay	8	4	4	4	20
	Algebraic code	9	9	9	9	36
	Gains	8		8		16
	Total					95

Table 2.1: Bit allocation of the AMR coding algorithm for 20ms frame

Chapter 3

Overview of IEEE 802.16a FEC Scheme

3.1 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 to 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 a set of particular physical layer specifications applicable to systems operating between 2 and 66 GHz. It supports point-to-multipoint and optional mesh topologies [14].

This standard is a 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. 3.1 (The numbers in the figure refer to IEEE standard designations). The family of standards deals with the Physical and the 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 the associated physical media, each appropriate for particular applications or system objectives. Other types are under investigation [14].

This thesis focuses on the Reed-Solomon decoder acceleration and the DSP implementation issues of the Reed-Solomon and Viterbi decoder in the IEEE 802.16a Forward Error Correction (FEC) Decoding scheme. Therefore, we will concentrate on introducing the FEC specifications defined in IEEE 802.16a physical layer part in the next section. In the last part of this chapter, we will show the block diagrams of the program conventionally implemented and also described briefly some modification and our contribution to improve the implementation structure by reducing the computational complexity. The detail of our improvement will be described in the latter chapter.

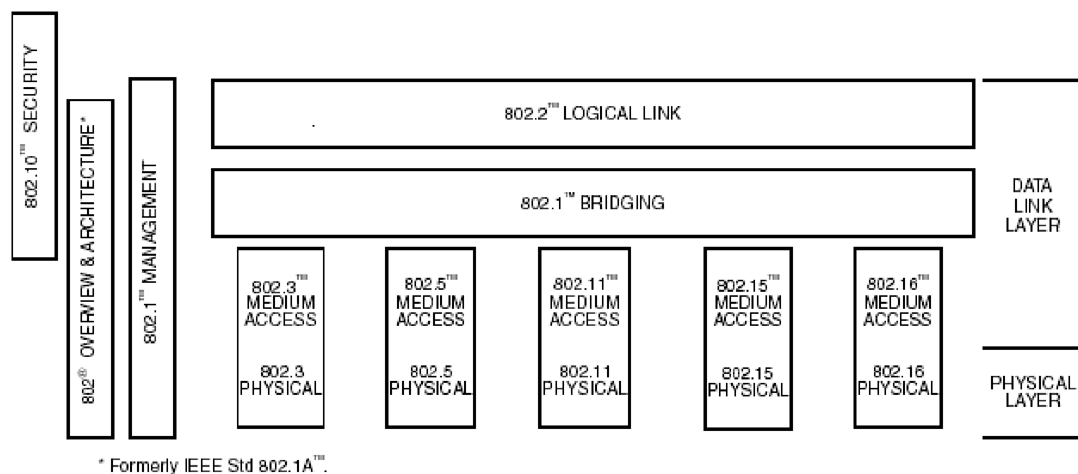


Figure 3.1: IEEE local and metropolitan area networks standards family.

3.2 IEEE 802.16a FEC Specifications

The overall physical layer structure of the channel coding scheme is shown in Fig. 3.2, where the Reed-Solomon Code and the Convolutional Code are major parts of the FEC scheme, and the randomizer and the interleaver are additional modules for further improving the error performance of the FEC scheme. The detailed specifications of each part are introduced in the following subsections, excluding the modulator, which is not implemented in our research subproject.

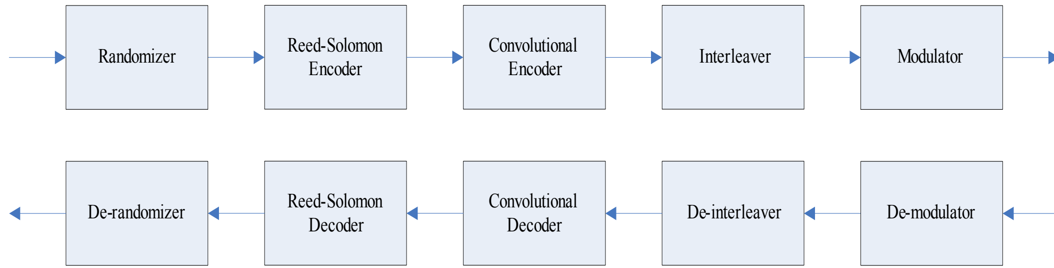


Figure 3.2: Channel coding structure at the transmitter side (top) and the receiver side (bottom).

3.2.1 Randomizer

Data randomization is performed on data transmitted on the downlink (DL) and uplink (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 match exactly the amount of data allocated, symbol “0xFF” (“1” only) should be padded to the transmission block until the allocated data are filled.

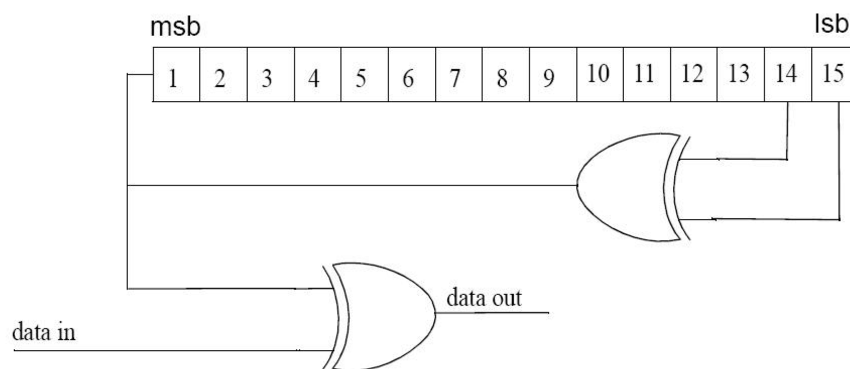


Figure 3.3: PRBS for Data Randomization.

The randomizer is a Pseudo Random Binary Sequence (PRBS) generator depicted in Fig. 3.3. As shown in the figure, source bit randomization is performed by the

modulo-2 adder and the Linear-Feedback Shift Register (LFSR) with characteristic polynomial $1+X^{14}+X^{15}$. Each data byte to be transmitted shall enter sequentially (msb first) into the randomizer to make the “0” and “1” bits well-distributed in the output data streams 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 (lsb).

In the uplink, the randomizer is initialized with the vector created as shown in Fig. 3.4.

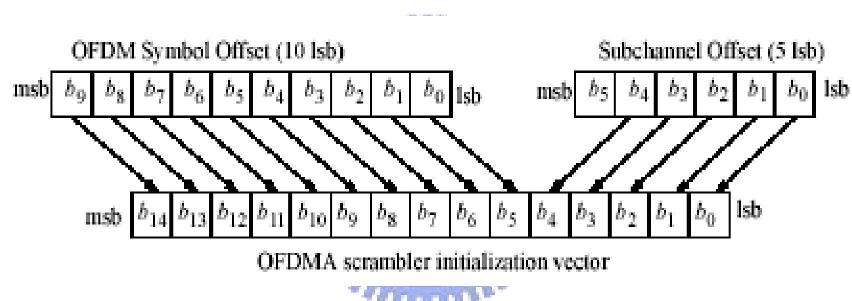


Figure 3.4: Creation of OFDMA Randomizer Initialization Vector.

3.2.2 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. 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. 3.5, 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 of which the 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.

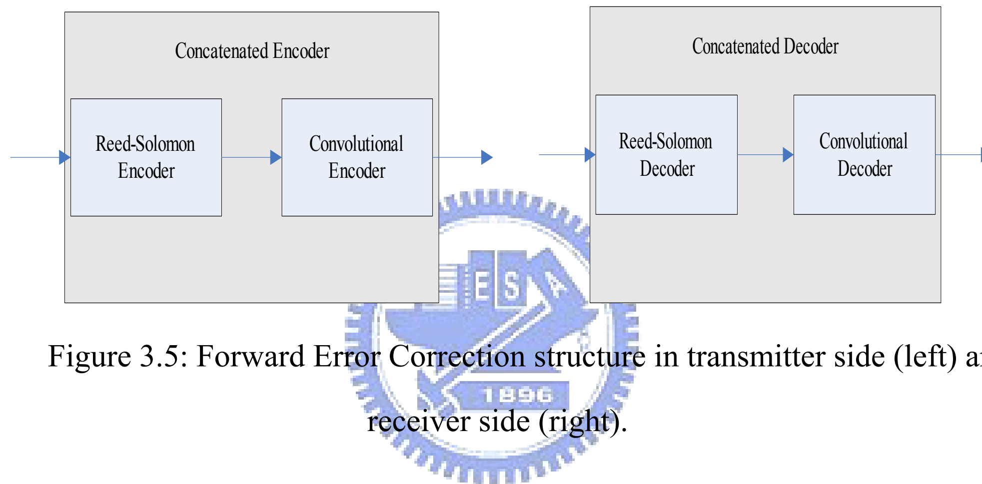


Figure 3.5: Forward Error Correction structure in transmitter side (left) and receiver side (right).

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 3.1(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-correction 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 3.1: Mandatory Channel Coding per Modulation.

3.2.2.1 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.

3.2.2.2 Convolutional Code Specification

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

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

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

The generator is depicted in Fig. 3.6.

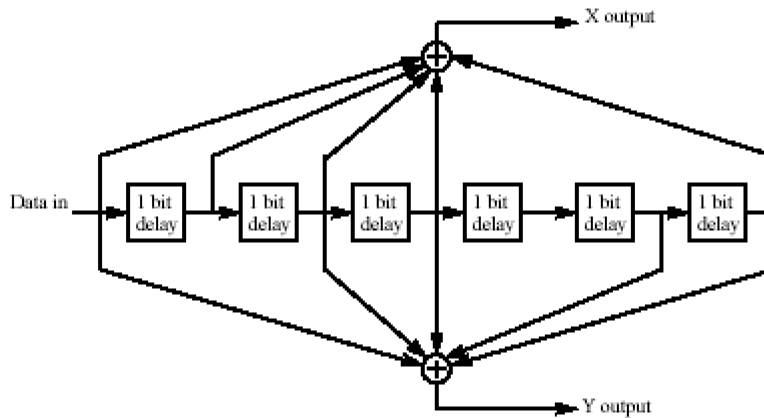


Figure 3.6: Convolutional Encoder of Rate 1/2.

Puncturing patterns and serialization order which is used to generate variable code rates are defined in Table 3.2. In the table, a "1" denotes a transmitted bit and a "0" denotes a removed bit, whereas ***X*** and ***Y*** correspond to Fig. 3.6.

	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 3.2: 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.

3.2.3 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 3.3) 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 3.3: 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.

3.3 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.

3.3.1 Reed-Solomon Code

3.3.1.1 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 3.2 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. $\mathbf{I}_{238} \sim \mathbf{I}_0, \mathbf{R}_{15} \sim \mathbf{R}_0$, 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:

$$\begin{aligned}\mathbf{I}(x) &= \mathbf{I}_{238}x^{238} + \mathbf{I}_{237}x^{237} + \ldots\ldots + \mathbf{I}_{36}x^{36} + \mathbf{I}_{35}x^{35} + \ldots\ldots + \mathbf{I}_1 x + \mathbf{I}_0 \\ &= (\mathbf{I}_{238}, \mathbf{I}_{237}, \ldots, \mathbf{I}_{36}, \mathbf{I}_{35}, \ldots, \mathbf{I}_1, \mathbf{I}_0)\end{aligned}$$

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

$$\begin{aligned}\mathbf{C}(x) &= \mathbf{I}(x) \cdot x^{16} + \mathbf{R}(x) \\ &= (\mathbf{I}_{238}, \mathbf{I}_{237}, \ldots, \mathbf{I}_{36}, \mathbf{I}_{35}, \ldots, \mathbf{I}_1, \mathbf{I}_0, \mathbf{R}_{15}, \mathbf{R}_{14}, \ldots, \mathbf{R}_3, \mathbf{R}_2, \mathbf{R}_1, \mathbf{R}_0)\end{aligned}$$

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

$$\begin{aligned}\mathbf{R}(x) &= \mathbf{I}(x) \cdot x^{16} \bmod \boldsymbol{g}(x) \\ &= (\mathbf{R}_{15}, \mathbf{R}_{14}, \ldots, \mathbf{R}_3, \mathbf{R}_2, \mathbf{R}_1, \mathbf{R}_0)\end{aligned}$$

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 3.2 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. 3.7, 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. 3.8, we employ the Linear Feedback Shift Register (LFSR) structure to implement the RS encoder block diagram as shown in Fig. 3.9 [15].

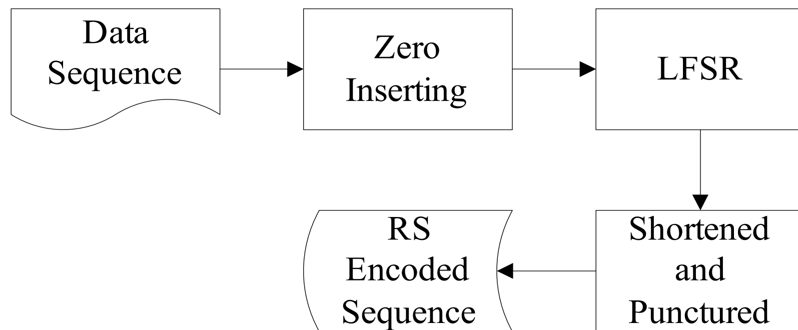


Figure 3.7: Block Diagram of the RS Encoder Program.

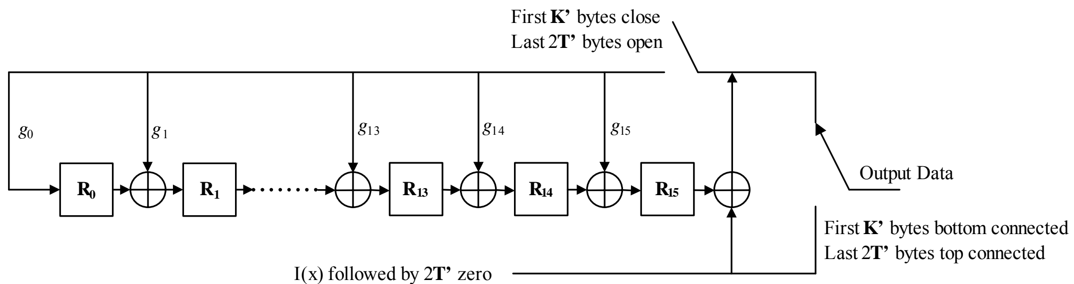


Figure 3.8: The Linear Feedback Shift Register Structure of RS Encoder.

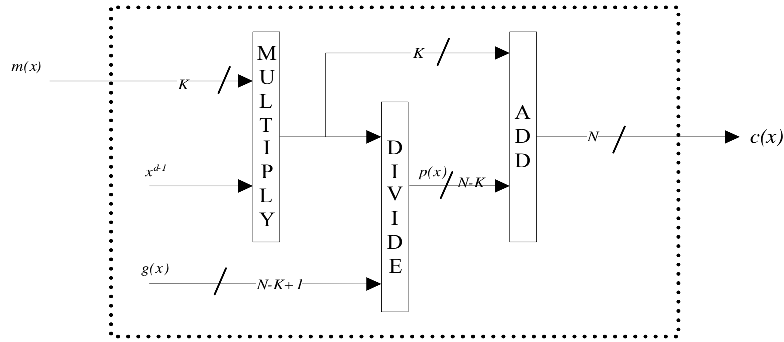
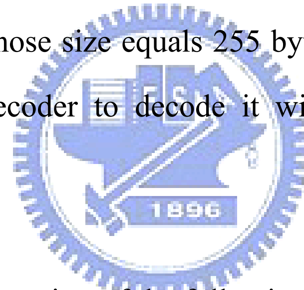


Figure 3.9: Block Diagram of a Conventional RS Encoder.

3.3.1.2 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

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

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

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

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).

Initiatively, we choose the BM algorithm to compute (1), and further simplify it by eliminating the pre-computation of the Forney syndrome and the post-computation of the errata locator polynomial in reference to the inverse-free Berlekamp-Massey algorithm. The simplified one just simply initializes 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.

As the above described, the BM algorithm is used to find the coefficients of the error/erasure locator polynomial while the chein search is used to solve its roots. It

requires multiplication of each coefficient by all the elements in GF(2^8) and however, the multiplication is much more complicated than the addition and requires a lot of computational time. It makes the chein search one of the bottlenecks of the RS decoder. To improve it, we choose an algorithm proposed as a novel algorithm for finding the roots of a special class of polynomials together with chein search to speed up the RS decoder by reducing the amount of multiplication. Moreover, we employ the Forney algorithm to solve (2).

The block diagram of the RS decoder described above is shown in Fig. 3.10, where the syndrome computation is done by employing the circuit shown in Fig. 3.11 then fed to the BM algorithm, the chein search is performed after BM algorithm, and the forney algorithm is for the purpose of computing the magnitude of the error/erasure.

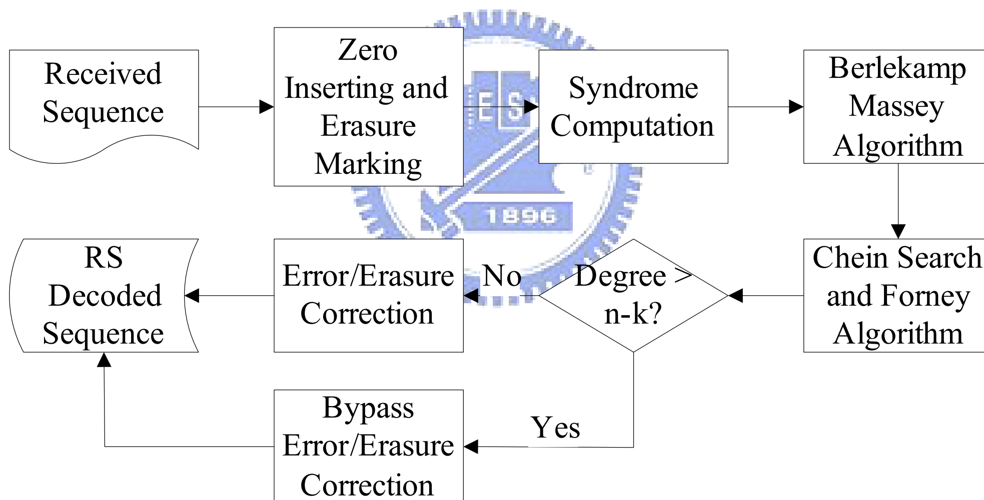


Figure 3.10: Block Diagram of the RS Decoder Program.

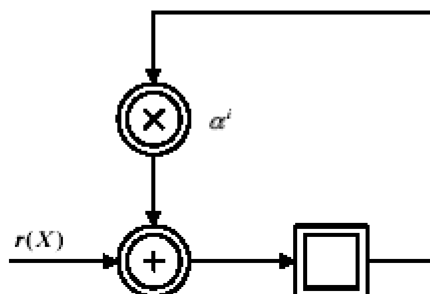
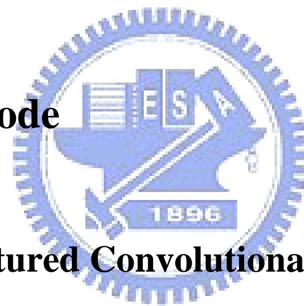


Figure 3.11: Syndrome Computation Circuit.

Except for the above algorithm, the alternative RS decoding algorithm is the remainder decoding algorithm, which has been introduced in the early 1980's. It is the other major bottleneck for the RS decoding process to calculate syndromes and sometimes we also need to have all syndromes independent of the number of errors in the received codeword. Therefore, the remainder decoding algorithm becomes popular because the prior computation of the syndromes is not required. This algorithm is achieved through solving the constrained polynomials, the Welch-Berlekamp (WB) equations, derived from the remainder of the received codeword divided by the generator polynomial. In our case, we also use the remainder decoding algorithm with WB equations to decode RS codes instead of BM algorithm together with syndrome computation for comparison and for investigating this current popular decoding algorithm.



3.3.2 Convolutional Code

3.3.2.1 Encoding of Punctured Convolutional Code

The convolutional code encoding structure is shown in Fig. 3.6. 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 3.2, 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. 3.6. 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. 3.12. 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.

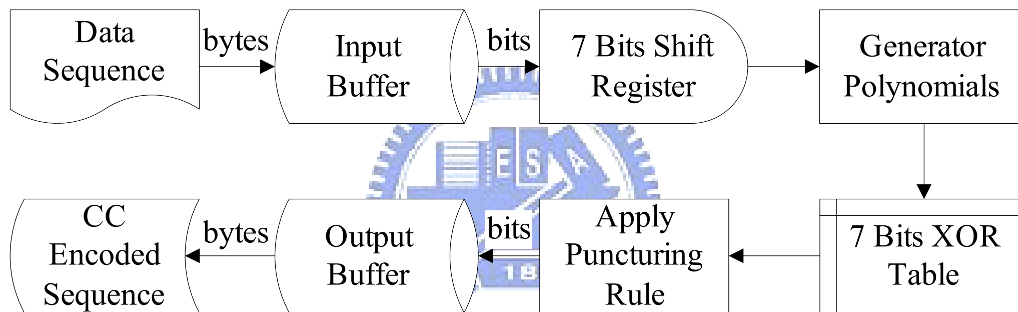


Figure 3.12: Block Diagram of the Convolutional Encoder Program.

3.3.2.2 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. 3.13, 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.

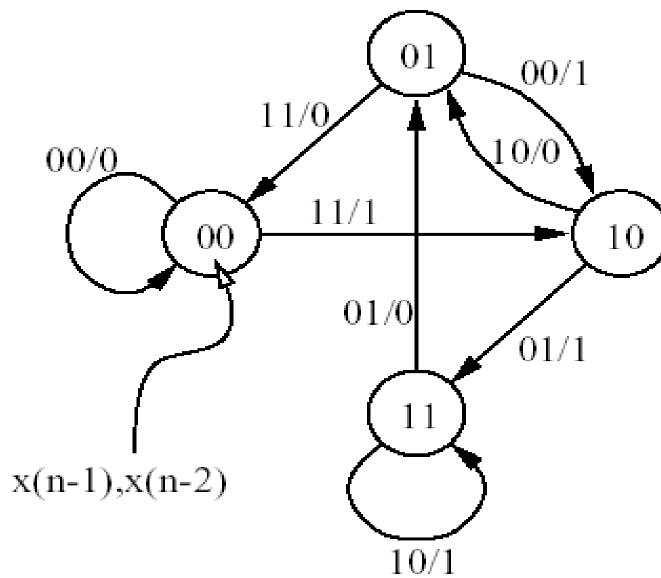


Figure 3.13: State Transition Diagram Example.

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. 3.14 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.

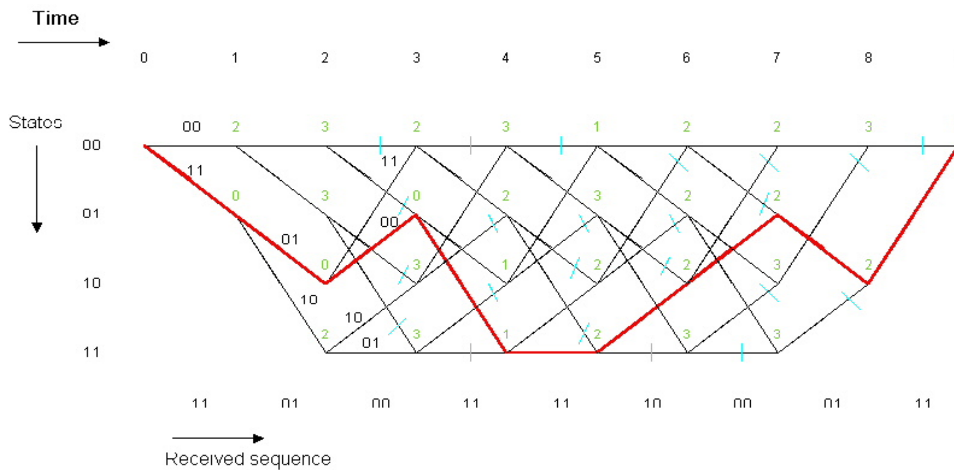


Figure 3.14: Trellis Diagram Example for a Viterbi Decoder.

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. 3.15 [16]. 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.

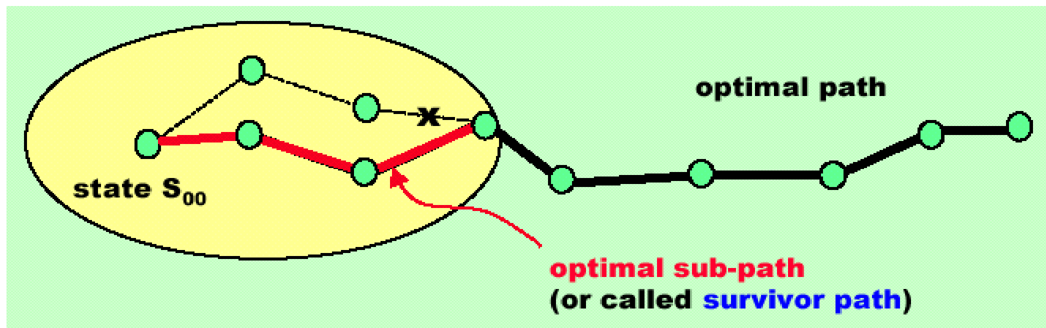


Figure 3.15: Survivor path of the Trellis Diagram.

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. 3.16, and the structure of the Viterbi algorithm is shown in Fig. 3.17. The extend received sequence block shown in Fig. 3.16 is included for decoding the puncturing and tail-biting convolutional code and will be discussed later in this subsection.

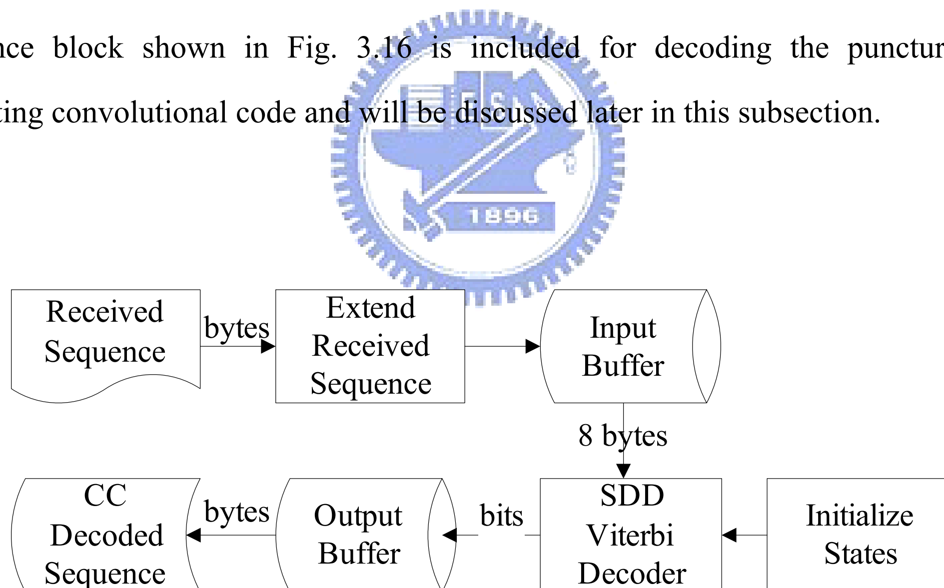


Figure 3.16: Block Diagram of the Viterbi Decoder Program.

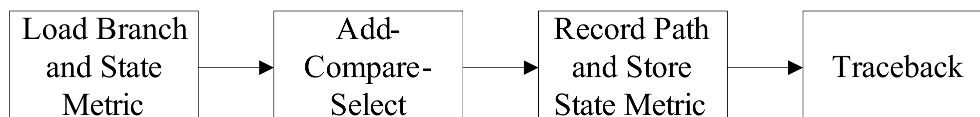
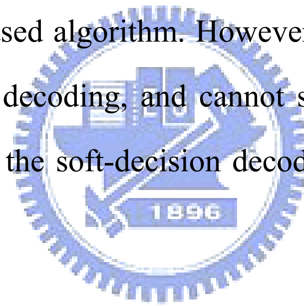


Figure 3.17: Structure of the Viterbi Algorithm.

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 [17]. Hence, the soft-decision decoding is adopted to implement our Viterbi decoder.



3.3.2.3 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 [18]. 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 [18], 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. 3.18, 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.

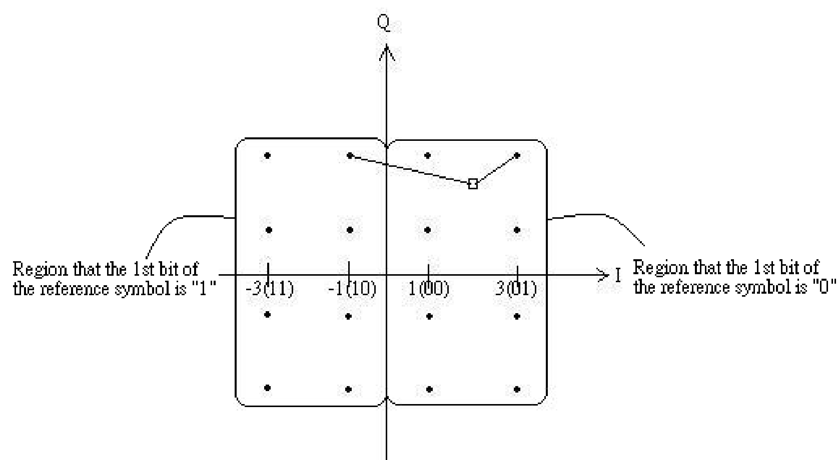


Figure 3.18: Partition of the 16-QAM Constellation.

3.3.2.4 Viterbi Decoding of Tail-Biting Convolutional Code

According to [17] and [19], the practical suboptimal tail-biting Viterbi decoder is shown in Fig. 3.19, 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 [17].

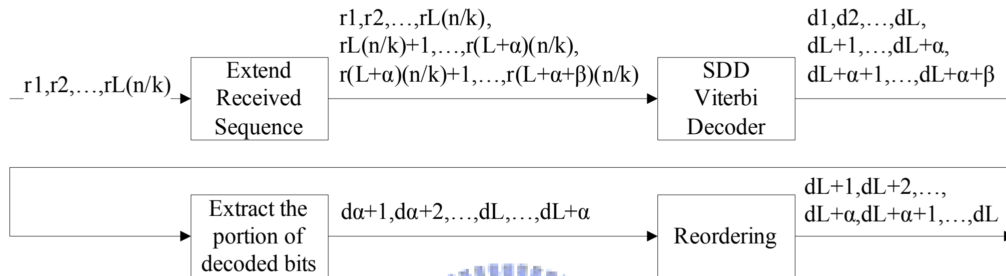


Figure 3.19: Block Diagram of the Suboptimal Tail-Biting Viterbi Decoder.

3.3.2.5 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. 3.20 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:

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

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

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.

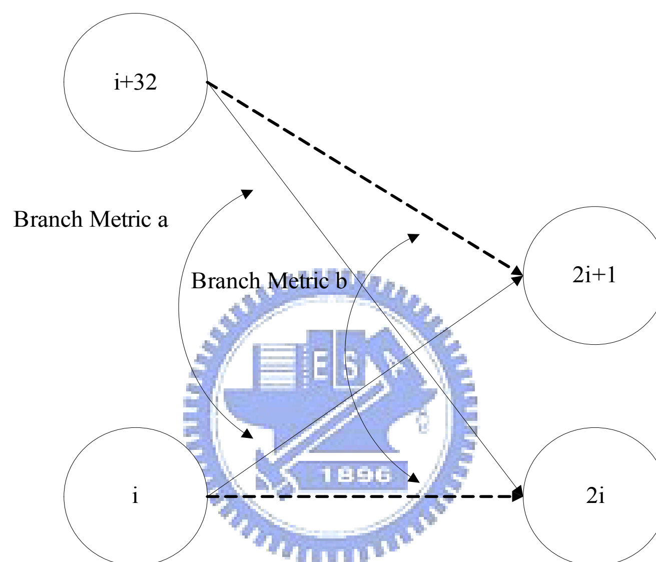


Figure 3.20: Butterfly Structure Showing Branch Cost Symmetry.

Chapter 4

DSP Implementation Environment

Our project is a subproject of an integrated group project. The TI DSP is chosen to be the platform of the whole system. The DSP baseboard we use is Innovative Integration's (II's) product in year 2003 called Quixote, which houses Texas Instruments' TMS320C6416 DSP chip. In this chapter, the specification of the DSP chip and the DSP baseboard and the data transmission process from the host PC to the target DSP are described. Moreover, some important techniques and features which benefit our acceleration work are also included.

4.1 The DSP Chip

The DSP chip we adopt is one in the TMS320C64x series. According to [21], TMS320C64x series is also 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. 4.1. 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 on only certain devices.

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

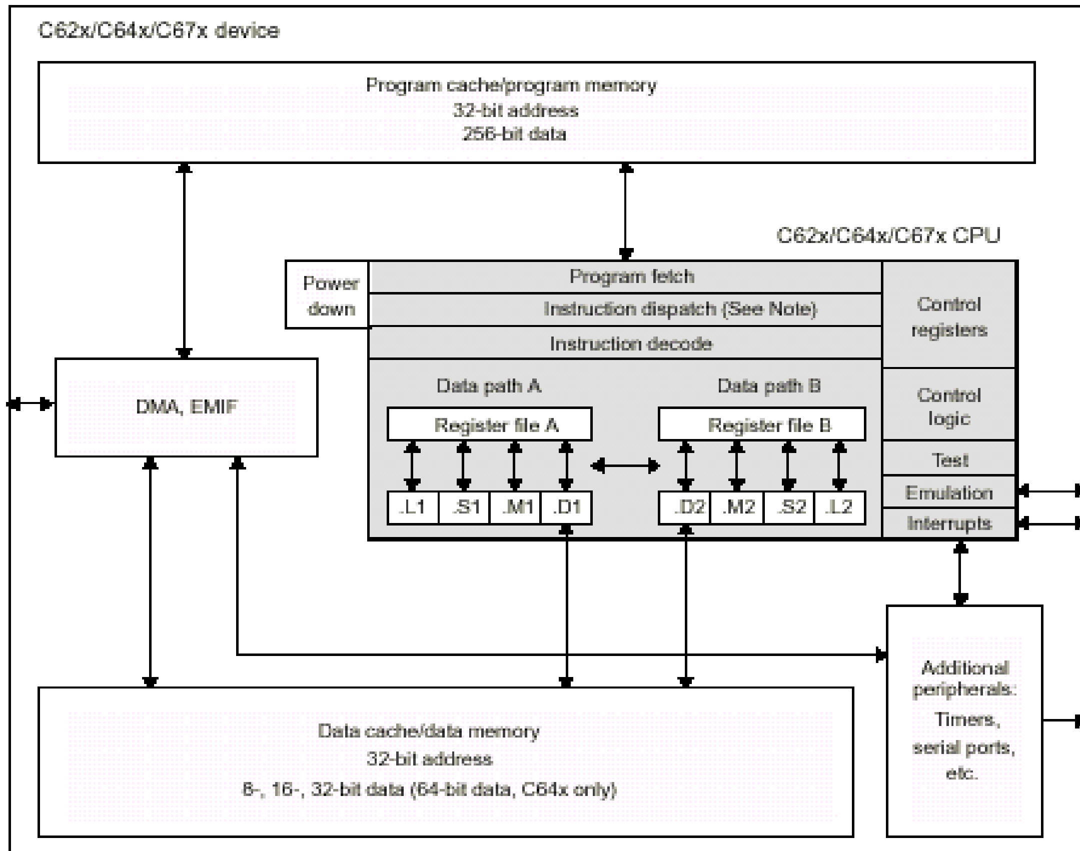


Figure 4.1: The Block Diagram of TMS320C6x DSP Chip.

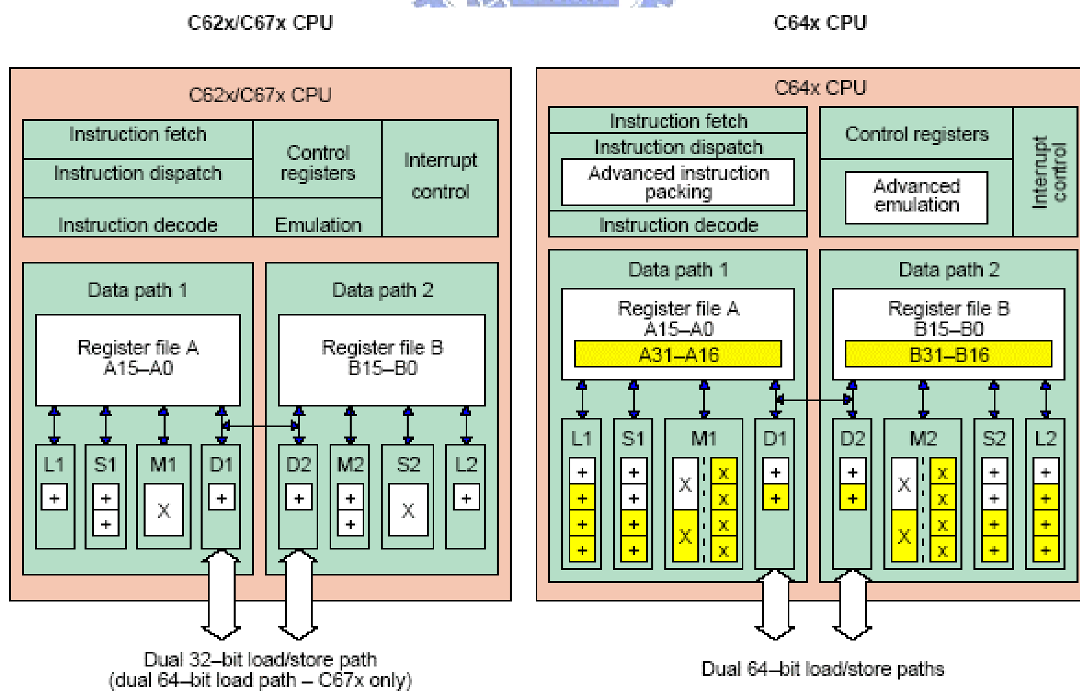


Figure 4.2: The TMS320C64x DSP Chip Architecture and Comparison with Ancient TMS320C62x/C67x Chip.

4.1.1 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 .4.2 [22]. 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. 4.2. 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.

4.1.2 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.

4.1.3 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.

4.2 The DSP Baseboard

The Quixote DSP Baseboard card is shown in Fig. 4.3 and the architecture is shown in Fig. 4.4 [25]. 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.

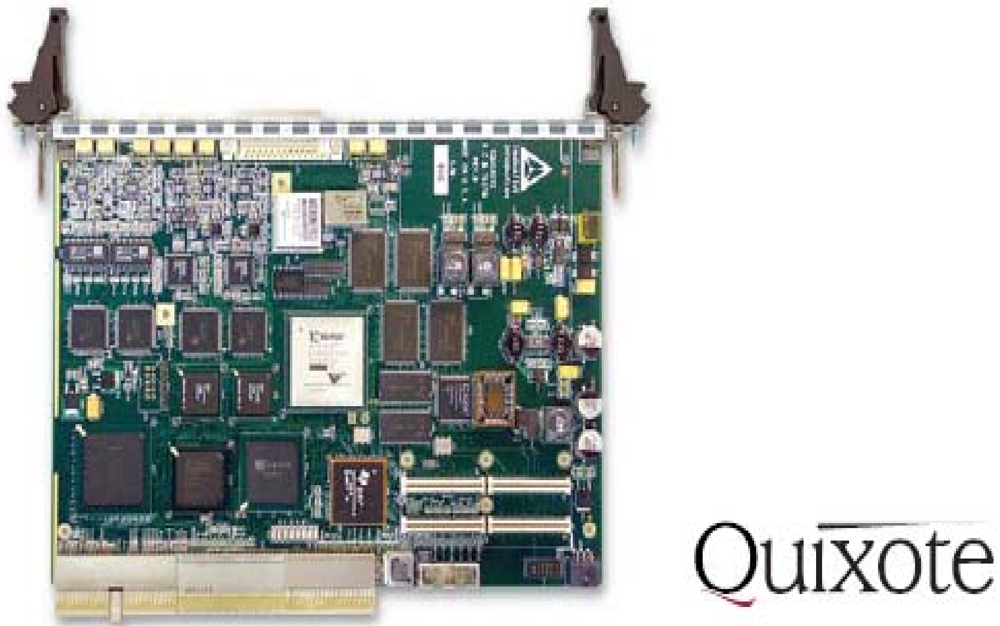


Figure 4.3: Innovative Integration's Quixote DSP Baseboard Card

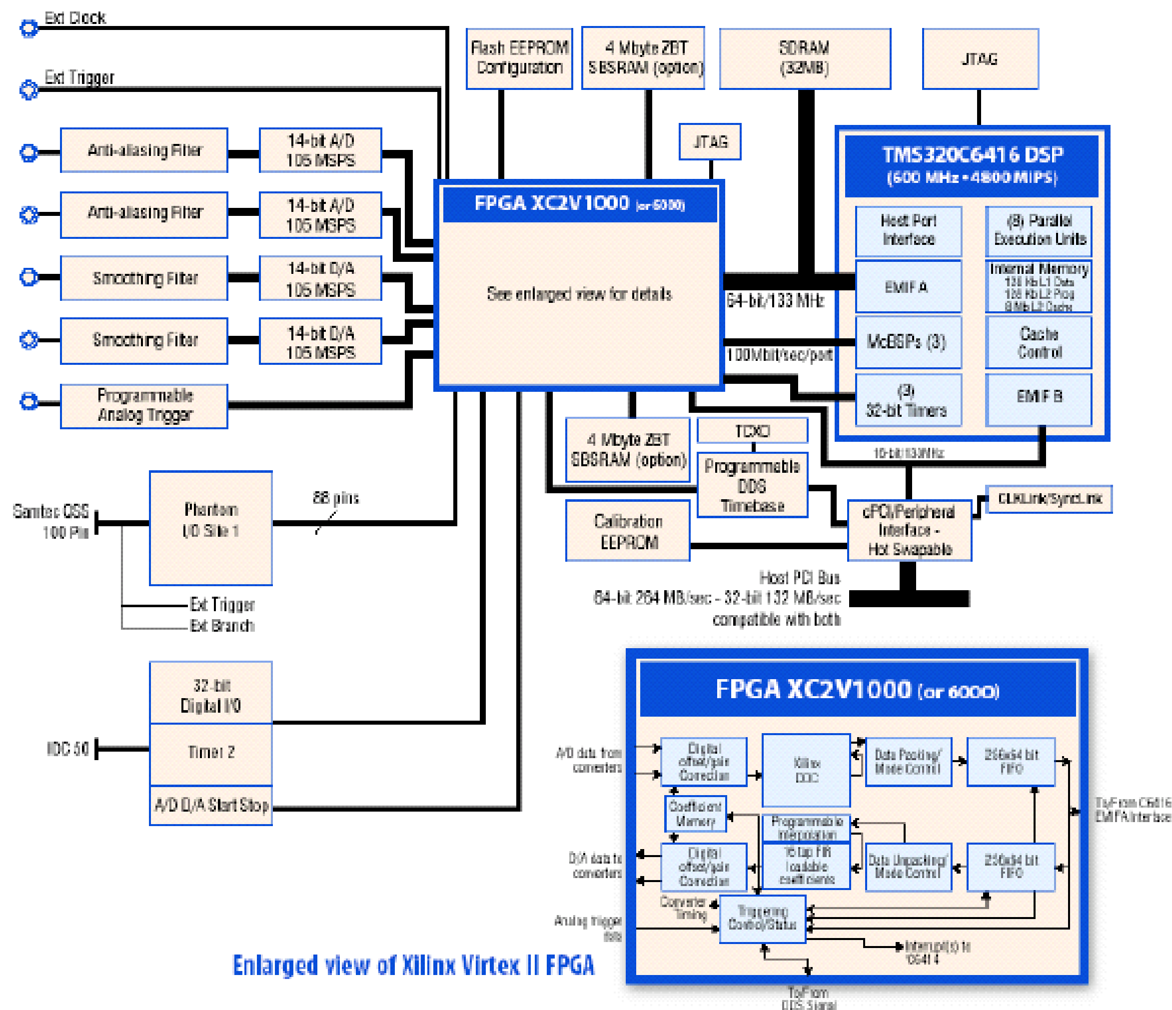


Figure 4.4: The Architecture of Quixote Baseboard

4.3 Data Transmission 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.

For user's different requirements, it provides different levels of support to efficiently accomplish. The simplest 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 the sending of text strings to the user.

The next level of support is given 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 does. 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 our application of AMR speech coding and RS coding scheme, the data sequence is first divided into RS blocks (or speech frames for AMR) then performed encoding and decoding procedure. Hence the continuous streaming may not be suitable for our requirements. 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. 4.5 [25].

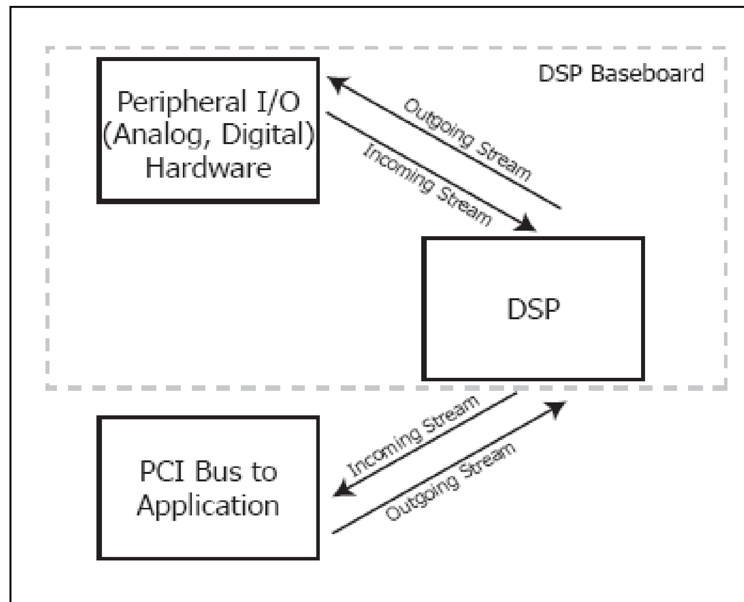


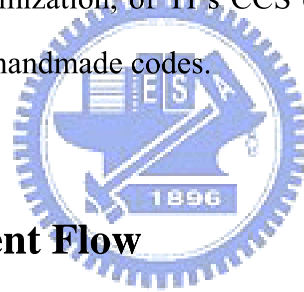
Figure 4.5: Block Diagram of DSP Streaming Mode.

DSP Streaming is initiated and started on the Host, using the Caliente component. On the target, the DSP interface uses 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. Typical desktop machines routinely support transfers at 80~100 MBytes/sec. Besides, maximum throughput supported by the driver is somewhat dependent on the size of the buffers used in the driver pool.

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

4.4 Features of TI TMSC6000 Family DSP for Optimization

In this subsection, first the code development flow is presented to show how to program a DSP efficiently and systematically by the handmade efforts only. Secondly, the TI C6000 family pipeline structure is introduced for the ease to understand how the processor arrange the pipeline stages and what instruction is more time consuming and shall be avoided using if possible. Thirdly, an important techniques used by TI's CCS compiler to improve the program speed performance, which is so-called "software pipelining", is introduced and a simple example is given here to explain how we can improve the program efficiency by software pipelining technique. At last, the important option, the compiler level optimization, of TI's CCS compiler is also involved for the advanced improvement of our handmade codes.



4.4.1 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 aid in optimization rather than forcing the programmer to code by hand in assembly. These advantages allow the compiler to do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation. Fig. 4.6 illustrate the three phases in the code development flow [23]. 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 hardly achieved or the

unbalanced resource allocation is hardly solved by the compiler or adjusting only the C code.

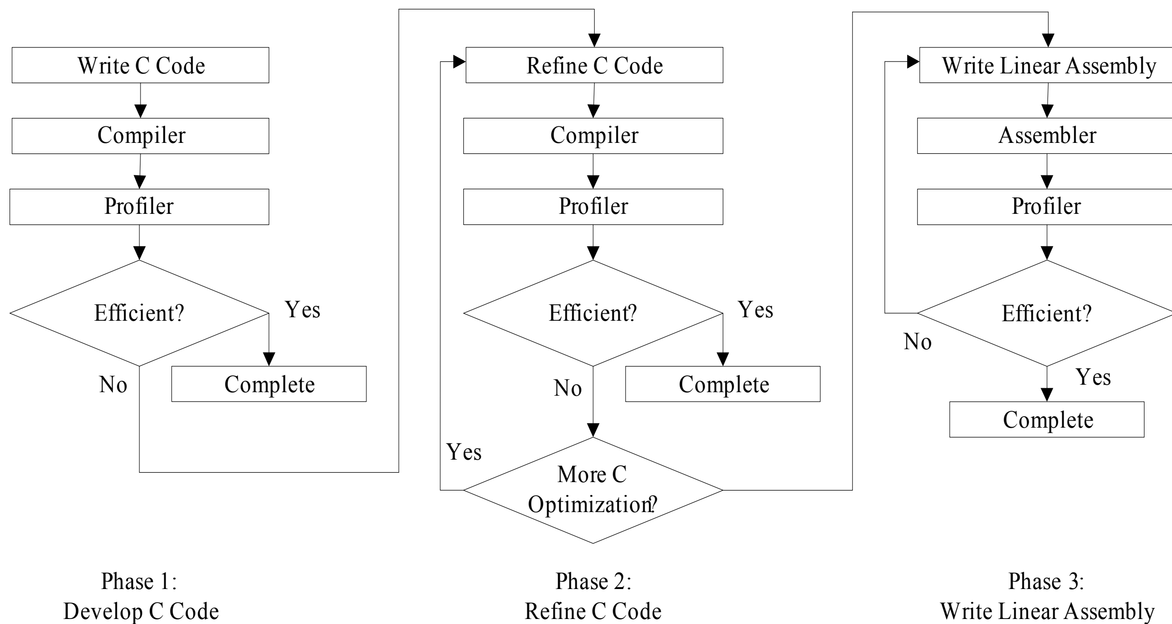


Figure 4.6: Code Development Flow.

4.4.2 Pipeline Structure of the TI TMSC6000 Family

Pipelining is an efficient way to increase the instruction throughput. There are some features with regard to the TI C6000 family's pipeline structure that can provide the advantages of optimum performance, low cost, and simple programming. The following are several useful features [21]:

- *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 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 (PF), Decode stage (D), and Execution stage (E). At the **F** stage, the CPU first generates an address, fetch the opcode of the specified instruction from memory, and then pass it to the program decoder. At the **D** stage, the program decoder efficiently routes the opcode to the specific functional unit which determined by the type of instruction (LDW, ADD, SHR, MPY, etc). Once the instruction reaches **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 finish 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.

4.4.3 Software Pipelining

Software pipelining is a technique for interleaving instructions from different iterations to eliminate the dependency within one iteration and exploit instruction level parallelism (ILP) in loops, so the delay slots can be filled and the functional units can be used more efficiently. TI's CCS compiler is also capable of this. Overall it makes the loop become a highly optimized loop code and hence accelerate the program operating speed significantly.

For the ease to understand how software pipelining actually works, here we give an example to illustrate it [26]. A simple *for* loop and its code after applying software pipelining are shown in Fig 4.7(a) and 4.7(b). The loop schedule length is reduced from four control steps to one control step for software pipelined loop. However the code size of software pipelined loop is three times larger than the original code size in this example. Fig. 4.8(a) and 4.8(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 4.7: (a) The Original Loop. (b) The Loop After Applying Software Pipelining.

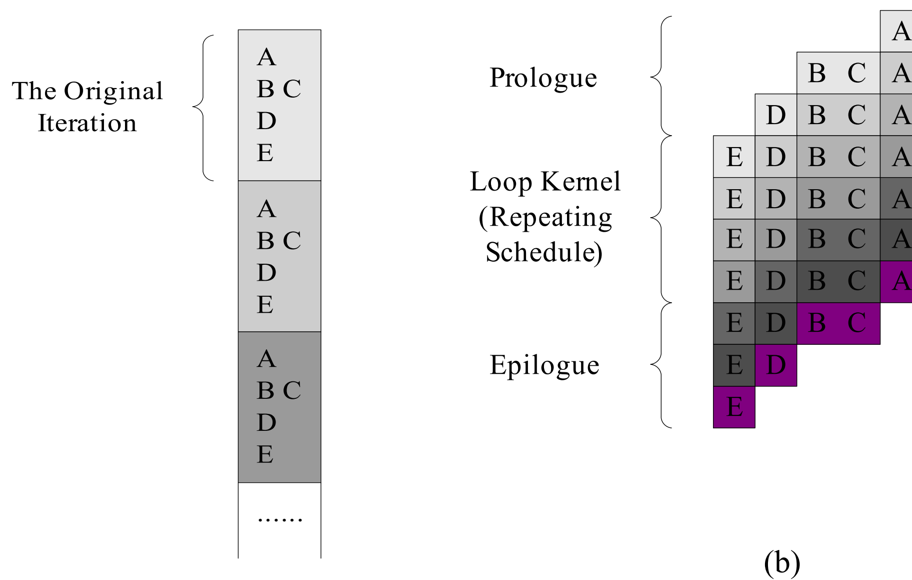


Figure 4.8: (a) Execution Record of the Original Loop. (b) Execution Record of the Software Pipelined Loop.

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 original loop, while there are all five instructions executed in parallel in 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 area 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 on our loops.

Concerning implementation on TI C6000 DSP family, C code loop performance is greatly influenced by how well the CCS compiler can software pipeline our loop. The compiler provides some feedback information for programmers to fine-tune the loop structure. By understanding the feedback information, we can quickly tune our C code to obtain the highest possible performance. The feedback is geared for explaining exactly what all the issues with pipelining the loop were and what the results obtained were. The compiler goes through three basic stages when compiling a loop, these stages are [23]：

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 factor 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 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 are available and require a longer compilation time.
- It cannot call another function from 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 on our loop 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 and 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 attempt to software pipeline our loop based on the knowledge it obtained from the previous two stages. The first thing the compiler attempts 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 schedule, and the compiler will keep proceeding to ii = (previous failed ii + 1) till it find a valid schedule and then the software pipeline is done.

4.4.4 Program-Level Optimization

Four optimization levels are provided by the CCS compiler. Program level is the highest one of optimization available. With program-level optimization, all our source files are compiled into one intermediate file giving the compiler complete program view during compilation. It performs various loop optimizations, such as software pipelining, unrolling, and SIMD, etc. and also other code size reducing like: eliminating unused assignments, eliminating local and global common sub-expressions,

and removing functions that are never called. It creates significant advantage for determining whether two pointers access the same memory location to eliminate the memory dependency in loops and lead to better schedules.

Chapter 5

Implementation and Acceleration of AMR Speech Coding on TI DSP Platform

As described in the previous chapter, we adopt the Texas Instruments (TI) digital signal processor (DSP) for implementing our AMR (Adaptive Multi-Rate) codec and RS (Reed-Solomon) decoder in the IEEE 802.16a wireless communication standard. In this chapter, we focus first on one of our major topic of this thesis – the implementation and acceleration of the AMR codec on the newly released II's Quixote DSP baseboard. At first, we introduce some special features of TI C6000 family DSP that is helpful when doing compiler level optimization. Secondly, we proposed some simple and yet practically useful techniques to speed up the computational performance of the AMR codec for TI C64 family DSP. Then, we show the simulation profile, which is generated by the TI's Code Composer Studio (CCS) built-in profiler, of the AMR codec after the acceleration. Finally, we describe the entire system structure and the operation of our AMR codec implemented on the TI C64 DSP platform. Moreover, the presentation of the execution time after our acceleration is also attached.

5.1 AMR Codec Acceleration

Follow the code development flow described in the previous chapter. Before actually revising our program code, we should first generate a profile by using the CCS built-in profiler to obtain exact execution cycles. Then we identify the parts of our program that consume the most execution time based on the profile data. And hence we concentrate on the most efficient method for speeding up them to make the whole program faster. The acceleration steps of our AMR codec program before it being implemented on TI DSP platform is discussed in the following subsections.

5.1.1 AMR Code Profile

There are two methods to use the standalone simulator for profiling [23]

- The –g option provides a profile of all of the functions in our application.
- If we are interested in only one or two functions or a region of code inside a function, the clock( ) function can be used to time the region specified.

For the purpose to find which parts take the most operation time, we choose the first method to compare all the functions in our AMR program. The source code we use here is the adjusted one of the fixed-point version from the AMR speech codec series of the 3GPP specification website. And the test sequences used to profile our code is also obtained from there. The function and the usage of the source code involves are described as follow.

The general command line syntax for the encoder program is

encoder [options] amr_mode input_filename bitstream_filename

or

encoder [options] -modefile=mode_file input_filename bitstream_filename

Basically, it contains the filenames of input and output files for user to specify, and the format of the input speech file is 16-bit linear encoded PCM speech samples with the 8 kHz sampling rate and the frame length of 160 samples. The frame of the encoder output bitstreams are structured as

1 word	frame type
244 words	encoded speech parameter bitstream (one bit per word, each word contains either 0x0001 or 0x0000), unused bits written as 0x0000 for modes < MR122
1 word	mode information
4 words	unused (written as 0x0000 by encoder)

In the first case of the syntax, “amr_mode”, which represents one of the eight source rates of AMR codec, must be one of MR475, MR515, MR59, MR67, MR74, MR795, MR102, and MR122. In the second case, the text file “mode_file” must contain the mode names to be used. This mode is capable of switching its bit-rate every 20-ms speech frame.

The option recognized by the encoder command line is “-dtx”, which is used to enable DTX operation. The information that explains the DTX operation can be found in the chapter 2.

The general command line syntax for decoder program is similar to the one for encoder except for the predetermined mode:

decoder [options] bitstream_file output_file

The structure of the input bitstream and output file for decoder are the same as the output bitstream and input file described in the encoder section. The mode and frame type for decoding also refers to the information contained in the received bitstream unless the option “-rxframetype” is used to force RX frame type (instead of TX frame type in the input file). However, this option is only useful for simulations.

We first profile this original version AMR program by the CCS simulator without compiler-direct optimizations and handmade improvements. First we roughly segment the AMR encoder procedure into a few major sections and the measure their operation cycles by the CCS build-in profiler.

Procedure	Cycles	Percentage (%)
Pre-processing	148,205	1.46
Linear Prediction Analysis	1,509,708	14.85
Open-Loop Pitch Analysis	1,789,777	17.61
Impulse Response and Target Signal Computation	857,060	8.43
Adaptive Codebook Search	1,745,931	17.18
Algebraic Codebook Search	3,265,651	32.13
Quantization of the Adaptive and Fixed Codebook Gains	789,478	7.77
Memory Update	58,466	0.58

Table 5.1: Profile of AMR Encoder Provided by 3GPP

As shown on Table 5.1, the algebraic codebook search part takes the most cycles in the AMR encoder. Therefore, we further analyze this module to find which sub-module uses the most percentage of operation time. It is found that the action of searching the best codevector is the most time-consuming unit, and it takes about 54.42% of the algebraic codebook search execution cycles. Then, we analyze the code structure of this unit, and it is presented as the combination of value assignments and various basic operations, such as addition, subtraction, multiplication, and division, etc. The other functions are written in a similar style – a sequence of function calls of the mentioned basic operations. Similar cases appear in the AMR decoder, too. Moreover, the profile data for individual functions of the encoder, which is shown in Table 5.2, supports our observation.

Function Name	Count	Average Cycles	Total Cycles	Percentage (%)
L_mult	101,960	46	4,690,160	19.00
L_add	84,888	38	4,057,107	16.44
L_mac	72,338	126	9,840,534	39.88
saturate	34,749	70	2,466,864	10.00
mult	20,545	128	2,640,418	10.70
L_sub	17,739	36	638,988	2.59
L_msu	15,133	124	1,881,897	7.63
round	10,016	113	1,132,027	4.59
add	7,110	116	824,760	3.34
sub	6,854	115	794,749	3.22

Table 5.2: Profile of the Top Ten Encoder Functions Called Most (Except for the Functions Containing Value Assignment Only)

More exactly the most frequently called functions are the mathematical arithmetics including mult (multiplication), add (addition), mac (multiplication and cumulation), saturate (saturating the 32-bit input to a 16-bit value), sub (subtraction), msu (multiplication and subtraction), round (rounding the 32-bit input to the MSB 16-bit value). In Table 5.2, the letter "L" in the function names represents 32-bit outputs. Although each individual arithmetic function has only a few operation cycles, the considerable account of calling them results in the enormous cumulative time of execution. Specially, "L_mac" occupies up to 39.88% of the whole encoding time although it calls the functions of addition and multiplication only. Hence based on the above obsevations, it is noticeable for us to accelerate the codes of the basic arithmetic functions.

5.1.2 Acceleration by Using the Intrinsics

Before introducing the acceleration methods we use, let us summarize our main points in the previous section.

1. *Referring to the profiling data and the discussion in the previous section, we know that to focus on the arithmetic functions is the overriding work for accelerating the AMR codec. Those arithmetic functions are called for numbers of times by various procedures, so simply improving their codes is quite efficient way to speed up more than one procedure and even the whole AMR codec.*
2. *It matters that we accelerate the AMR codec by improving the coding style of those arithmetic functions instead of changing the algorithm of the procedure taking the most cycles. The reason is that the specification of the AMR speech codec is fixed by the standard, and even the detail operations of any procedure are defined. Not like the audio and video standards, algorithms of the speech coding standard are always fixed and not flexible for us to modify.*

The above points sufficiently support us to accelerate the AMR codec primarily by the code improvement of the arithmetic functions. First, we profile those functions before acceleration as shown in Table 5.3.

Function Name	Count	Code Size	Average Cycles	Total Cycles
saturate	34,749	152	70	2,466,864
abs_s	19	116	48	927
shl	327	320	110	35,970
shr	1,098	252	68	75,681
mult	20,545	124	128	2,640,418
L_mult	101,960	124	46	4,690,160
negate	162	76	32	5,184
L_add	84,888	148	47	4,057,107
L_negate	5	68	25	125
mult_r	240	148	136	32,826
L_shr	1,815	236	63	115,737
L_abs	180	96	37	6,696
norm_s	10	204	202	2,027
norm_l	103	180	287	29,648

Table 5.3: Profile of AMR Codec Arithmetic Functions (Not Counted are Value Assignments or Function Calling Only).

It is clear that the calling account is the most important factor that contributes the total execution cycles. Each function takes a few cycles and has a simple structure because it contains a couple of basic arithmetic operations. It seems that the most obvious way to accelerate it without increasing the code size is using intrinsic functions (or intrinsics).

The intrinsics, which are special functions provided by the C6000 compiler, map directly to inline C64x instructions and hence result in no increase of the code size [23]. They can speed up the codes quickly and efficiently and are accessed by just calling them as an ordinary function with a leading "_". The intrinsics we use to accelerating those arithmetic functions are introduced below [23].

int _spack2(int src1, int src2)	Two signed 32-bit values are saturated to 16-bit values and packed into the return value.
int _abs2(int src2)	Calculates the absolute value for each 16-bit value.
int _sshl(int src2, uint src1)	Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result.
int _sshvr(int src2, int src1)	Shifts src2 to the right of src1 bits. Saturates the result if the shifted value is greater than MAX_INT or less than MIN_INT.
int _mpy(int src1, int src2)	Multiplies the 16 LSBs of src1 by the 16 LSBs of src2 and returns the result.
int _smpy(int src1, int src2)	Multiplies src1 by src2, left-shifts the result by one, and returns the result. If the result is 0x80000000, saturates the result to 0x7FFFFFFF.
int _ssub(int src1, int src2)	Subtracts src2 from src1, saturates the result size, and returns the result.
int _sadd(int src1, int src2)	Adds src1 to src2 and saturates the result. Return the result.
int _abs(int src2)	Returns the saturated absolute value of src2.
uint _norm(int src2)	Returns the number of bits up to the first nonredundant sign bit of src2.

The function names shown in the above are the intrinsics we use, where “int” and “uint” represent the data types of integer and unsigned integer. The arithmetic operations of the AMR codec program are of many different kinds, but the amount of intrinsics C64x provided is a few. To accelerate every operation which takes excessive

execution time, we need to modify its argument before calling it and/or add additional simple operations to its output.

For example, a simple way to realize the calculation of a 16-bit absolute value is to use "_abs2" because this instruction only imports the 16-bit LSBs of its two inputs even if the input data type supported by the instruction is a 32-bit integer. Hence we can import a zero and the value which needs to be calculated its absolute value as the two inputs and truncate the output to a 16-bit LSB.

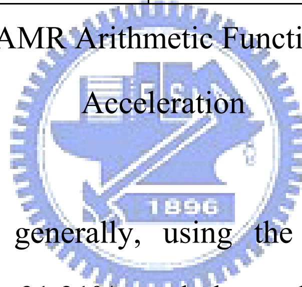
Another example is to realize the "negate" operation, which is to negate one 16-bit value with saturation when the minimum negative input appears. Here, we use the intrinsic "_ssub" to achieve this operation by subtracting the input value from zero. However, the intrinsic is designed for the 32-bit operation and performs only the 32-bit saturation instead of the 16-bit one. We shift the input to the left by 16 bits and then shift back the output to the right by 16 bits after the intrinsic "_ssub" realizes the "negate" operation. It corresponds to enlarging the 16-bit input to 32-bit to make it suitable for the 32-bit operation of "_ssub" and recover the result after the intrinsic is executed.

The "saturate" function is the operation to saturate a 32-bit value to 16 bits and has to be speeded up to reduce its total execution cycles. By a way similar to the previous cases, we have two choices of the intrinsics to perform the "saturate" operation. One is the intrinsic of "_sshl", which performs shift left and checks if the saturation happens. Hence, it is immediate that we can use "_sshl" to shift the input to the left of 16 bits and saturate the result, and then we have to shift back its output by hand. Alternatively, "_spack2" is the other choice. Both of its 32-bit inputs are packed into one 32-bit value after their 16-bit LSBs are satureated. Corresponding to the case of calculating 16-bit absolute value, a zero and the input value are imported into the intrinsic "_spack2", and its output truncated to 16 bits is the outcome of the "saturate" function.

Previous examples are several of our works in using the intrinsics to accelerate the arithmetic functions. The profile of the codes after improvement is shown in Table 5.4, where their percentage of acceleration is also listed.

Function Name	Count	Code Size	Average Cycles	Total Cycles	Improvement Percentage (%)
saturate	34,749	96	35	1216285	50.70
abs_s	19	48	24	456	50.81
shl	327	268	85	27795	22.73
shr	1,098	148	44	48312	36.16
mult	20,545	100	82	1684690	36.20
L_mult	101,960	108	43	4384280	6.52
negate	162	56	26	4212	18.75
L_add	84,888	116	45	3887331	4.18
L_negate	5	40	16	80	36.00
mult_r	240	120	89	21360	34.93
L_shr	1,815	152	43	78045	32.57
L_abs	180	40	16	2880	56.99
norm_s	10	76	34	340	83.23
norm_l	103	64	25	2575	91.31

Table 5.4: Profile of AMR Arithmetic Functions Listed in Table 5.3 after Acceleration



Referring to Table 5.4, generally, using the intrinsics can achieve a high acceleration gain even up to 91.31%, and the code size is also highly decreased because some block of codes can be replaced with one intrinsic function. However, between their codes there are still some arithmetic functions of insignificant improvement efforts because a few flags, like “overflow” or “carry”, which discourage the usage of the intrinsics. For example, the function “L_add” performs the saturation after addition, and the intrinsic with this operation is also available. However, if the constraints of saturation are matched, the output of addition is not only adjusted to the maximum or minimum of an integer, but also the flag “overflow” is set by the function. Hence, only the operation of the addition and saturation in the “L_add” can be replaced with the intrinsic. The branch instructions for the judgment of saturation still cannot be eliminated because there is a flag operation in the “L_add”. It is also the restriction of the intrinsics that the user has to make sure that the properties of the

target operations match well the intrinsics, otherwise, little improvement after implementation.

5.1.3 Compiler Level Improvement

In the last part for acceleration of the AMR codec, we try to improve the speed of our program by tuning the CCS compiler's setting. The compiler is always more conversant with the structure of DSP hardware than the programmers. Even if the handmade improvement has been finished, the codes may not be suitable for software pipelining.

In order to make the compiler work more efficiently, we try to set the "Opt. Level" option to the "File" level [23]. It enables the compiler to comprehend the information of the entire program. As described in the chapter 4, this setting can deal with certain function calls inside a loop and eliminate the coding styles that disable the software pipelining. The execution cycles after the file level optimization are presented in Table 5.5 and are also compared here to the non-optimized version of the codes with and without the intrinsics.

Encoder Version	Code Size	Cycles	Improvement Percentage (%)
Original	31,791,683	24,673,217	N/A
Modification with Intrinsics	31,790,850	22,656,174	8.18
File-Level Optimization	31,757,874	7,678,555	66.11

Table 5.5: Profile of Different Improved Versions of AMR Encoder

Decoder Version	Code Size	Cycles	Improvement Percentage (%)
Original	31,681,519	3,412,267	N/A
Modification with Intrinsics	31,680,687	3,190,223	6.51
File-Level Optimization	31,662,943	1,155,983	63.76

Table 5.6: Profile of Different Improved Versions of AMR Decoder

The lists of “Improvement Percentage” in Table 5.5 and Table 5.6 show the improvement gain between the previous and present versions. Compared to the original setting, the program is executed much more efficiently, and a large percentage of cycles about 68.88% for the encoder and 66.12% for the decoder is reduced in the final version. We also measure the processing cycles using the file-level optimization without intrinsics. The improvement percentage decreases to 58.05% for the encoder and 57.15% for the decoder because of the reason below:

The intrinsic integrates multiple lines of codes into one instruction, and this benefits the compiler to establish the software pipelines. Hence, the gain of the file-level optimization without intrinsics may be lower, and, on the other hand, the intrinsics without compiler-direct optimization obtain limited benefits because the software pipeline disables without suitable function inlining.

5.2 AMR Codec on C64x DSP Platform

5.2.1 Structure of AMR Implementation

The code development environment is Visual C++ with Armada library provided by II at the host end and Code Composer Studio (CCS) with Matador Pismo library at the DSP end. We choose the GUI interface to import the input and show the results for convenience and visualization.

The program located at the host end responses for the interface initialization, the definition of button clicks, and the message handler. The message, which is one of the data transmission mechanism supported by our DSP platform, is described previously in Chapter 4. It is used for the transmission of small amount of data at low speed such as signaling between the host and DSP. Thus, the message handler should include the actions to reply to various kinds of messages. The other transmission mechanism we use in our implementation is block-based data transfer, which is also described in Chapter 4 and it is responsible for the data transfer between the host and DSP end.

The DSP program consists of the main function, the definition of thread, and also the message handler. The thread, which is a class with the procedure we want to execute, is the primary part at the DSP end. The DSP platform supports the execution of multiple threads. Multi-thread execution benefits only when the processor is idle during the program execution in the single-thread mode. In our case, the next data is fed to the DSP right after the present data is processed completely. Hence this function is needless to us. The structure described above is summarized by Figure 5.1.

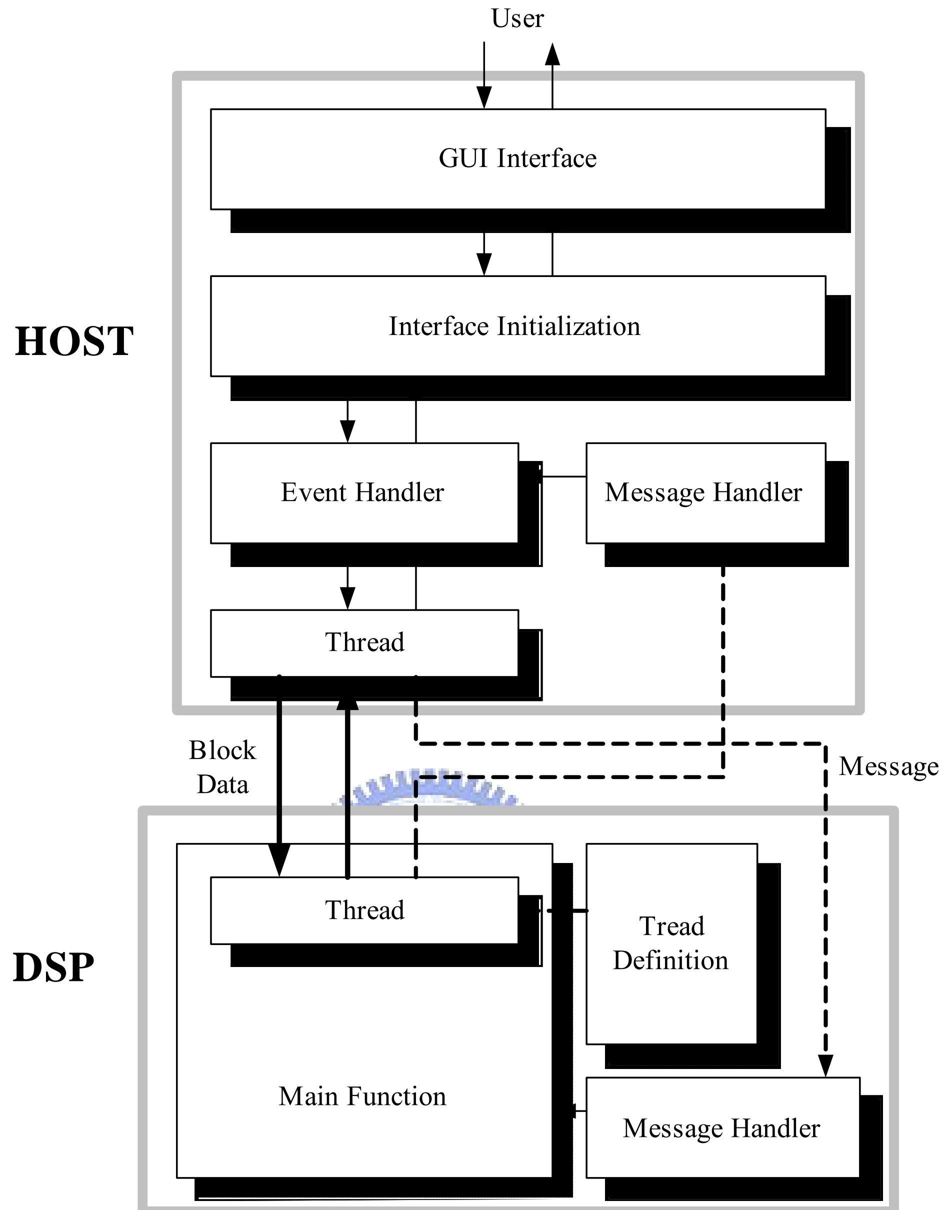


Figure 5.1: Structure of AMR Speech Codec Implementation on the Host and DSP

5.2.2 Execution Flow of AMR Implementation

We focus on the implementation of the AMR encoder program first. Its interface is shown in Fig. 5.2. This interface has one editorial text field and three buttons on the right side. The text field on their left side is used to show state messages to users. We can input a source rate in the editorial text field, which is supported by the AMR speech

encoder, and the default is 4.75 kbit/s. The button "Coff File…" is used to choose the path of the compiled bitstream file. The button "Download" is then clicked to download the chosen bitstream. The message "Download Complete." shall be shown in the left text field when the whole bitstream has been downloaded. "Transfer" is the third button for running the downloaded program, and the text field for importing the source rate is disabled until "Transfer" is clicked again to stop the program. If the program is stopped in the middle, and the source rate is changed by the user, our program is capable to update the AMR coding mode for the speech frames of rest. Moreover, some information, the number of blocks transferred, the byte rate, and the transferring state, shall be shown below during running the program.

The flowchart of the AMR encoder processing is shown in Fig. 5.4. First, the GUI interface should be initialized right after the program executing at the host end. All objects on the interface like text editing boxes, buttons, or check boxes, etc. are mapped to the parameters available to the program, and the events of objects also need to be assigned to the functions defined by the programers. Then we have to choose the path of compiled bitstream file of the AMR encoder program to download to the DSP baseboard. Once the bitstream file is download completely, the DSP end sends a login message to inform the host end, and we can start to execute our program by clicking the button "Transfer". The thread is generated when the data transfer begins. The imported source rate is also read and saved to a parameter to specify the AMR encoder processing mode. And the text field for importing the source rate is set non-active at the same time. The thread is used to read inputs and write outputs to files. It also manages sending blocks of data to DSP for encoding and receiving the processed blocks from DSP. Also the mode information is transmitted together with the speech data to achieve higher efficiency. At the host, one frame data is transmitted, and a flag which is set as long as "Transfer" is clicked again to stop the transfer is also inspected for each loop. This flag is used to decide whether the transfer is on or not. If the transfer is off, the flag shall be set, and the text field is set active for users to specify a new coding rate. The speech

frames of rest are encoding with the updated mode information after the transfer begins again.

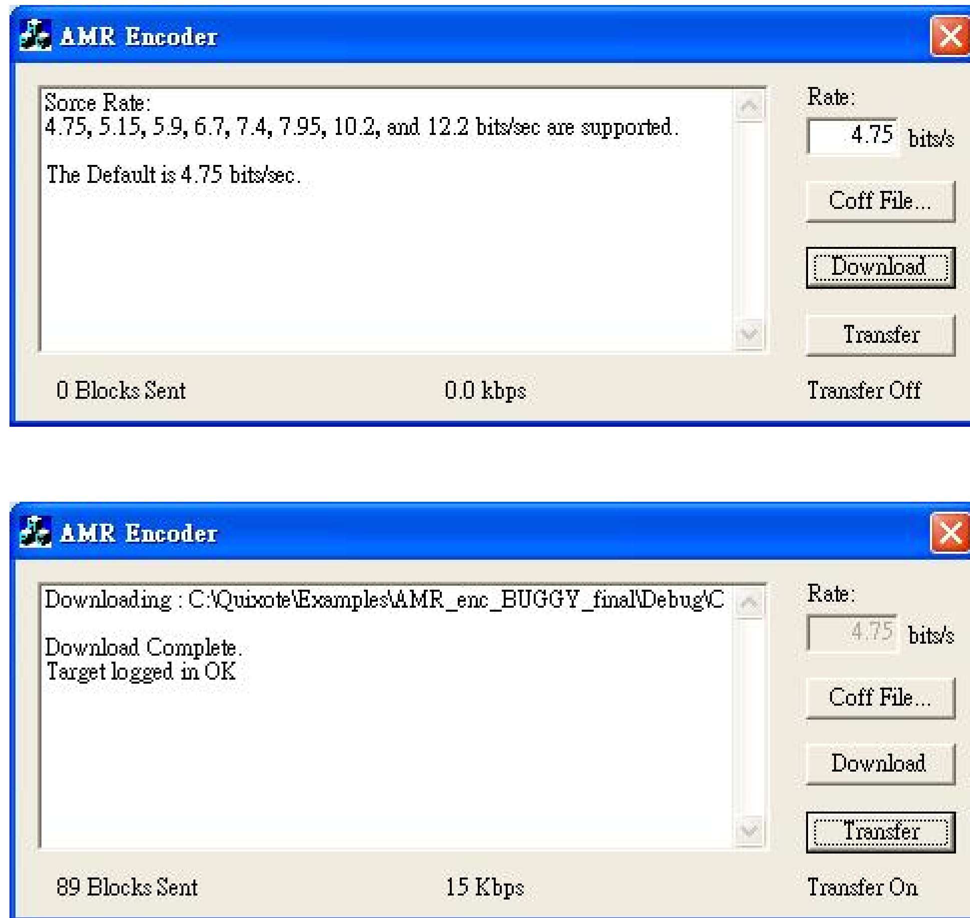


Figure 5.2: (a) Graphical Interface of the AMR Encoder Implementation. (b) A Snapshot of Running the Program.

At the DSP end, the thread with our AMR encoding procedure is generated and executed after the program bitstream is downloaded to the DSP baseboard completely. The program in the thread is also initialized for the memory allocation. Then, it waits to receive the data blocks from the host end. After the input data blocks are received, the program performs the AMR encoding process and transmits back the coded data. Finally, some parameters like the excitation signal of the previous frame shall be updated for the next loop.

The AMR decoder implementation is the same as that of the AMR encoder. However, the AMR coding mode is fixed in the coded bitstream, and hence no input text field on the interface is provided for the user to specify the AMR decoding mode. Its interface is shown as Fig. 5.3. The program execution flow is also the same as the encoder except that the encoder part is replaced by the decoder and the parts involving the coding mode should be deleted, so we do not describe here again.

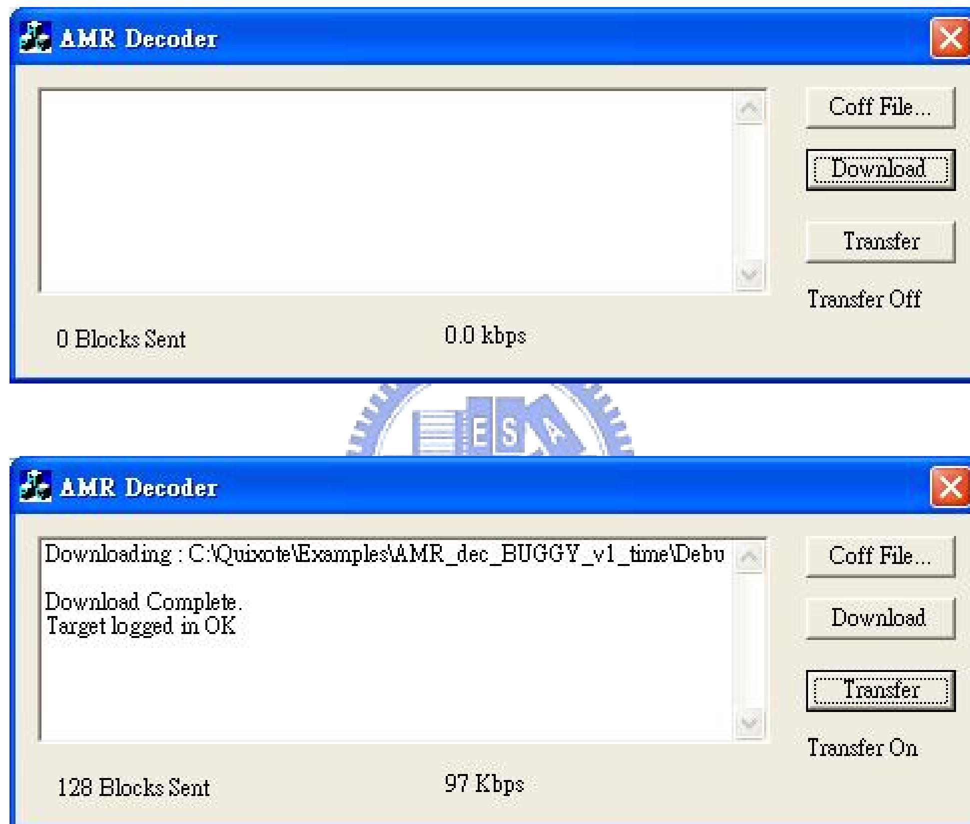


Figure 5.3: (a) Graphical Interface of the AMR Decoder Implementation. (b) A Snapshot of Running the Program.

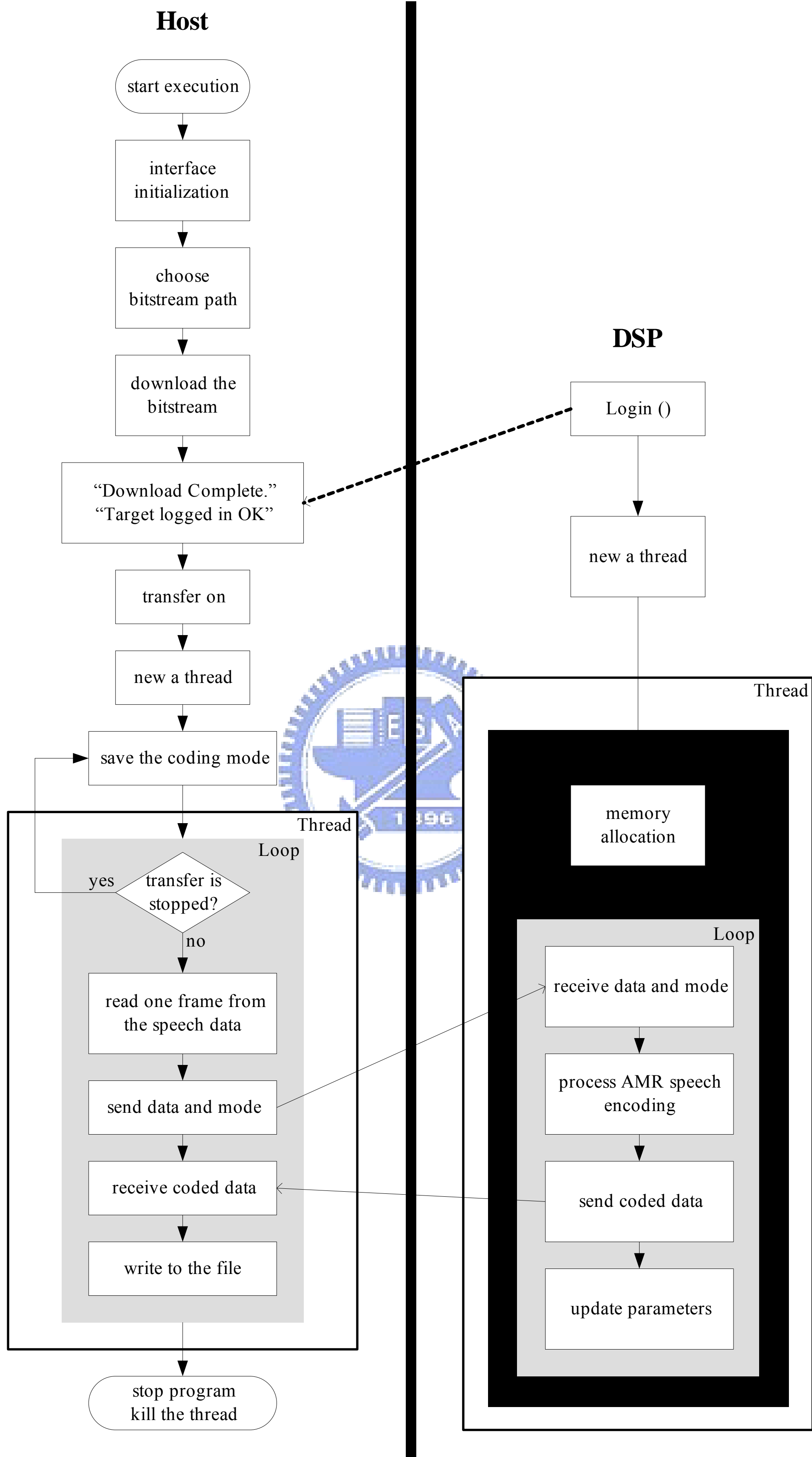


Figure 5.4: the Flowchart of the AMR Encoder Implementation

5.2.3 Performance Analysis

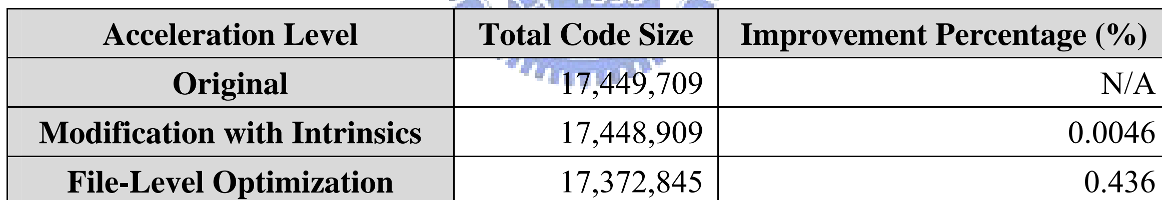
In this section, we present the execution time of our implementation for each source rate supported by the AMR codec. We use the test sequences provided by 3GPP. They are marked as “TSx” and described as follow [27]:

TS0: Synthetic harmonic signal. The pitch delay varies slowly from 18 to 143.5 samples. The minimum and maximum amplitudes are -997 and +971. 285 frames.

TS1: Female speech, active speech level: -19.4 dBov, flat frequency response, 301 frames.

TS2: Male speech, active speech level: -18.7 dBov, flat frequency response, 224 frames.

We first show the code size of the AMR codec downloaded to the DSP baseboard at the different acceleration levels in Table 5.7 and 5.8.



Acceleration Level	Total Code Size	Improvement Percentage (%)
Original	17,449,709	N/A
Modification with Intrinsics	17,448,909	0.0046
File-Level Optimization	17,372,845	0.436

Table 5.7: Code Size of the AMR Encoder for Different Acceleration Level

Acceleration Level	Total Code Size	Improvement Percentage (%)
Original	17,337,934	N/A
Modification with Intrinsics	17,334,566	0.019
File-Level Optimization	17,280,686	0.311

Table 5.8: Code Size of the AMR Decoder for Different Acceleration Level

The total code size contains the on chip memory, L2 cache, and SDRAM. Referring to Tables 5.7 and 5.8, it is shown that the code size dose not benefit much for

all acceleration levels, and the code size is about 17.4 MB for the encoder and 17.3 MB for the decoder.

Next we present the execution time for each test sequence under different source rates and acceleration level. To make it perspicuous, we divide the result data into two subsections, the encoder and decoder part, and also attach the improvement percentage between the different acceleration level for each table.

5.2.3.1 AMR Encoder Performance Analysis

We use the time or clock function to obtain the processing time for each test sequence at the host end. The time function is inserted before sending data blocks and after receiving coded blocks to count the coding duration. Hence this duration consists of the AMR encoding time and the block transfer overhead. It is measured and sown as follow:

E S
1896

1. the Original AMR Encoder (Provided by 3GPP)

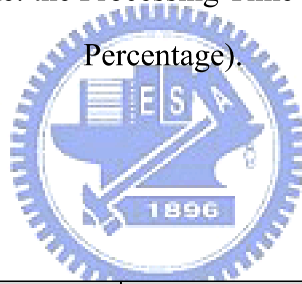
Source Rate (bits/sec)	Encoding Time (ms/frame)		
	TS0	**TS1**	**TS2**
4.75	42.06	42.19	42.11
5.15	33.31	33.34	33.40
5.9	37.21	37.33	37.29
6.7	43.89	44.12	43.86
7.4	41.36	41.52	41.44
7.95	43.19	43.62	43.55
10.2	43.25	43.39	43.23
12.2	45.01	45.15	45.25
Average	41.16	41.33	41.27

Table 5.9: Execution time of the DSP Implementation under Different Source Rate for Each Test Sequence

2. Improved AMR Encoder with Intrinsics

Source Rate (bits/sec)	TS0		TS1		TS2	
	ms/frame	%	ms/frame	%	ms/frame	%
4.75	39.46	6.18	39.49	6.40	39.30	6.67
5.15	31.41	5.70	31.58	5.28	31.56	5.51
5.9	35.07	5.75	35.14	5.87	35.05	6.01
6.7	41.22	6.08	41.32	6.35	41.17	6.13
7.4	38.97	5.78	39.06	5.92	38.98	5.94
7.95	40.69	5.79	40.99	6.03	40.95	5.97
10.2	40.62	6.08	40.79	5.99	40.73	5.78
12.2	42.31	6.00	42.45	5.98	42.43	6.23
Average	38.72	5.93	38.85	6.00	38.77	6.06

Table 5.10: Execution time of the DSP Implementation under Different Source Rate for Each Test Sequence (ms/frame: the Processing Time for one frame, %: Improvement Percentage).



3. File-Level Optimization

Source Rate (bits/sec)	TS0		TS1		TS2	
	ms/frame	%	ms/frame	%	ms/frame	%
4.75	14.09	64.29	14.07	64.37	14.13	64.05
5.15	11.18	64.41	11.25	64.38	11.17	64.61
5.9	12.83	63.42	12.81	63.55	12.83	63.40
6.7	15.00	63.61	15.04	63.60	15.02	63.52
7.4	14.34	63.20	14.37	63.21	14.31	63.29
7.95	14.93	63.31	15.07	63.23	15.02	63.32
10.2	14.69	63.84	14.74	63.86	14.67	63.98
12.2	15.28	63.89	15.24	64.10	15.24	64.08
Average	14.04	63.74	14.07	63.78	14.05	63.76

Table 5.11: Execution time of the DSP Implementation under Different Source Rate for Each Test Sequence (the Lists Representation is the Same as Table 5.10).

The format of all the three test sequences is 8 kHz sampling rate, 16 bits/sample, and 160 samples/frame speech. The duration of one frame is 20 ms. Referring to Tables 5.8, 5.9, and 5.10, it is observed that the coding time relates with the AMR coding mode more than the different test sequences. The final AMR encoder implemented on the DSP baseboard after our acceleration takes the coding time about 14.05 ms/frame and is improved up to 65.94% with respect to the original for average. Hence it reaches the coding speed of real time. Moreover, it is noted that the coding time shown in the three tables contains the time of the data transfer between the host and DSP end, and the data transfer time is measured alone about 0.28 ms/frame for average. Hence the pure AMR encoding time is about 13.77 ms/frame.

5.2.3.2 AMR Decoder Performance Analysis

We measure the AMR decoding time by the same method as the encoder.

1. the Original AMR Decoder (Provided by 3GPP)

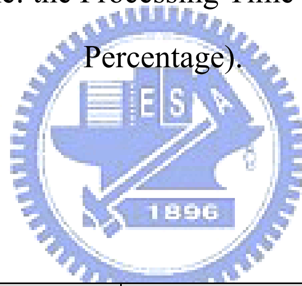
Source Rate (bits/sec)	Encoding Time (ms/frame)		
	TS0	TS1	TS2
4.75	6.25	6.29	6.26
5.15	6.33	6.22	6.26
5.9	6.33	6.22	6.26
6.7	6.40	6.39	6.26
7.4	6.18	6.12	6.04
7.95	6.32	6.36	6.35
10.2	6.15	6.29	5.99
12.2	6.57	6.39	6.30
Average	6.32	6.29	6.22

Table 5.12: Execution time of the DSP Implementation under Different Source Rate for Each Test Sequence

2. Improved AMR Decoder with the Intrinsics

Source Rate (bits/sec)	TS0		TS1		TS2	
	ms/frame	%	ms/frame	%	ms/frame	%
4.75	5.90	5.60	5.92	5.88	5.81	7.19
5.15	5.90	6.79	5.86	5.79	5.90	5.75
5.9	5.94	6.16	5.92	4.82	5.86	6.39
6.7	5.97	6.72	5.89	7.82	5.90	5.75
7.4	5.84	5.50	5.82	4.90	5.72	5.30
7.95	5.94	6.01	5.96	6.29	5.99	5.67
10.2	5.66	7.97	5.89	6.36	5.68	5.18
12.2	6.05	7.91	5.95	6.89	5.94	5.71
Average	5.90	6.65	5.90	6.20	5.85	5.95

Table 5.13: Execution time of the DSP Implementation under Different Source Rate for Each Test Sequence (ms/frame: the Processing Time for one frame, %: Improvement Percentage).



3. File-Level Optimization

Source Rate (bits/sec)	TS0		TS1		TS2	
	ms/frame	%	ms/frame	%	ms/frame	%
4.75	2.46	58.30	2.46	58.45	2.37	59.21
5.15	2.43	58.81	2.46	58.02	2.37	59.83
5.9	2.39	59.76	2.43	58.95	2.42	58.70
6.7	2.46	58.79	2.46	58.23	2.42	58.98
7.4	2.43	58.39	2.36	59.45	2.42	57.69
7.95	2.46	58.59	2.46	58.72	2.42	59.60
10.2	2.39	57.77	2.43	58.74	2.37	58.27
12.2	2.46	59.34	2.43	59.16	2.42	59.26
Average	2.44	58.64	2.44	58.64	2.40	58.97

Table 5.14: Execution time of the DSP Implementation under Different Source Rate for Each Test Sequence (the Lists Representation is the Same as Table 5.13).

The final AMR decoder implemented on the DSP baseboard after our acceleration takes the processing time about 2.43 ms/frame and is improved up to 61.31% with respect to the original for average. It matches the real time requirement. The data transfer time alone is also about 0.28 ms/frame. Hence, the pure AMR decoding time is about 2.15 ms/frame.

Chapter 6

Implementation and Acceleration of 802.16a Reed-Solomon Decoder on TI DSP Platform

After introducing the AMR speech coding part, in this chapter, we are going to discuss the second major topic – the implementation and acceleration of the specified Reed-Solomon coding scheme on the same DSP platform. The AMR codec and the RS coding scheme are both specified in the IEEE 802.16a wireless communication standard. The AMR codec belongs to the source coding part, while the RS coding belongs to the channel coding part. The RS coding scheme connects directly to the block of AMR speech coding and provides it with the ability against channel errors. The acceleration work of the RS code would be mainly focused on the decoder because it is more complicated than the encoder.

At first, as the general flow of acceleration, the structure and profile of the original RS decoder is introduced. Then we describe the algorithms proposed to obtain the further improvement. Also, an alternative procedure for RS decoding, the remainder decoding algorithm [30] [31] [35], is implemented for comparison with the former system. Finally, we report the total effort of acceleration and the DSP implementation of our system.

6.1 Acceleration on Reed-Solomon Decoder

We first generate a computational profile by using the CCS built-in profiler to obtain the execution cycles. Then, we identify which parts of our program consume the most execution time based on the profile data, and hence we pay our attention on these parts to speed up the whole program. In the following subsections, the processing flow of our RS decoder program on TI DSP platform is divided into several procedures to improvement work.

6.1.1 Profiling the Original RS Decoder

The starting point of our RS decoder is the version that has been improved using several acceleration techniques on the well-known RS decoding flow. It was written by Y.-T. Lee in 2004 for his MS thesis [20]. We call it the Lee decoder. The well-known RS decoding flow has been described in Chapter 3, which consists of the four procedure units:

- Syndrome computation
- Berlekamp-Massey algorithm (BM algorithm)
- Chien search
- Forney algorithm

The Lee decoder program we intend to accelerate uses a look-up table to realize the Galois field multiplier and has improved the BM algorithm and Chien search by some fast versions.

The inversion of discrepancy needed during the computation of the original BM algorithm is complex and time-consuming due to the requirement of chain multiplication. Hence the inverse-free BM algorithm is used to reduce the inversion operations to one time. Compared to the original BM algorithm, the Lee decoder program has greatly reduced the number of inversion operations.

Two features of Chien search are used to improve it. One feature is early termination. We can substitute elements to find the roots until the number of roots match the order of the errata locator polynomial instead of substituting all the elements. The other is skipping nonused position in Chien search. The inputs of different block sizes defined in IEEE 802.16a standard should be padded with zeros in the (255, 239, 8) RS encoding. Thus, we also have to pad the same zeros to the input at the RS decoder. Therefore, the positions of zero padding are never wrong and cannot be the roots of the errata locator polynomial. Those positions can be skipped in checking roots.

The improvement described above has been done in the version of RS decoder we start with, and we call this version the Lee RS decoder for convenience. The profile of the Lee RS decoder is shown in Table 6.1 without compiler level optimization.

<table>
<tr><th colspan="2">Function Name</th><th>Code Size</th><th>Cycle</th><th>Percentage (%)</th></tr>
<tr><td colspan="2">Syndrome Computation</td><td>480</td><td>249,294</td><td>80.98</td></tr>
<tr><td colspan="2">BM Algorithm</td><td>1,920</td><td>23,962</td><td>7.78</td></tr>
<tr><td rowspan="2">Chien Search</td><td>Worst Case</td><td rowspan="2">804</td><td>25,375</td><td>8.24</td></tr>
<tr><td>Best Case</td><td>902</td><td>N/A</td></tr>
<tr><td colspan="2">Forney Algorithm</td><td>1,064</td><td>9,211</td><td>2.99</td></tr>
</table>

Table 6.1: Profile of the Lee RS Decoder

The "Percentage" in Table 6.1 represents the execution cycles of individual functions in percentage of the whole RS decoder. The Chien search is discussed for two cases because it may early terminate when the number of roots reaches the order of the errata locator polynomial in the Lee RS decoder. The worst case represents that one of the errors happens in the last position, and therefore we have to substitute all the elements for finding the last roots. Respectively, the best case represents that no error happens. It is clear that the possibility of the best case is very low. To insure real-time operation, we focus mainly on the worst case, and the details will be discussed in the following sections.

Referring to Table 6.1, it shows that the procedures of syndrome computation and Chien search take the most execution time, and our acceleration work on them are described in the next section.

6.1.2 Modifications of RS Decoder

6.1.2.1 Syndrome Computation Improvement

The syndrome can be formally defined as follow:

S_i = R mod G where i = (0, 1, 2, 3, …, 15) for GF(2^8)

The received codeword may be expressed in polynomial form as follow:

$R_i = r_0X^{N-1} + r_1X^{N-2} + \ldots + r_{N-1}$

Where the length of the received codeword is N. In our case of (255, 239, 8) RS code, N equals to 255. Let the first 2T powers of beta be specified as shown below, where beta = {β_0, β_1, ⋯, β_{15}}. The 16 syndromes are now expanded as follows:

$S_0 = r_0\beta_0^{N-1} + r_1\beta_0^{N-2} + \ldots + r_{N-2}\beta_0^{1} + r_{N-1}$

$S_1 = r_0\beta_1^{N-1} + r_1\beta_1^{N-2} + \ldots + r_{N-2}\beta_1^{1} + r_{N-1}$

……

$S_{15} = r_0\beta_{15}^{N-1} + r_1\beta_{15}^{N-2} + \ldots + r_{N-2}\beta_{15}^{1} + r_{N-1}$

It can be seen that computing the syndromes amounts to polynomial evaluation at the roots as defined by beta. In the Lee RS decoder, this is done recursively using the Horner's rule. For example, the recursive computation of S_0 is shown below:

$S_0 = (\ldots((r_0\beta_0 + r_1)\beta_0 + r_2)\beta_0 + \ldots r_{N-2})\beta_0 + r_{N-1}$

According to the computation procedure shown in Figure 6.1, the C code implementation involves two loops, an outer loop that iterates once for every syndrome and an inner loop that iterates over all the field elements. In order to obtain a better performance from the architecture, we unroll the inner loop.

```
for (j = 1; j <= 16; j++) {
    for (i = 0; i < 255; i++) {
        product = gf_mul_tab(Alpha_to[B0-1+j],s[j]);
        s[j] = product ^ data[i];
    }
}
```

Figure 6.1: the C Code of the Syndrome Computation in the Lee Decoder

We should choose a way to unroll the loop efficiently. Here is an approach similar to that of a radix-4 FFT [28]. The received codeword is read starting at locations 0, N/4, N/2, and 3N/4. Horner's rule is now applied recursively to all four parts of the syndrome polynomial using the input data read in all four locations (N/4 – 1) times. The syndrome polynomials are thus segmented as shown below:

$s_0 = r_0\beta_0^{63} + r_1\beta_0^{62} + \ldots + r_{62}\beta_0^{1} + r_{63}$

$s_1 = r_{64}\beta_0^{63} + r_{65}\beta_0^{62} + \ldots + r_{126}\beta_0^{1} + r_{127}$

$s_2 = r_{128}\beta_0^{63} + r_{129}\beta_0^{62} + \ldots + r_{190}\beta_0^{1} + r_{191}$

$s_3 = r_{192}\beta_0^{63} + r_{193}\beta_0^{62} + \ldots + r_{255}\beta_0^{1} + r_{256}$

The four segments use the same powers of beta, and it means that only one beta value has to be read in one iteration for computing the terms of these four polynomials. Then, these four segments has to be weighted and cumulated as follow to obtain the syndrome we want:

$S_0 = s_0\beta_0^{192} + s_1\beta_0^{128} + s_2\beta_0^{64} + s_3$

It should be noticed that our received codeword length is 255, so we have to assign a zero to r_0 to use this method. This method has the benefit in the reduction of the memory access of beta values. It is also able to reduce the number of the inner loops. The profile data of the modified syndrome computation is compared in Table 6.2.

Version	Code Size	Cycle	Improvement Percentage (%)
Lee Decoder RS Syndrome Computation	480	249,294	N/A
Modified RS Syndrome Computation	748	172,607	30.76
Using the Intrinsic _gmpy4	680	47,486	72.49
Improved with More Intrinsics	816	34,058	28.28
Compiler File-Level Opt.	564	5,503	83.84
Compiler File-Level Opt. (Lee Decoder)	296	104,378	58.13

Table 6.2: Improvement of Syndrome Computation

The list of the modified RS syndrome computation in Table 6.2 is the version using the method we propose here, and it improves the original one up to 30.76% of cycles without compiler-level optimization. The versions using the intrinsics are also listed in Table 6.2, where "_gmpy4" is the intrinsic for Galois field multiplier [23], and the more intrinsics means we further pack four symbols into a 32-bit integer by the other intrinsics and perform four Galois field multiplications simultaneously. Finally we turn on the file-level optimization and obtain the improvement percentage 97.79% compared to the Lee decoder syndrome computation. The improvement percentage of the Lee decoder syndrome computation is only 58.13% after the file-level optimization and is lower than the syndrome computation with our modification.

6.1.2.2 Chien Search Improvement

The Chien-search method is used to find the roots of an errata locator polynomial. It requires multiplication for each term in calculating the errata locator polynomial. Hence, we choose the Berlekamp-Rumsey-Solomon (BRS) algorithm together with the Chien-search method proposed in [29] for our RS decoder. The new fast algorithm makes the root-finding problem quite practical and efficient because it can eliminate a

lot of multiplications and is structured regularly for compiler to achieve the software pipeline more easily.

The BRS algorithm is first described below, which is an algorithm in finding the roots of a special class of polynomials as proposed by [29]. Before introducing the algorithm, here are two definitions and a theorem that are needed for this algorithm:

Definition 1: the polynomial L(y) over $GF(2^m)$ is called a p-polynomial for p = 2 iff

$$L(y) = \sum_i c_i y^{2^i}$$

where c_i are restricted to $GF(2^m)$ and the exponents are restricted to be the powers of two.

Definition 2: a polynomial A(y) over $GF(2^m)$ is called an affine polynomial iff

$$A(y) = L(y) + \beta$$

where L(y) is a p-polynomial as defined previously and $\beta \in GF(2^m)$.

Theorem 1: let $y \in GF(2^m)$ and let $\alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{m-1}$ be a standard basis. If y is represented in the standard basis, i.e., if

$$y = \sum_{k=0}^{m-1} y_k \alpha^k$$

where $y_k \in GF(2)$, then

$$L(y) = \sum_{k=0}^{m-1} y_k L(\alpha^k)$$

Using Theorem 1, a simplified algorithm is proposed to find the roots of an affine polynomial, which needs only to compute the eight values $L(\alpha^0)$, $L(\alpha^1)$, …, and $L(\alpha^7)$ instead of all the 255 elements needed in the Chien search. The elements of rest simply need to be judged whether the term $L(\alpha^k)$ should be cumulated or not according to each y_k. This is done by checking the k-th bit of the element y.

It can be observed that most of the Galois field multiplications are eliminated. It is only needed to compute the eight terms imported with the standard bases. The BRS algorithm is used only for solving affine polynomials. Hence, in our method, we first arrange and sort our errata locator polynomial into an affine polynomial and the

remainder, and then the value of the affine polynomial is obtained by the BRS algorithm and the roots of the remainder is by the Chien search. If their values are equal for a Galois field element, we can claim a root is found. Note that this method benefits only when the order of the errata locator polynomial is not more than eleven [29].

Function Version	Code Size	Cycle	
		Worst Case	Best Case
Lee Decoder Chien Search	804	25,375	902
Modified Chien Search	1,268	14,013	4,248

Table 6.3: Profile of Chien Search without the Intrinsics and Compiler Optimization

Function Version	Code Size	Cycle	
		Worst Case	Best Case
Lee Decoder Chien Search	856	4,186	345
Modified Chien Search	960	1,100	183

Table 6.4: Profile of Chien Search with _gmpy4 and File-Level Optimization

Table 6.3 and Table 6.4 show the comparison of the Lee decoder Chien search and the modified one by the method we describe. In Table 6.3, it is the case without using the intrinsics and any compiler-level optimization, where the modified one is more efficient than the original in the worst case but is slower in the best case because the overhead of codes is increased to rearrange our errata locator polynomial. However, the best case is of very low probability. We apply the intrinsic "_gmpy4" and the file-level optimization to the two functions, and shown as Table 6.4, the modified Chien search is always more efficient than the Lee decoder Chien search. The improvement is up to 73.72% in the worst case and 46.96% in the best case because the most of Galois field multiplications are substituted in the modified Chien search to achieve the software pipeline more easily.

6.1.3 Performance Analysis

In this section, we present the simulation profile generated by the CCS built-in profiler for our RS decoder specified in IEEE 802.16a. The results of all improvements described formerly are also shown in the simulation profile, and the one which involves the efforts of all the former improvements is called the modified RS decoder on the list.

Decoder Version	Code Size	Cycle	Improvement Percentage (%)
Lee RS Decoder	5284	447,109	N/A
Using the Intrinsics	4936	238,050	46.76
Modified RS Decoder	5584	121,466	48.97
Compiler File-Level Opt.	5048	11,650	90.41
Compiler File-Level Opt. (Lee RS Decoder)	4732	121,169	72.90

Table 6.5: Simulation Profile for RS Decoder

Referring to Table 6.5, the cycles of the RS decoder are measured under the worst case condition, i.e., all elements are searched in the Chien search, and all the symbols are decoded correctly. It can be observed that in the case without the file-level optimization, the RS decoder with our improvement is accelerated up to 48.97% even compared to the one with the intrinsics. Respectively, it is accelerated up to 72.83% compared to the Lee RS decoder . The file-level optimization can further obtain 90.41% of acceleration. The final speed corresponds to 1.85 Mbytes/sec. The improvement of the Lee decoder only with the file-level optimization is also attached.

We also measure the speed and the ratio of correct decoding through the AWGN channel of the different SNR. Here we generate random data for the input to the RS encoder and pass the coded data through the convolutional coder and then the AWGN channel. At the receiver end, the soft-decision Viterbi decoder recovers the received data into the RS coded blocks. Then, we start to decode those RS blocks and count their decoding time. The process in the above is repeated ten times to make the results more accurate. The convolutional coder and Viterbi decoder used here are the ones designed

in IEEE 802.16a standard and are described in Chapter 3. We focus on the RS decoding cycles under different channel conditions, and the results are shown in Table 6.6. The relationship is plotted as Fig. 6.2 for the decoding cycle versus SNR and Fig. 6.3 for the correct decoding ratio versus SNR.

E_S/N_0 (dB)	Correct Decoding Ratio (%)	Decoding Cycle
7	100	11073
6.5	100	11574
6	96.43	12646
5.5	85.71	13181
5	67.86	14221
4.5	35.71	15030
4	7.14	15435
3.5	0	15269
3	0	15264

Table 6.6: the Decoding Ratio and Cycle under the Channel with Different SNR

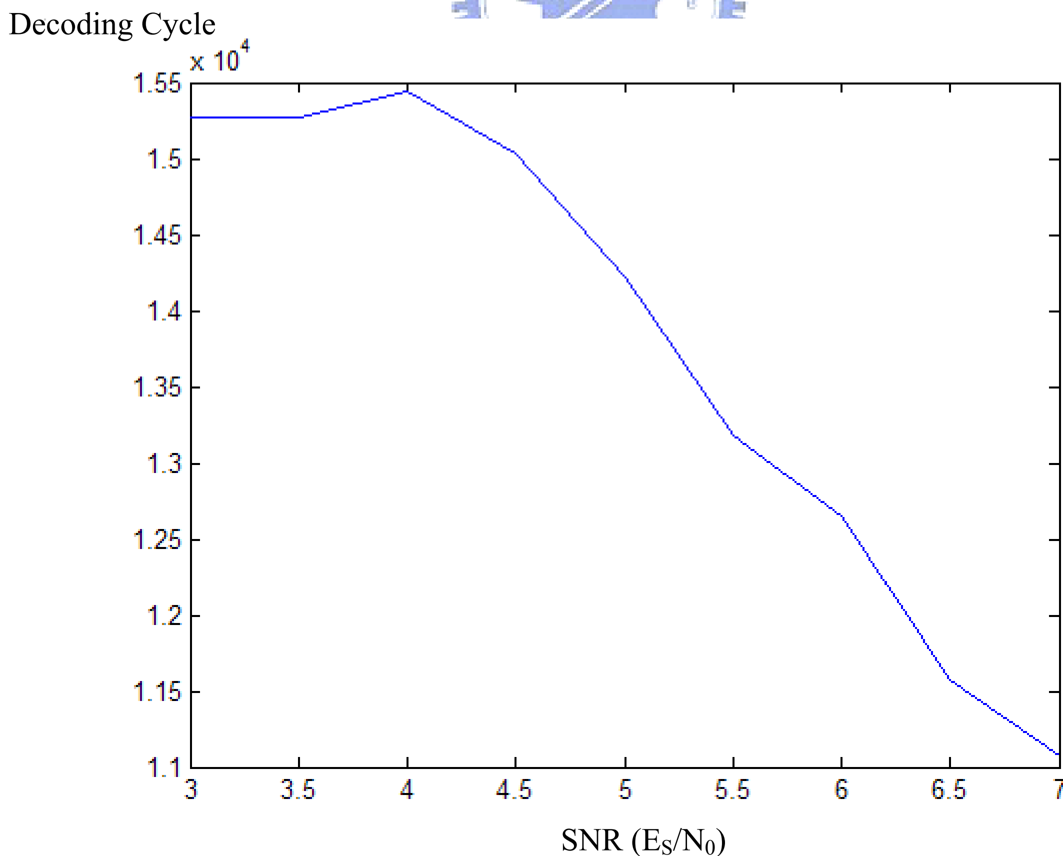


Figure 6.2: the Plot of the Decoding Cycle versus SNR

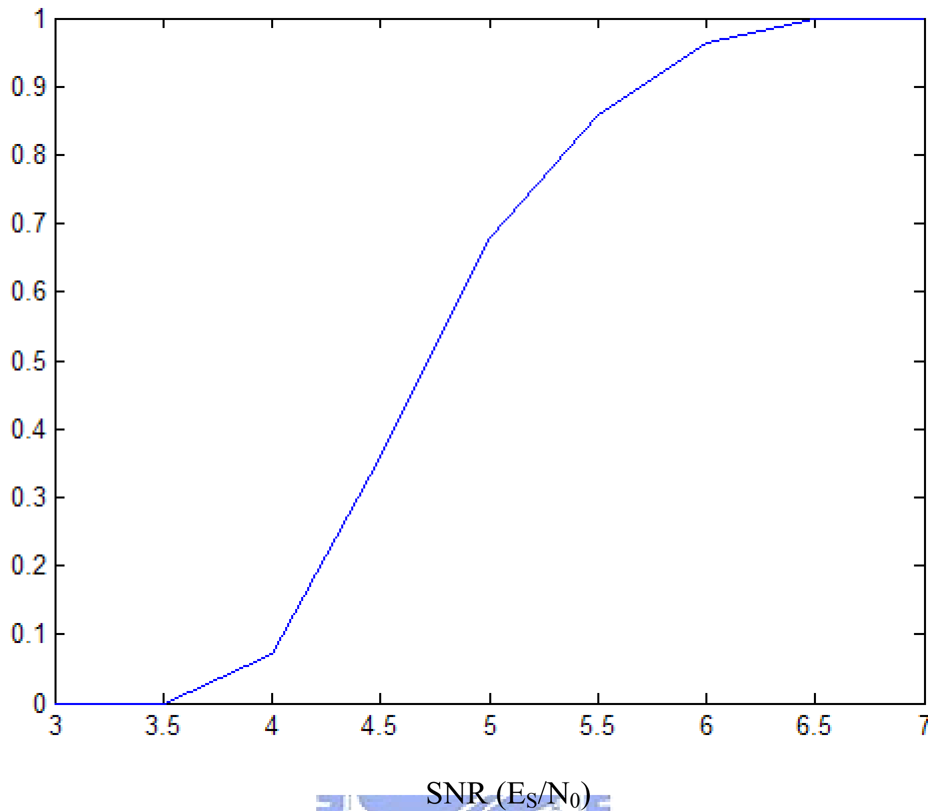


Figure 6.3: the Plot of the Correct Decoding Ratio versus SNR

It is clear that the decoding cycles are decreased and the correct decoding ratio is increased as the SNR goes up. The reason for the decrement of the decoding cycles is that because more error locations should be searched and more error values should be corrected, processing time is higher. The Chien search shall go through all the elements for the error locations but the Forney algorithm is not further executed when the number of errors is reaching the decoding capability for our RS decoder. It is why the decoding cycles of the zero correct decoding ratio are slightly less than some case with non-zero correct decoding ratios in Table 6.6.

6.2 Remainder Decoding Algorithm for RS Decoder

The decoding algorithm for RS codes has been investigated for a long time. Both the Berlekamp-Massey and Euclidean algorithms are well known, which solve the key-equations for RS codes. Generally, the key-equation can be generated by syndrome sequences, which are derived from the received codewords. Therefore, the syndromes have to be calculated. However, the syndrome calculation takes a large amount of execution time as shown in the profile data in the earlier sections. In 1983, L. Welch and E. R. Berlekamp proposed a new decoding algorithm, the remainder decoding algorithm [37], for RS codes without the need of computing the syndromes, and hence it becomes an alternative and popular algorithm that it is worthy of our attention and study. They presented a new key-equation and the solving algorithm for decoding RS codes. It should be noted that the proposed key-equation is quite different from the conventional key-equation which was proposed by E. R. Berlekamp [38]. In the next subsections, we introduce the decoding flow for the remainder decoding algorithm and write the C codes for it. The performance analysis of the system is also shown and is compared in the final subsection.

6.2.1 Remainder Decoding Algorithm

The remainder decoding algorithm represents a decoding algorithm, which dose not compute the syndromes. There are two main points. One is that a new key-equation has been derived. This is a relationship between the coefficients of remainder polynomial and the errors occurring in a received codeword. It is very special that it is quite different from the conventional key-equation. The other is that Welch and Berlekamp have proposed an efficient algorithm, Welch-Berlekamp (WB) algorithm, for solving the new key-equation. The solution technique we adopt is proposed in [32], a modified version of the original WB algorithm. It is similar to but an improved version of the WB algorithm. Here, we call it the modified WB algorithm for convenience. Now, we shall briefly describe the decoding algorithm. However, the

proof for this algorithm dose not be presented here, and it can be find in [30], [31], and [35].

At first, we re-encode the received codeword R(x) and yield the remainder polynomial

$$r(x) = (R(x) \bmod g(x)),$$

where g(x) is the generator polynomial same as the one used in the encoder. A few polynomials are derived for the remainder decoding as follows:

$$r_j W(\alpha^j) = \frac{N(\alpha^j)}{g'(\alpha^j)}, j = 0, \ldots, d\text{-}2,$$

where r_j is the j-th coefficient of the polynomial r(x), W(x) is the error-locator polynomial, and N(x) is a unique polynomial whose degree is less than that of W(x). The formal derivative applied here is defined as [30]

$$g'(x) = \sum_{i \in E} \alpha^i \prod_{\substack{k \in E \\ i \neq k}} (\alpha^k x - 1)$$

where E is the set of indices for which e_i, the error pattern in the position i, is nonzero,

$$E = \{i \mid e_i \neq 0\}$$

The RS decoding can then be formulated as a problem of solving the set of the key equations

$$N(\alpha^j) = W(\alpha^j) r_j g'(\alpha^j), \text{ for } j = 0, \ldots, d\text{-}2$$

Our goal is to find the unique pair of polynomials (W, N). The error locations correspond to the roots of W(x), and we denote it as Z_j. If Z_j is a message location, then the error values are given by the following equation:

$$Y_j = \beta(Z_j) \frac{N(Z_j)}{W'(Z_j)}$$

where

$$\beta(Z_j) = \frac{1}{\prod_{i=0}^{d-2} (\alpha^i - Z_j)}$$

The values of $g'(\alpha_j)$ and $\beta(Z_j)$ can be calculated in advance when the specification of the RS code system is fixed.

6.2.2 Program Flow and Performance Analysis

In our program, first we re-encode the received codeword with the LFSR structure. Then the algorithm proposed in [32] is used to solve the key equations for obtaining the pair (W, N). Then the roots of the error-locator polynomial should be found. We can apply the Chien search to solve this problem. Finally, the error values can be derived by using the equation described in the previous subsections or the Forney algorithm. Here we choose the Chien search and Forney algorithm to complete the last half of our program flow, and hence it equals the last procedures used in the original RS decoder. We only need to compare the re-encoding part to the syndrome computation and the key-equation solving part to the Berlekamp-Massey algorithm. For the former, they both compute the necessary information for RS decoding. For the latter, they both use the information computed by the former to solve the constrained polynomial congruence. It is noted that there is an additional procedure, the re-encoding, in the remainder decoding algorithm although it is claimed that the syndrome computation is not needed for the remainder decoding algorithm. The comparisons simulated by CCS built-in profiler are presented as follow:

Procedure	Code Size	Cycle
Syndrome Computation	212	149,972
Re-Encoding	436	191,484
Inverse-Free BM Algorithm	1,716	14,046
Modified WB Algorithm	2,036	33,683

Table 6.7: Comparison of the Remainder Decoding Algorithm and the Lee Decoder (without the Intrinsics)

Table 6.7 is the comparison of the remainder decoding with the Lee RS decoder. In Table 6.8, the re-encoding and the modified WB algorithm with the improvement of the

intrinsics are compared to the Lee decoder. Both Table 6.7 and Table 6.8 are obtained with the file-level optimization.

Procedure	Code Size	Cycle	Improved Percentage (%)
Re-Encoding without Intrinsics	436	191,484	N/A
Re-Encoding with Intrinsics	996	2,926	98.47
Modified WB Algorithm without Intrinsics	2,036	33,683	N/A
Modified WB Algorithm with Intrinsics	2,208	2,672	92.07

Table 6.8: Profile of the Improved Remainder Decoding Algorithm

Referring to Table 6.7, it seems that the C code implementation of the remainder decoding algorithm on the DSP platform is less efficient than that of the original RS decoder. For the re-encoding, its structure consists of a LFSR and the calculation of multiplying r_j by $g'(\alpha_j)$. The LFSR is implemented as the method of the syndrome computation in the original RS decoder. However the multiplication of r_j by $g'(\alpha_j)$ adds the complexity to the re-encoding procedure and leads to the fact that the re-encoding takes more cycles than the syndrome computation. For the modified WB algorithm, there are two primary factors reducing its performance:

- *The over too-many memory accesses are caused by the operation of array. The modified WB algorithm needs to initialize four arrays and imports six arrays for operation while the inverse-free BM algorithm only needs to initialize two arrays and imports three arrays for operation. Furthermore, it contains the multiplication of polynomials, the swap of polynomials, and the calculation of polynomials imported by some value. These operations cost a large number of memory accesses, too.*
- *The poor structure of the modified WB algorithm is difficult to form software pipelines by the compiler. To complete the operations described in the*

previous list, we have to call the other functions. However, the compiler dose not do software pipelining for the loops which contain a loop or a function call. Moreover, the structure of the loop content must be simple enough to activate the software pipeline, but it seems that most loops in the modified WB algorithm is more complicated than that in the inverse-free BM algorithm.

To eliminate the above shortcomings, the intrinsics are used here to reduce these problems. In the re-encoding, it takes a large amount of time to add the previous value of the register and to shift in each register in the LFSR, and it also increases the dependency between the iterations. We employ adding register values and shifting simultaneously as much as possible by the intrinsics as illustrated by Fig. 6.4.

The structure of LFSR used to implement the re-encoding procedure is shown as Fig. 3.8 in Chapter 3. Referring to Fig. 6.4, at first we calculate the feedback symbol by performing modulo-2 addition of the LFSR fifteenth register in the previous iteration for the present iteration, and pack the feedback symbol for the present iteration into a 32-bit integer variable by using the intrinsics. We can pack the coefficients of RS generator polynomial into the individual four 32-bit variables by the same method and perform the multiplication on them with the feedback symbol in one iteration. These four packages of results continue to perform the modulo-2 addition with the individual 32-bit variables packed with the fifteen registers of LFSR. Here, we call these variables packed with the registers in the LFSR the register variables. Then we save the results back to the register variables. At last, we use the intrinsics to right shift to each register variable one symbol size (one byte) to the next register variable, and the symbol shifted out of the end register variable is used for calculating the feedback symbol for the next iteration.

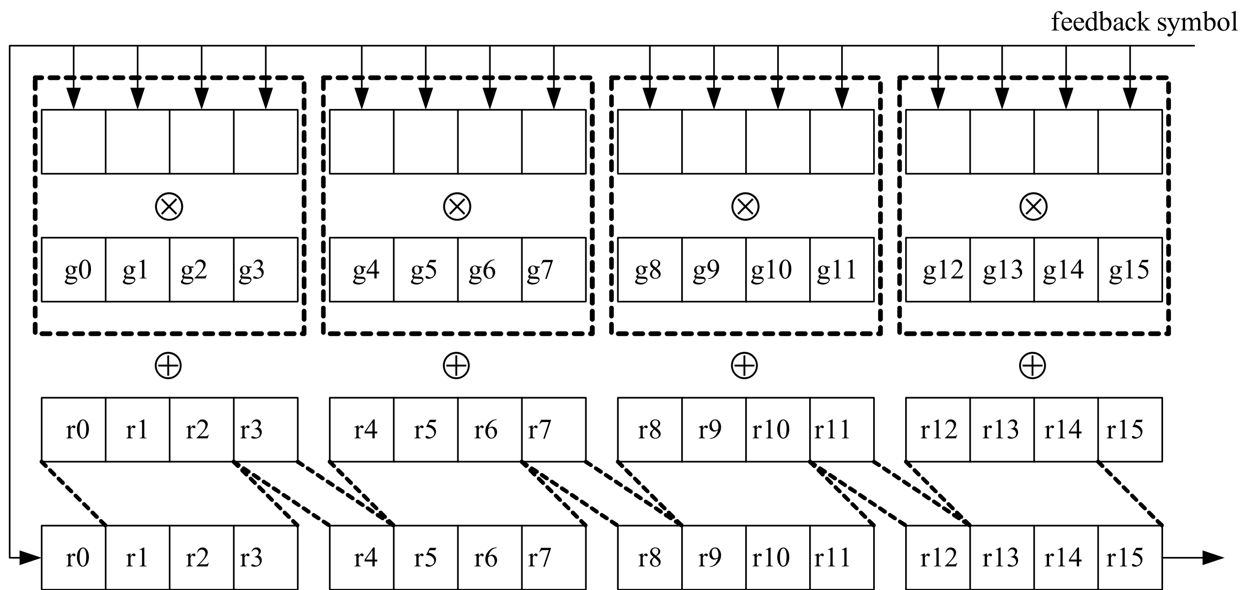


Figure 6.4: Implementation of LFSR with the Intrinsics

Thus each modulo-2 addition and Galois field multiplication in one iteration are in parallel for every four operands. Also, it leads that we can replace the original memory accesses by the register operations because the number of operands is reduced by packaging. The other benefit is that the shift of each register in LFSR is realized directly by the intrinsic instruction instead of the sequential value assignment of array elements. The intrinsics not only speed up the shift operation but also allow shifting four registers simultaneously. The effort of our improvement with the method we described previously is shown in Table 6.8.

We also accelerate the modified WB algorithm with the techniques of the intrinsics. However, the operation of this algorithm is complex and the situation is different from the re-encoding process. We also use the intrinsics to accelerate it but apply the intrinsics only to the primary operations it calls. Similar to the earlier discussions, the cause which disables the software pipeline is that a large number of memory accesses and function call are used in the modified WB algorithm. The intrinsics can be used to pack data and to reduce the number of operands to reduce memory access. We also do inline functions and use the intrinsics to build software pipelining. The functions calls in

the modified WB algorithm often contain the Galois field multiplication, the multiplication of polynomials, and the calculation of the polynomial value with a specified input, and we use the intrinsics to make the execution in them in parallel as much as we can by the similar method in re-encoding. The comparison of the modified WB algorithm improved by our method is also presented in Table 6.8.

According to Table 6.8, it is observed that the percentage of improvement is up to 98.47% for the re-encoding and 92.07% for the modified WB algorithm. They are much more efficient than the version before our improvement and even better than the syndrome computation and BM algorithm in the Lee RS decoder.

6.3 DSP Implementation of Reed-Solomon Decoder and Viterbi Decoder

for the following subsections, the DSP implementation of our RS decoder and the Viterbi decoder is divided into the system structure, the program flow, and the performance analysis. The Viterbi decoder is one module in the receiver of our IEEE 802.16a standard project. It is investigated for a long time and is considered generally very efficient for the DSP implementation. The algorithm of the Viterbi decoder is fixed for the most parts and it has been tuned by our group previously [20]. We simply use this version. The RS decoder we choose to implement is the conventional RS decoding procedure with our acceleration instead of using the remainder decoding algorithm because at the present stage, it is still less efficient.

E S A
1896

6.3.1 Structure of RS Decoder and Viterbi Decoder Implementation

The structures implemented on the DSP platform of our RS decoder and Viterbi decoder are similar to that of the AMR codec and are also illustrated by Fig. 5.1 in Chapter 5. Because the DSP platform is the same for the AMR codec, our RS decoder, and the Viterbi decoder implementation, the data communication mechanism and the code development supported by the DSP platform is also the same. Hence, we do not repeat it again, and the details can be referred to Section 5.2.1.

6.3.2 Execution Flow of RS Decoder and Viterbi Decoder

6.3.2.2 DSP Program Flow for RS Decoder

The implemented interface of the RS decoder is shown in Fig. 6.5 and similar to that of the AMR encoder. There is also a text edit box for the user to key in the coding mode supported by the RS decoder in IEEE 802.16a. The default coding mode is (60, 54, 3) RS decoder. The program flow for the host and DSP is similar to that of the AMR codec. We have to choose the path of the bitstream we want to download and click the buttons “Download” and “Transfer” for downloading and executing the bitstream. We use the block transfer mode to transfer data and coding information similar to the AMR codec implementation. The coding scheme is also capable of being changed in the middle of the RS decoding, and the blocks of rest shall be decoded with the updated coding scheme. The program flow decribed above is illustrated by Fig. 6.6. The details of the communication between the host and DSP end can be referred to Section 5.2.2.

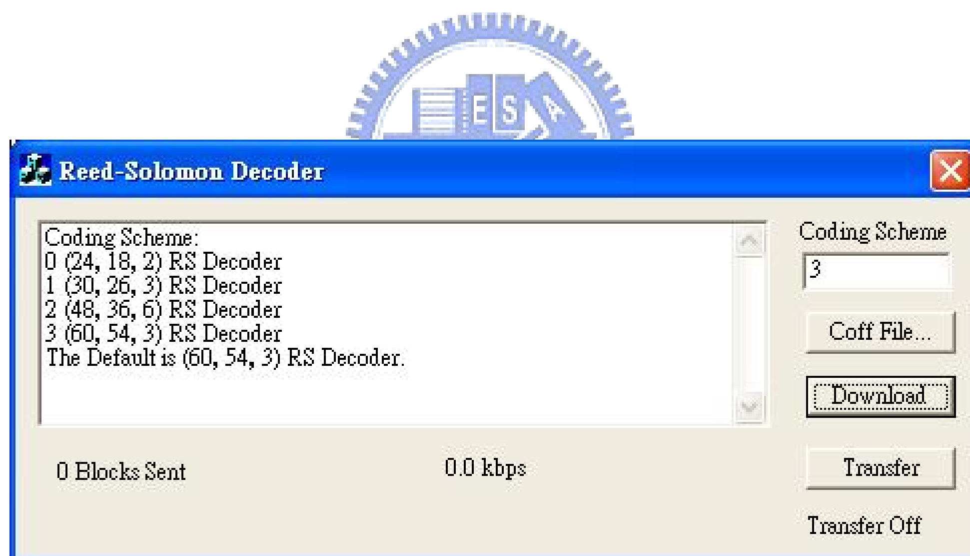


Figure 6.5: the Interface of our RS Decoder Implementation

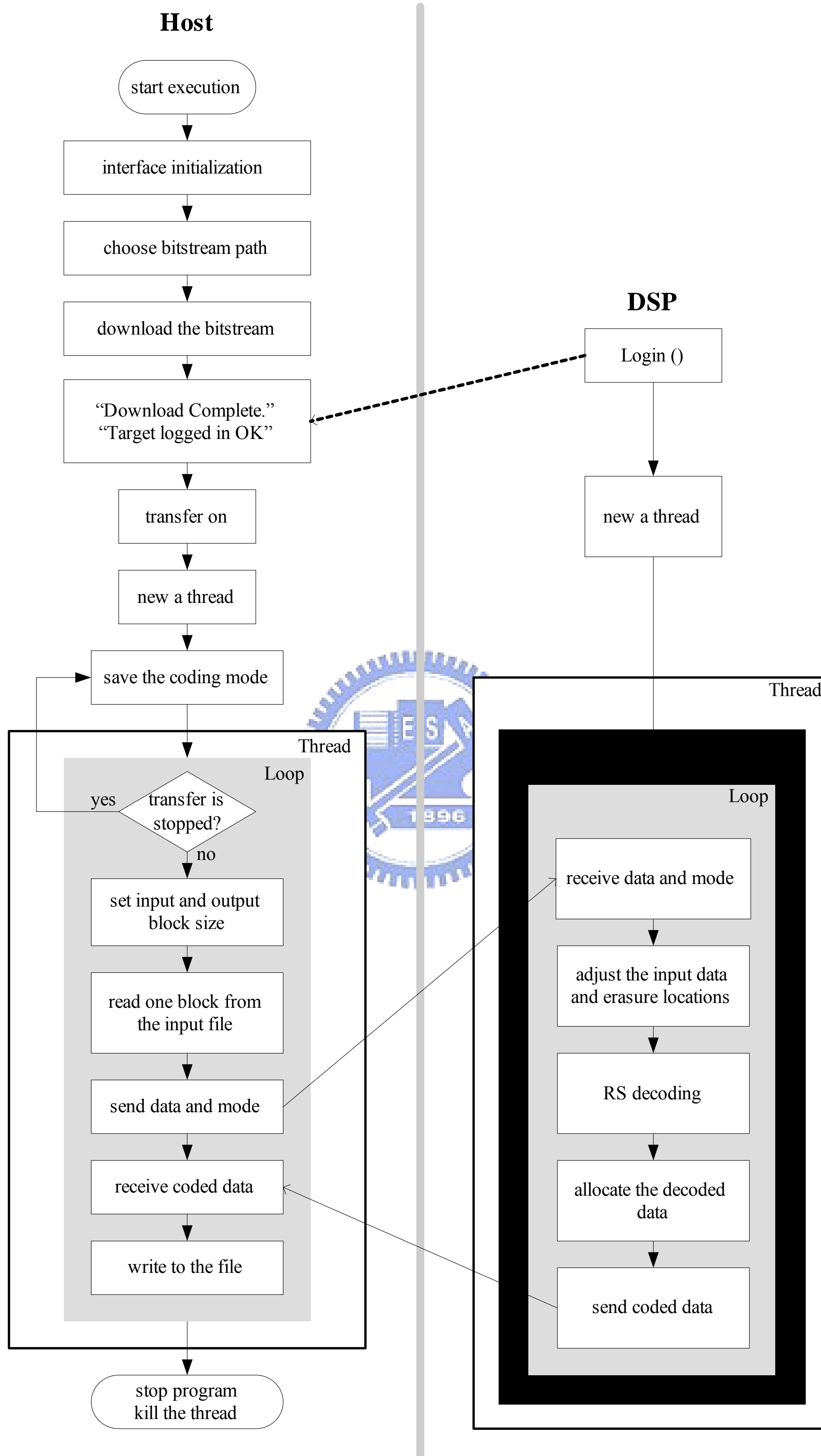


Figure 6.6: the Flowchart of our RS Decoder Implementation

6.3.2.2 DSP Program Flow for Viterbi Decoder

The interface of the Viterbi decoder implementation is shown in Fig. 6.7 and is similar to that of the RS decoder except for the text edit box, which is the coding mode. The program execution flow is also similar to that of the RS decoder, shown as Fig. 6.6, but no code mode is needed to be judged in the Viterbi decoder.

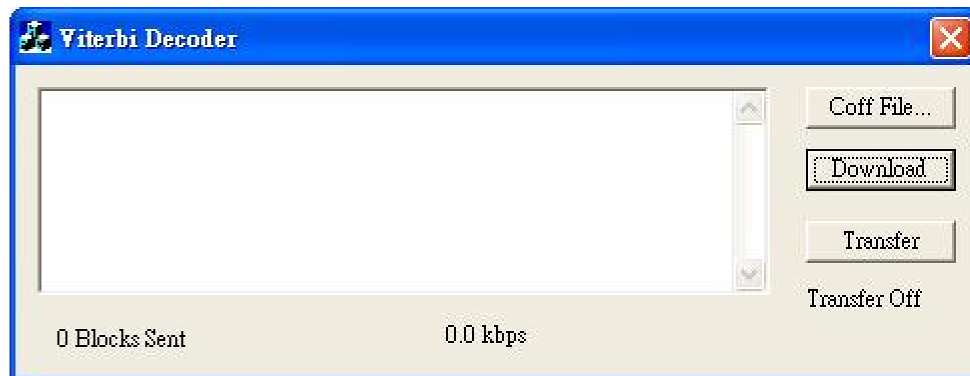


Figure 6.7: the Interface of the Viterbi Decoder Implementation

6.3.3 Performance Analysis

In this section, we present the execution time of our implementation for the RS decoder and Viterbi decoder of the IEEE 802.16a wireless communication standard. The execution time and the code size of our proposed implementation system is shown in Table 6.9.

Implemented Decoder Name	Code Size	Processing Rate (Kbytes/sec)	Improvement Percentage (%)
the Original RS Decoder	17,137,575	58.80	N/A
Improved RS Decoder	17,139,055	176.40	96.44
Viterbi	17,120,975	17.42	N/A

Table 6.9: Profile of our Implementation for RS Decoder and Viterbi Decoder

It is observed that the code sizes of the both decoder implementations are almost the same because the largest part included in the final code is the overhead of the transfer mechanism, the functions, and the constants that have been ready by the library. The improved RS decoder is up to 176.4 Kbytes/sec of the processing rate, and its improvement gain is up to 96.44% compared to the Lee RS decoder without the file-level optimization. The processing rate of the Viterbi decoder is about 17.42 Kbytes/sec. To accelerate the Viterbi decoder, it seems better to design the logic for parallelize its operation than to execute it sequentially on the DSP platform. Moreover, the algorithm of the Viterbi decoder is almost fixed, and we are only able to measure its efficiency on the DSP platform.

Chapter 7

Conclusions and Future Works

7.1 Conclusions

The speech coding approach taken by AMR is a way to adjust the speech and channel coding rate to the channel condition without losing too much quality. The Reed-Solomon codec in IEEE 802.16a provides several coding rates and error capabilities for the wireless communication. However the multiple speech coding modes and the additional channel coding for reducing channel errors increase the complexity of the implementation on the hardware. However, the technique of VLSI and architecture design advances rapidly at the present time. It gives us the opportunity to implement complicated algorithms on hardware. In this thesis, the AMR speech codec is implemented on the DSP platform, which is used mainly for multimedia coding purposes. And so is the Reed-Solomon decoder, which is used wildly because of its high capability of correcting both random and burst errors.

In the previous chapters, we first focus on the AMR speech codec. We profile the C program provided by 3GPP and find that most functions mainly consist of the function call of arithmetic operations. Hence it is an effective way to reduce much execution time by accelerating the arithmetic operations. We also use the TI DSP intrinsics, which are efficient instructions supported by the C64x DSP to take the advantage of the DSP architecture, to accelerate the AMR codec. It has been improved

up to 68.88% for the encoder and 66.12% for the decoder when the compiler-level optimization is also enabled. Finally, we implement the accelerated program on the DSP platform, and its speed is up to 14.05 ms/frame for the encoder and 2.43 ms/frame for the decoder. The measured time includes the data transfer and still meets the real time.

The other topic in this thesis is the Reed-Solomon decoder in IEEE 802.16a. The conventional decoding algorithm is described and treated as the original one for further improvement. The original decoder is first profiled. And then it is accelerated in the syndrome computation and chien search modules, which are two most time consuming procedures. We reduce their complexity and simplify their structure for the software pipeline. It is improved up to 97.79% in the syndrome computation and 73.72% in the chien search. The improved Reed-Solomon decoder is also implemented on the DSP platform. Its processing speed is up to 176.4 Kbytes/sec and is 96.44% faster than the original one. The Viterbi decoder is also implemented to complete the FED scheme in our IEEE 802.16a project. Its processing rate of DSP implementation is 17.42 Kbytes/sec. The final version of both the Reed-Solomon decoder and the Viterbi decoder in IEEE 802.16a reaches our goal of real time for the AMR speech coding.

7.2 Future Works

As discussed in the above, the processing speed of Viterbi decoder is the bottleneck in our IEEE 802.16a FED procedure. However, we have adopted the most efficient algorithm we know of and it is hard to further accelerate it by algorithm fine tuning. One way to implement and accelerate the Viterbi decoder is to design VLSI logic and parallelize its operations. So, the FED scheme may be accelerated by implementing the Viterbi decoder using the FPGA with the help of DSP. The DSP platform we use in this project contains an Xilinx FPGA. It may worth to try.

There are also other issues in the AMR codec implementation. It is not yet implemented for the analog input and output although they are included on the DSP

baseboard we use. Reading and writing files are the primary I/O for our present implementation. It would be more useful in practice to process real-time input speech or audio using the microphone and the speaker. However, we are limited by the time and not yet to test and use the I/O port. This can be another subject to explore.

Bibliography

[1] O. Corbun, M. Almgren, and K. Svanbro, "Capacity and Speech Quality aspects using Adaptive Multi-Rate (AMR)," *The Ninth IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, vol. 3, pp. 1535-1539, 1998.

[2] 3G TS 26.071: "AMR Speech Codec; General Description," 3GPP, Aug. 1999.

[3] 3G TS 26.090: "AMR Speech Codec; Speech Transcoding Functions," 3GPP, Dec. 1999.

[4] D. A. F. Florencio, "Investigating the use of Asymmetric Windows in CELP Vocoders," *ICASSP*, vol.2, pp. 427-430, 1993.

[5] R. Salami, C. Laflamme, J. P. Adoul, and D. Massaloux, "A Toll Quality 8 Kb/s Speech Codec for the Personal Communications System (PCS)," *IEEE Transactions on Vehicular Technology*, vol. 43, no. 3, pp. 808-816, Aug. 1994.

[6] R. Salami, C. Laflamme, J. P. Adoul, A. Kataoka, S. Hayashi, T. Moriya, C. Lamblin, D. Massaloux, S. Proust, P. Kroon, and Y. Shoham, "Design and Description of CS-ACELP: A Toll Quality 8 kb/s Speech Coder," *IEEE Transactions on Speech and Audio Processing*, vol. 6, no. 2, pp. 116-130, Mar. 1998.

[7] P. Kabal and R. P. Ramachandran, "The computation of line spectral frequencies using Chebyshev polynomials," *IEEE Transactions on ASSP*, vol. 34, no. 6, pp. 1419-1426, Dec. 1986.

[8] C. R. Galand, J. E. Menez, and M. M. Rosso, "Adaptive Code Excited Predictive Coding," *IEEE Transactions on Signal Processing*, vol. 40, no. 6, pp. 1317-1326, Jun. 1992.

[9] P. Kroon and B. S. Atal, "On the Use of Pitch Predictors with High Temporal Resolution," *IEEE Transactions on Signal Processing*, vol. 39, no. 3, pp. 733-735, Mar. 1991.

[10] E. Ekudden, R. Hagent, I. Johansson, and J. Svedberg, "The Adaptive Multi-Rate Speech Coder," *IEEE Proceeding of Speech Coding*, pp.117-179, 1999.

[11] A. Uvliden, S. Bruhn, and R. Hagen, "Adaptive Multi-Rate – A Speech Service Adapted to Cellular Radio Network Quality," *IEEE Tirty-Second Asilomar Conference*, vol. 1, pp. 343-347, 1998.

[12] K. Jarvinen, J. Vainio, P. Kapanen, T. Honkanen, and P. Haavisto, "GSM Enhanced Full Rate Speech Codec," *ICASSP,* vol. 2, pp. 771-774, 1997.

[13] A. M. Kondoz, *Digital Speech: Coding for Low Bit Rate Communication Systems.* Wiley, 2004.

[14] IEEE Standard for local and metropolitan area networks, Part 16, Amendment 2, 2003.

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

[16] 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.

[17] 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.

[18] F. Tosato and P. Bisaglia, "Simplified Soft-Output Demapper for Binary Interleaved COFDM with Application to HIPERLAN/2," *IEEE International Conference Communications,* vol. 2, pp. 664-668, 2002.

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

[20] Y.-T. Lee, *DSP Implementation and Optimization of the Forward Error Correction Scheme in IEEE 802.16a Standard.* M.S. thesis, National Chiao Tung University, Dep. of Elect. Eng., Hsinchu, Taiwan R.O.C., Jun. 2004.

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

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

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

[24] Innovative Integration, *Quixote User's Manual.* 2003.

[25] Innovative Integration, *Quixote Architecture.* 2003.

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

[27] 3G TS 26.074: "AMR Speech Codec Test Sequence," 3GPP, Dec. 2004.

[28] Texas Instruments, *Reed Solomon Decoder: TMS320C64x Implementation.* Literature Number: SPRA686, Dec. 2000.

[29] T.-K. Truong, J.-H. Jeng, and I. S. Reed, "Fast Algorithm for Computing the Roots of Error Locator Polynomials up to Degree 11 in Reed-Solomon Decoders," *IEEE Transactions on Communications,* vol. 49, no. 5, May 2001.

[30] M. Morii and M. Kasahara, "Generalized Key-Equation of Remainder Decoding Algorithm for Reed-Solomon Codes," *IEEE Transactions on Information Theory,* vol. 38, no. 6, Nov. 1992.

[31] X. Ma and X.-M. Wang, "On the Minimal Interpolation Problem and Decoding RS Codes," *IEEE Transactions on Information Theory,* vol. 46, no. 4 Jul. 2000.

[32] W. G. Chambers, R. E. Peile, K. Y. Tsie, and N. Zein, "Algorithm for Solving the Welch-Berlekamp Key-Equation, with a Simplified Proof", *Electronics Letters,* vol. 29, no. 18, Sep. 1993.

[33] S. R. Blackburn, "Fast Rational Interpolation Reed-Solomon Decoding, and the Linear Complexity Profiles of Sequence," *IEEE Transactions on Information Theory,* vol. 43, no. 2, Mar. 1997.

[34] A. Mahmudi, Dr. M. Benaissa, and Dr. P. Sweeney, "The Implementation of Generalized Minimum Distance Decoding for Reed Solomon Codes," *IEEE International Symposlum on Circuits and Systems,* May 2000.

[35] D. Dabiri and I. F. Blake, “Fast Parallel Algorithm for Decoding Reed-Solomon Codes,” *IEEE Transaction on Information Theory,* vol. 41, no.4, Jul. 1994.

[36] T.-K. Truong, J.-H. Jeng, and T. C. Cheng, “A New Decoding Algorithm for Correcting Both Erasures and Errors of Reed-Solomon Codes,” *IEEE Transactions on Communications,* vol. 51, no. 3, Mar. 2003.

[37] E. R. Berlekamp and L. Welch, “Error Correction for algebraic block codes,” U.S. patent 4633470, 1986.

[38] E. R. Berlekamp, *Bounded Distance+1 Soft Decision Reed-Solomon Decoding*. preprint.