

國立交通大學

電子工程學系 電子研究所碩士班

碩 士 論 文

IEEE 802.16a 分時雙工正交分頻多重進接上行傳
收系統在數位訊號處理器平台上之整合及最佳化



IEEE 802.16a OFDMA TDD Uplink Transceiver System
Integration and Optimization on DSP Platform

研 究 生：董景中

指 導 教 授：林大衛 博士

中 華 民 國 九 十 四 年 六 月

IEEE 802.16a 分時雙工正交分頻多重進接上行傳收系統在數位訊號

處理器平台上之整合及最佳化

**IEEE 802.16a OFDMA TDD Uplink Transceiver System Integration
and Optimization on DSP Platform**

研 究 生：董景中

Student: Ching Chung Tung

指 導 教 授：林大衛 博士

Advisor: Dr. David W. Lin

國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

in

Electronics Engineering

June 2005

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 四 年 六 月

IEEE 802.16a 分時雙工正交分頻多重進接上行傳 收系統在數位訊號處理器平台上之整合及最佳化

研究生：董景中

指導教授：林大衛 博士

國立交通大學

電子工程學系 電子研究所碩士班



本篇論文主要介紹 IEEE 802.16a 分時雙工正交分頻多重進接上行傳輸系統的軟體實現，我們整合前向誤差改正編碼器於傳送端，並於接收端加入同步裝置、通道等化裝置、及前向誤差改正解碼器。

我們先針對接收端的同步演算法做些修改，並在數位訊號處理 (DSP) 平台上對程式做最佳化的處理。我們的數位訊號處理平台包括一台個人電腦、Innovative Integration 公司的 Quixote 板子及其上裝置的 Texas Instrument 公司的 TMS320C6416 數位訊號處理晶片。

我們在接收端的上行同步處理機制是利用上行傳輸資訊 (preamble) 的不變性，直接對收到的信號作相關性 (correlation) 的運算。藉此找到第一個到達基地台之使用者的時間，以減低符元間的干擾 (inter symbol interference)。

為了能有效提升 DSP 運算效率，我們系統中所有的運算皆是以定點 (fixed-point) 的格式來處理。而最佳化的目標是加速程式執行的速度，以期能達

到即時運算的要求。我們提出數個針對程式所做的改善技巧，如軟體管線 (software pipelining)，或是使用 C6416 內具有的指令 (intrinsic) 來做處理。並從編譯器所提供的相關資訊做進一步的分析討論，以清楚了解程式的運作情形。

最後，傳送端的插值濾波器 (interpolator filter) 及接收端同步器的速度分別改善了 85.85 倍和 1.74 倍，且在 DSP 上執行的效率也各達到 90.94% 和 85.87%。



IEEE 802.16a OFDMA TDD Uplink Transceiver System Integration and Optimization on DSP Platform

Student: Ching Chung Tung

Advisor: Dr. David W. Lin

Department of Electronics Engineering

Institute of Electronics

National Chiao Tung University



Abstract

This thesis introduces the software implementation of the IEEE 802.16a TDD uplink transceiver system. We integrate the FEC encoder in the transmitter, the synchronizer, the channel equalizer, and the FEC decoder in the receiver.

We first do some modifications to the uplink synchronization algorithm, and then optimize our programs on the digital signal processing (DSP) platform, which includes a personal computer (PC), Innovative Integration's Quixote DSP board, and the TI's TMS320C6416 DSP chip.

The uplink synchronization mechanism is using the invariance of the preamble which is also known to the base station. We correlate it to the received signals directly, and thus find the first coming subscriber station's time to reduce the inter-symbol interference.

The data formats on this system are all "fixed-point" for improving the computational efficiency in DSP. Our optimization goal is to accelerate the program's execution speed so that it can satisfy the requirement of real-time processing. We present some optimization techniques, such as software pipelining, and using the

intrinsic of DSP, to deal with the most time-consuming parts of the program. We also discuss and analyze the compiler feedbacks to understand how the program works in the DSP.

Finally, the speed of the interpolator filter in the transmitter and the uplink synchronizer in the receiver can be improved by 85.58 and 1.74 times, respectively. The computational efficiencies of them are 90.94% and 85.87%, respectively.



誌謝

本篇論文方得以順利完成，首先想感謝林大衛老師。在兩年的研究所生涯裡，由於他的細心指導及在專業領域的博學精深，使得我在學習研究這條路上，一直都能順利地往前行。祝福老師在忙碌之餘，能保有健康的身體。

另外，感謝通訊電子與訊號處理實驗室所有的成員，包含各位師長、同學、學長姐與學弟妹們。感謝吳俊榮學長、洪崑健學長給予我在研究過程上的指導與建議，還有簡志凱同學、陳昱昇同學、陳汝芬同學、王盈閔同學、陳志楹同學、徐漢光等同學，因為能和你們共同討論及分享求學的經驗，使得實驗室一直是一個燈光美、氣氛佳的好地方。

最後，我要感謝我的家人和朋友們，感謝他們一直都在背後支持著我，讓我能心無旁騖地完成學業。

在此，將此篇論文獻給所有給予我幫助的人。



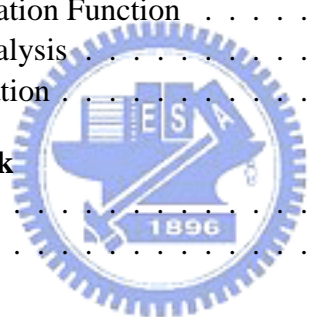
董景中

民國九十四年六月 於新竹

Table of Contents

Table of Contents	vi
List of Tables	viii
List of Figures	ix
1 Introduction	1
2 The IEEE 802.16a TDD OFDMA Uplink Transmission Scheme	5
2.1 Introduction to OFDM	5
2.2 Overview of OFDMA	8
2.3 Overview of the IEEE 802.16a Standard	9
2.3.1 UL Carrier Allocation	11
2.3.2 OFDMA Data Mapping	12
2.3.3 OFDMA Frame Structure for TDD	13
2.4 Transmitter - Receiver System Architecture	14
2.4.1 Modulation	15
2.4.2 TX/RX SRRC filter	18
2.5 UL Synchronization Problems	18
2.6 UL Synchronization	20
2.7 UL Synchronization Result	22
2.7.1 Simulation Parameters and Environments	22
2.7.2 UL Synchronization	24
3 Introduction to the DSP Implementation Platform	27
3.1 DSP Board [16]	27
3.2 DSP Chip [18]	28
3.3 Data Transmission Mechanism [16]	34
3.3.1 DSP Streaming Interface	34
3.3.2 CPU Busmastering Interface	36
3.3.3 Packetized Message Interface	38

4	Integration and Optimization of the IEEE 802.16a OFDMA TDD Uplink Transmitter-Receiver System	41
4.1	Structure of the Implemented System	41
4.1.1	CPU Busmastering Interface	43
4.2	Fixed-Point Data Formats	45
4.3	TI's Code Development Environment [21]	48
4.3.1	Code Development Flow [19]	49
4.3.2	Compiler Optimization Options [19]	51
4.3.3	Software Pipelining [22]	52
4.3.4	Intrinsics [19]	54
4.4	Performance of the Original Program	54
4.5	The Modulation Function	55
4.6	The Framing and Deframing Functions	60
4.7	The IFFT and FFT Functions	65
4.7.1	Analysis of the Output Performance	66
4.7.2	Complexity Analysis	68
4.8	Transmission Filtering	71
4.8.1	Complexity Analysis	75
4.9	The Uplink Synchronization Function	76
4.9.1	Complexity Analysis	79
4.10	Conclusion in Optimization	82
5	Conclusion and Future work	85
5.1	Conclusion	85
5.2	Potential Future work	86
	Bibliography	87



List of Tables

1.1	Comparison of OFDMA Uplink Carrier Allocations in IEEE 802.16-2004 and IEEE 802.16a	2
2.1	OFDM Advantages and Disadvantages	7
2.2	OFDMA UL Carrier Allocation	13
2.3	System Parameters Used in Our Study [14]	22
2.4	ETSI “Vehicular A” Channel Model in Different Units [23]	24
2.5	Relation Between Speed and Maximum Doppler Shift at Carrier Frequency 6 GHz. Subcarrier Spacing is 5.58 kHz	25
3.1	Functional Units (.L, .S) and Operations Performed [18]	32
3.2	Functional Units (.M, .D) and Operations Performed [18]	33
3.3	CIIMessage Header Field [16]	39
3.4	CIIMessage Data Section Interface [16]	39
4.1	Q1.14 Bit Fields	46
4.2	Range of Data Values After Modulation	47
4.3	Profile of Transmitter Function Blocks	55
4.4	Profile of Receiver Function Blocks	55
4.5	Breakdown of Clock Cycles for Three Modulation Functions	59
4.6	Breakdown of Clock Cycles for framing()	64
4.7	Breakdown of Clock Cycles for deframing()	65
4.8	Computational Complexity for FFT algorithm	68
4.9	Complexity and Efficiency of DSP_fft16x16r and DSP_fft32x32	70
4.10	Breakdown of Clock Cycles for IFFT()	71
4.11	Breakdown of Clock Cycles for TX_SRRC()	75
4.12	Breakdown of Clock Cycles for Modified Code using DSP_fir_gen()	75
4.13	Breakdown of Clock Cycles for TX_SRRC()	76
4.14	Complexity and Efficiency of SRRC Filter	76
4.15	Breakdown of Clock Cycles for sync()	79
4.16	Complexity and Efficiency of sync()	82
4.17	Profile of 802.16a UL Transmitter Function Blocks	83
4.18	Profile of 802.16a UL Receiver Function Blocks	83

List of Figures

1.1	Frame structures of IEEE 802.16 2004 (top) [5] and IEEE 802.16a (bottom) [1].	4
2.1	Bandwidth efficiency comparison of traditional FDM and OFDM systems.	6
2.2	The use of cyclic prefix.	7
2.3	Carrier allocation of an OFDMA symbol.	9
2.4	Carrier allocation of an OFDMA symbol (modified from [9]).	10
2.5	Illustration of carrier usage in OFDMA UL.	12
2.6	Mapping of FEC blocks to OFDMA subchannels and symbols (from [1]).	14
2.7	Time plan of one OFDMA frame (from [1]).	15
2.8	UL transmitter structure.	16
2.9	UL receiver structure.	16
2.10	QPSK, 16-QAM, and 64-QAM constellations [1].	16
2.11	PRBS for generation of data pilots and preamble pilots [1].	17
2.12	Positioning of the FFT window.	19
2.13	Three UL signals arrive at different times, and the CP correlation peak may occur between them [11].	20
2.14	Illustration of UL synchronization in time domain.	21
2.15	The received samples and the time plan of the UL synchronization.	22
2.16	Frame structure used in UL synchronization.	23
2.17	The transition instant for BS to turn around.	24
2.18	Error distribution under different maximum Doppler shifts.	25
2.19	Power-delay profile of the multipath channel [14].	26
2.20	Performance of UL symbol time synchronization: error distribution under different maximum Doppler shifts.	26
3.1	Quixote-II board [24].	28
3.2	Block diagram of Quixote-II(from [16]).	29
3.3	Functional block and CPU (DSP core) diagram [17].	30
3.4	The C64x CPU block diagram [18].	31
3.5	DSP streaming mode [16].	35
3.6	Simple target to host messaging configuration [16].	39
4.1	Structure of implemented system.	42
4.2	System structure on transmitter side (modified from [15]).	43
4.3	System structure on receiver side (modified from [15]).	43

4.4	Organization of transmitter and receiver using CPU busmastering interface.	45
4.5	Fixed-point data formats at the transmitter side.	45
4.6	The fixed-point data formats at the receiver side.	47
4.7	Code development flow of C6000 (from [19]).	50
4.8	Software pipeline loop (from [18]).	53
4.9	Compiler's feedback of the modulation().	55
4.10	A part of C code in the main function.	56
4.11	C code for modulation_16QAM().	56
4.12	A part of the assembly code in the modulation_16QAM().	57
4.13	Compiler's feedback of the modulation_QPSK().	58
4.14	Compiler's feedback of the modulation_16QAM().	58
4.15	Compiler's feedback of the modulation_64QAM().	59
4.16	C code for original PRBS generator (from [14]).	60
4.17	C code for generating the carrier locations.	60
4.18	Two versions of C program for framing of the preamble.	61
4.19	Two versions of C program for framing of other symbols.	61
4.20	The resulting assembly code for the revised code.	62
4.21	Compiler feedback of the optimized code for other symbols.	63
4.22	Block diagram of the IFFT function.	67
4.23	Performance of IFFT when the modulation is 16-QAM.	69
4.24	Performance of IFFT when the modulation is 64-QAM.	69
4.25	A part of the assembly code in DSP_fft16x16r.	70
4.26	Implementation of interpolation filter with polyphase decomposition [11].	72
4.27	Convolution kernel at the boundary of a finite-length sequence.	72
4.28	C code for convolution with $E_0(z)$ and $E_2(z)$	73
4.29	Compiler's feedback for convolution with $E_0(z)$ and $E_2(z)$	73
4.30	A part of assembly code for convolution with $E_0(z)$ and $E_2(z)$	74
4.31	C code in sync() before optimization.	77
4.32	C code in sync() after optimization.	78
4.33	Graphical illustration of $c = \text{dotp2}(b,a)$ [19].	78
4.34	Compiler's feedback of the code shown in Fig. 4.31.	80
4.35	Compiler's feedback of the code shown in Fig. 4.32.	80
4.36	A part of the assembly code in sync().	81

Chapter 1

Introduction

Orthogonal frequency division multiple access (OFDMA) is a variation scheme of orthogonal frequency division multiplexing (OFDM), which is a special case of multicarrier transmission that transmits one data stream over a number of subchannels. What makes OFDMA different from OFDM is that multiple users can share one OFDM symbol. It is the combination of OFDM and frequency division multiple access (FDMA), but the guard band of each user could be neglected. OFDMA provides a highly flexible and efficient structure for multiuser communication. At present, OFDMA has been proposed for use in wireless broadband multimedia communications systems (WBMCS) in IEEE 802.16a [1] and in cable TV networks [2].

The IEEE 802.16a standard is an extension of the global IEEE 802.16 WirelessMAN standard for 10 to 66 GHz published in April 2002. It provides for fixed broadband wireless access (BWA) between 2 and 11 GHz for non-line-of-sight connections up to 31 miles at speeds up to 70 Mbps.

The IEEE 802.16a, “Air Interface for Fixed Broadband Wireless Access Systems — Medium Access Control Modifications and Additional Physical Layer Specifications for 2–11 GHz,” sets the platform for the extensive deployment of 2 to 11 GHz wireless metropolitan area networks (MANs) as an economical alternative to wireline “first-mile” connections to public networks. “It closes the first-mile gap, giving users an easily installable, wire-free method to access core networks for multimedia applications,” states

Table 1.1: Comparison of OFDMA Uplink Carrier Allocations in IEEE 802.16-2004 and IEEE 802.16a

Parameters	IEEE802.16 2004	IEEE 802.16a
Number of dc subcarriers	1	1
$N_{subchannels}$	70	32
N_{used}	1681	1696
Number of data carriers per subchannel	48	48
Guard subcarriers: Left, Right	184, 183	176,175

Roger Marks, Chair of the 802.16 Working Group on Broadband Wireless Access [3].

The new 802.16d upgrade to the 802.16a standard was recently approved in June 2004 (now named 802.16-2004), and primarily introduces some performance enhancement features in the uplink [4]. It consolidates IEEE Std 802.16, IEEE Std 802.16a, and IEEE Std 802.16c, retaining all modes and major features without adding modes [5]. Table 1.1 gives a comparison between IEEE 802.16-2004 and IEEE 802.16a in OFDMA uplink carrier allocations. The number of subchannels is increased to 70, while in IEEE 802.16a it is 32. The TDD frame structure has also been modified in IEEE 802.16-2004, which is shown in Figure 1.1. We can see that in IEEE 802.16-2004, each frame begins with a preamble followed by a downlink transmission period and an uplink transmission period. This is quite different from the frame structure in IEEE 802.16a, where a preamble appears only in the uplink subframe.

Since the project that this thesis is based was started in 2002, the algorithms implemented in this work have been designed to meet the requirements of IEEE 802.16a. In this thesis, we will discuss the DSP implementation and optimization of the IEEE 802.16a uplink system, instead of IEEE 802.16-2004.

Generally speaking, the basic elements in a communication system can be divided into two parts, a transmitter and a receiver. In our case, we will implement the uplink transmitter-receiver pair that includes the transmitter in the subscriber station (SS) and the receiver in the base station (BS). Our work is mainly based on the simulation program

in [14], adding the FEC (forward-error-correcting coding) encoder and decoder of [15].

We briefly introduce the references [14] and [15]. In [14], the intent is to introduce the uplink synchronization scheme by using digital signal processor. The work also includes the implementation of the framing/deframing structure, IFFT/FFT block and TX/RX SRRC filter. In [15], it focuses on the implementation of FEC/FED (forward-error-correcting decoding) scheme of the IEEE 802.16a on II Quixote DSP board.

The hardware environment of our system involves one host personal computer (PC) and a digital signal processing (DSP) chip housed on Innovative Integration's Quixote PC plug-in card. The DSP core, Texas Instruments, TMS320C6416, "meets the need for today's high processing speed for digital data transmission" [18]. It uses an advanced very long instruction word (VLIW) architecture, VelociTI, which can allow the functional units to work in parallel so that the execution time can be greatly reduced.

In our system, we will make use of the busmastering interface [16] provided by Quixote as a method for host-to-target communication. A major issue dealt with in this work, besides system integration, is the efficiency in DSP software implementation employing various optimization techniques. Our optimization is aim to accelerate the execution speed of the programs.

This thesis is organized as follows. In chapter 2, we introduce the IEEE 802.16a TDD OFDMA uplink transmission scheme. Chapter 3 introduces the DSP platform. Chapter 4 discusses the DSP optimization methods and presents the optimization results. Finally, in chapter 5 we give a conclusion and point out some potential future work.

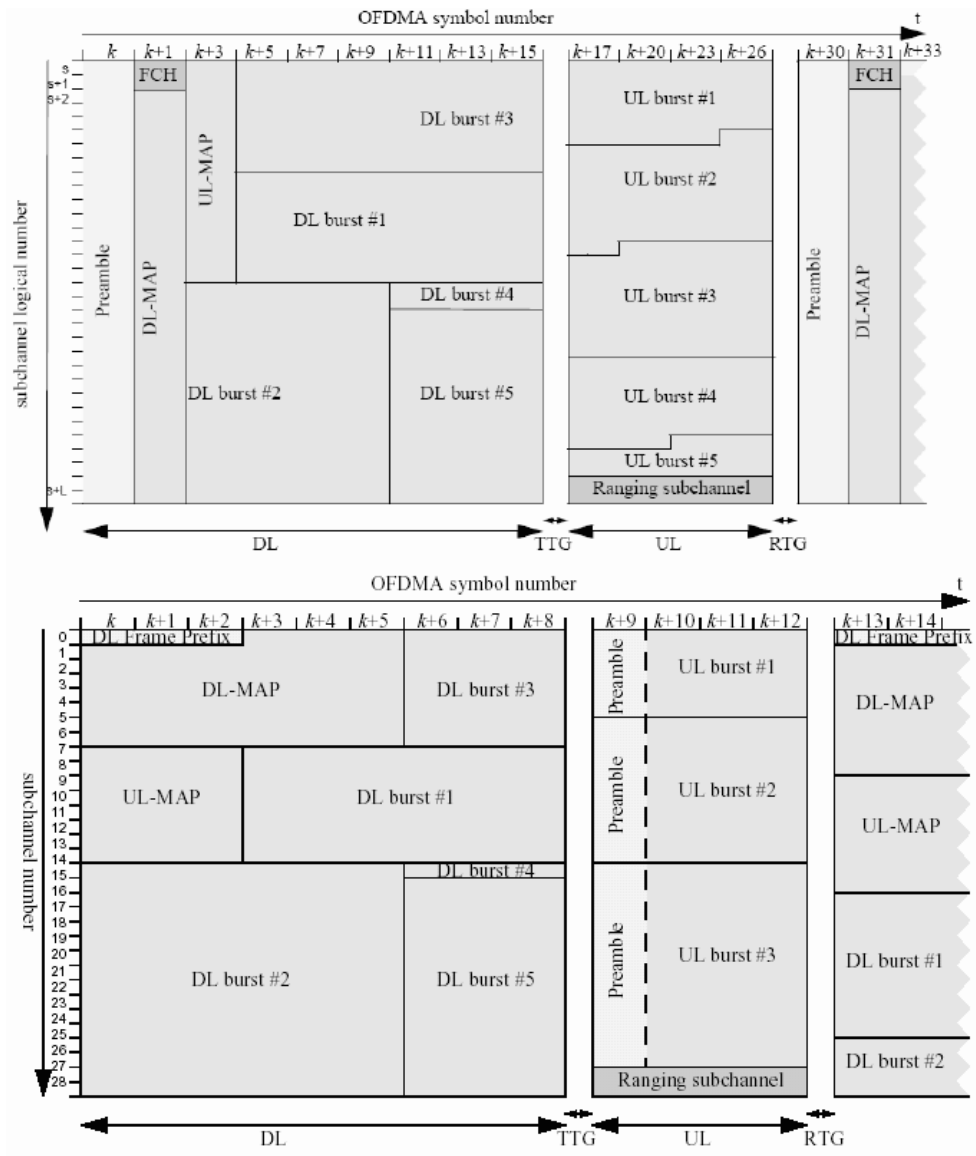



Fig. 1.1: Frame structures of IEEE 802.16 2004 (top) [5] and IEEE 802.16a (bottom) [1].

Chapter 2

The IEEE 802.16a TDD OFDMA Uplink Transmission Scheme

In this chapter, we first introduce some basic concepts regarding OFDM and OFDMA. Then, we give a brief overview of the relevant specification in IEEE 802.16a and describe our transmission and reception schemes in detail.

2.1 Introduction to OFDM



Orthogonal frequency division multiplexing (OFDM) is a special case of multicarrier transmission technique, where a single datastream is transmitted over a number of lower rate subcarriers [6]. It has been successfully applied in many digital communication systems in recent years. The concept of OFDM is to use parallel data transmission and frequency multiplexing. It divides the available spectrum into several narrow subcarrier bands, and each subcarrier only transmits part of the information.

We would like to emphasize that the orthogonality of OFDM constitutes one major difference from the classical parallel data system, making its use of the available spectrum more efficient. Figure 2.1 shows the differences. As we can see, the subcarriers in an OFDM symbol can be arranged so that the sideband of each subcarrier overlaps but the received symbols still live without adjacent interference. This can be accomplished by using the discrete Fourier transform (DFT) proposed by Weinstein and Ebert in 1971 [7].

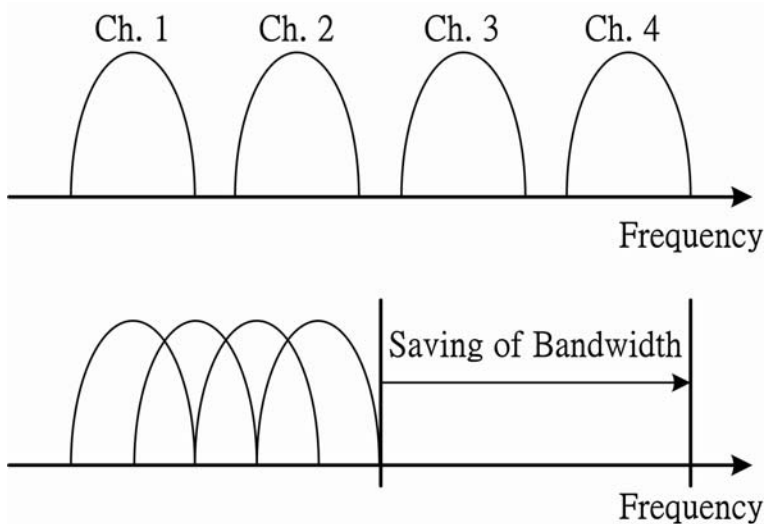


Fig. 2.1: Bandwidth efficiency comparison of traditional FDM and OFDM systems.

The complexity of DFT, however, is too expensive. Fortunately, modern advances in very-large-scale integration (VLSI) make it possible to use the fast fourier transform (FFT) for a more efficient implementation of the DFT. The complexity is reduced from N^2 in DFT to $N \log_2 N$ in FFT.

One of the most important reasons to do OFDM is that it can deal with multipath delay spread in a more efficient way. This is achieved by introducing a guard time for every OFDM symbol such that intersymbol interference can be eliminated. The guard time is chosen larger than the expected delay spread, such that multipath components from one symbol cannot interfere with the next symbol. However, if the guard time is filled with zeros, the orthogonality among subcarriers will no longer exist, and this causes serious intercarrier interference (ICI).

To preserve the orthogonality among subcarriers and eliminate ICI, the OFDM symbol should be cyclically extended in the guard time rather than just extended with zero. Hence the guard time is usually called cyclic prefix (CP). Figure 2.2 shows how to add cyclic prefix in front of an OFDM symbol.

Hence if the maximum multipath delay is smaller than the guard time, we can ensure

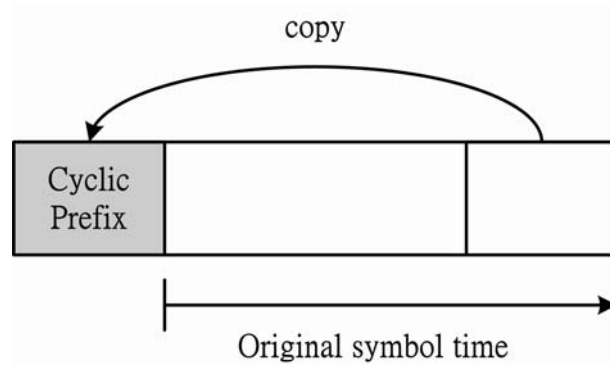


Fig. 2.2: The use of cyclic prefix.

Table 2.1: OFDM Advantages and Disadvantages

OFDM Advantages	OFDM Disadvantages
Bandwidth efficiency	Peak power problem
Resistant to multipath effect	SNR loss
Efficient implementation	Sensitive to frequency offset and phase noise

that the delayed replicas of the OFDM symbols will still have an integer number of cycles within the FFT intervals. After all, any multipath signals that have delay spread smaller than the guard time will not cause ICI or ISI.

The advantages and disadvantages of OFDM are summarized in Table 2.1. Bandwidth efficiency is already shown in Figure 2.1. The bandwidth is saved by almost 50%. Resistance to multipath effect is already discussed above. Efficiency of implementation means that OFDM can be realized by using the FFT and IFFT instead of lots of sinusoidal generators and coherent demodulators required in a parallel system.

The disadvantages of OFDM are high peak-to-average power ratio (PAPR), loss of signal-to-noise ratio (SNR), and sensitivity to frequency offset and phase noise. The high peak-to-average power ratio in OFDM signals can increase the complexity of the analog-to-digital and digital-to-analog converters and reduce the efficiency of the RF power amplifier. SNR loss is due to the insertion of the guard time, reducing the efficiency in bandwidth and power. Because of the overlapping of the subcarriers, OFDM is very sensitive

to frequency offset and phase noise.

2.2 Overview of OFDMA

For network applications where a base station communicates with multiple subscribers, the system resources must be partitioned among the subscribers to provide “multiple access” services. The sharing of spectrum is required to achieve high capacity by simultaneously allocating the available bandwidth to multiple users. For high quality communication, this must be done without severe degradation in the performance of the system. At present, it is often realized by using the techniques of time division multiple access (TDMA), frequency division multiple access (FDMA), or spread spectrum multiple access (SSMA) which is also referred to as code division multiple access (CDMA).

OFDMA, also referred to as multi-user OFDM, is now considered one of the most promising multiple access methods for fourth generation wireless networks. OFDM or OFDMA is currently the modulation of choice for high speed data access systems such as IEEE 802.11a/g wireless LAN (WiFi) and IEEE 802.16a/d/e wireless broadband access systems (perhaps more widely known as WiMAX).

In present OFDM systems, multiple access can be supported by employing time division or frequency division, and only one user is allowed to transmit data on all of the subcarriers. This scheme does not realize the fact that different users see different wireless channels. OFDMA, however, allows multiple users to transmit on different subcarriers of an OFDM symbol concurrently. Figure 2.3 shows an example with four users in it. In this figure, we illustrate that the subcarriers can carry information of different users. Because of the very low probability that all users experience a deep fade in the same subcarriers, it is possible to assure that subcarriers are assigned to the users who see good channel gains on them.

Fig. 2.4 shows an example carrier allocation of an OFDMA symbol. The frequency response of a typical broadband wireless channel is also depicted. In this example, the

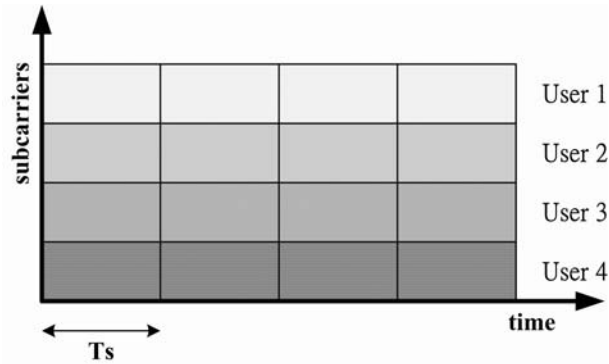


Fig. 2.3: Carrier allocation of an OFDMA symbol.

deep-fading condition and narrowband interference are also considered. In the top plot, we see that when the channel is in deep fade, the subcarriers are not sufficiently energy efficient to carry information. These wasted subcarriers can be utilized in OFDMA, thus achieving higher efficiency and capacity. Very few, if any, subcarriers are wasted in OFDMA, since no particular subcarrier is likely to be bad for all users.

2.3 Overview of the IEEE 802.16a Standard

For years, there exists a continuing challenge for service providers to satisfy the growing demand for broadband wireless access (BWA) in underserved business and residential markets [8]. They are seeking a solution to build systems that support infrastructure build outs comparable to cable, digital subscriber lines (DSL), and fiber. Recently, the IEEE 802.11x or WiFi wireless LAN technology has been used in BWA applications; however, it was evident that they are not suitable for outdoor BWA use for their limited capacity in terms of bandwidth and subscribers, range and other issues [8].

The IEEE conducted a multi-year effort to develop this standard, culminating in final approval of the 802.16a air-interface specification in January 2003. The 802.16a standard delivers carrier-class performance in terms of robustness and QoS and has been designed

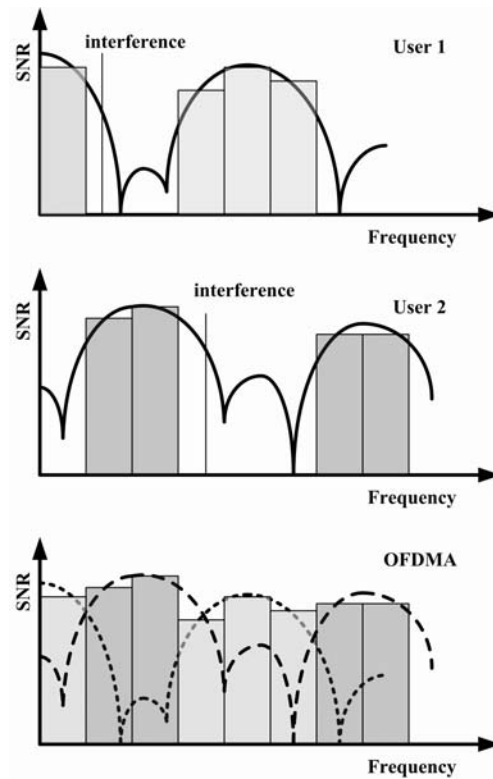


Fig. 2.4: Carrier allocation of an OFDMA symbol (modified from [9]).

from the ground up to deliver a suite of services over a scalable, long range, high capacity “last mile” wireless communications for carriers and service providers around the world [8]. The 802.16a standard specifies a protocol that among other things supports low latency applications such as voice and video, provides broadband connectivity without requiring a direct line of sight between subscriber terminals and the base station and will support hundreds if not thousands of subscribers from a single BS [8].

The IEEE 802.16a is an amendment of the 802.16 standard to cover frequency bands in the range between 2 and 11 GHz, and it specifies a metropolitan area networking protocol that enables a wireless alternative for cable, DSL and T1 level services for last mile broadband access [8]. The major reason for using 2–11 GHz bands is that they have the ability to deal with non-line-of-sight (NLOS) operation. The longer wavelengths allow

for non-directional NLOS operation with the ability to serve much broader geographic regions, allowing underserved customers to take advantage of this technology. Compared to the higher frequencies, such spectra offer the opportunity to reach many more customers less expensively, although at generally lower data rates [10].

The 2–11 GHz spectrum does not require line-of-sight and directionality, and therefore requires multiplexing techniques supporting multi-path propagation. Because residential applications are expected, rooftops may be too low for a clear sight line to a BS antenna. Therefore, significant multipath propagation is expected [10]. As a result, the 802.16a did some major changes to the PHY layer specification, which includes a single carrier PHY, a 256-point FFT OFDM PHY, and a 2048-point FFT OFDMA PHY, to address the needs of 2–11 GHz bands. In this thesis, we consider the 2048-point FFT OFDMA.

The glossary we will often use in the following sections is introduced here. The direction of transmission from the base station (BS) to the subscriber station (SS) is called downlink (DL), and the opposite direction is uplink (UL). The SS is usually known as the mobile station or the user. The BS is a generalized equipment set providing connectivity, management, and control of the SS.

2.3.1 UL Carrier Allocation

The number of subcarriers in one OFDMA symbol is 2048. These carriers are divided into as three types: data carriers for data transmission, pilot carriers for various estimation purposes, and null carriers (guard bands and DC carrier) which transmit nothing at all. The data and pilot carriers together are termed the used carriers for they transmit useful information. The allocation is as shown in Fig. 2.5 for UL. Among the 2048 subcarriers, there are 1696 used carriers, composed of 1536 data carriers and 160 pilot carriers. The rest 352 subcarriers are unused subcarriers as the guard band distributed on the edges of the symbol, and one DC carrier right in the middle of the band of the OFDMA symbol.

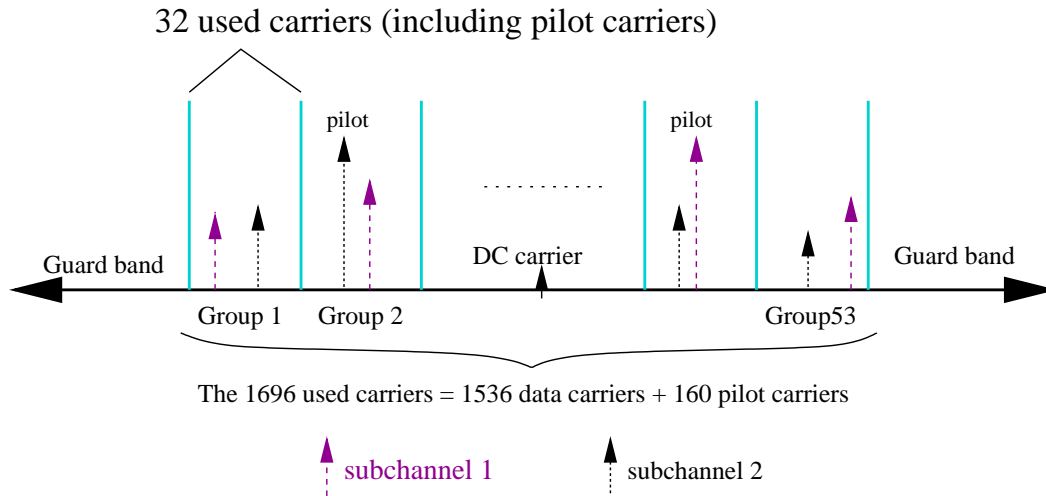


Fig. 2.5: Illustration of carrier usage in OFDMA UL.

In 802.16a, the used subcarriers are divided into 32 subchannels, where each subchannel contains 48 data carriers, 1 fixed pilot carrier, and 4 variable location pilot carriers. The carrier allocation for UL is listed in Table 2.2.

The carrier index of the fixed-location pilots never change in different symbols. The variable-location pilots, however, shift their locations every symbol periodically every 13 symbols, according to $L_k = 0, 2, 4, 6, 8, 10, 12, 1, 3, 5, 7, 9, 11$, where $k = 0$ to 12. L_k is the amount of carrier spacing which will be added to L_0 to shift to the right of the subcarrier position. For $k = 0$, the variable-location pilots are positioned at indexes 0, 13, 27, and 40. For other values of k , these locations change by adding L_k to each index.

2.3.2 OFDMA Data Mapping

A PHY burst in OFDMA is allocated a group of contiguous subchannels, in a group of contiguous OFDMA symbols using an FEC block as a unit. Note that one FEC block spans one OFDMA subchannel in the subchannel axis and three OFDM symbols in the time axis. Fig. 2.6 illustrates the order in which FEC blocks are mapped to OFDMA subchannels and OFDM symbols [1].

Table 2.2: OFDMA UL Carrier Allocation

Parameter	UL Value
Number of DC carriers	1
Number of guard carriers, left	176
Number of guard carriers, right	175
N_{used} , number of used carriers	1696
Total number of carriers	2048
$N_{varLocPilots}$	128
Number of fixed-location pilots	32
Number of variable-location pilots which coincide with fixed-location pilots	0
Total number of pilots	160
Number of data carriers	1536
$N_{subchannels}$	32
$N_{subcarriers}$ per subchannel	53
Number of data carriers per subchannel	48

2.3.3 OFDMA Frame Structure for TDD

The 802.16a is designed to operate in the frequency band between 2 to 11 GHz. The duplexing method of OFDMA system in this band shall be either frequency division duplexing (FDD) or time division duplexing (TDD) in licensed bands and TDD in license-exempt bands. We consider the TDD mode in this thesis, since TDD is better suited to data communications, which is often highly asymmetric. TDD flexibility permits efficient allocation of the available traffic transport capacity, and thus the uplink and downlink traffic transport ratio can vary with time.

Fig. 2.7 shows the frame structure of TDD OFDMA. A frame consists of one DL subframe and one UL subframe, and they are transmitted by the BS and the SS, respectively. The allowed duration of a frame is from 2 to 20 ms and is specified by the frame duration code. A subframe contains several transmission bursts, which are composed of multiples FEC blocks. In each frame, the Tx/Rx transition gap (TTG) and Rx/Tx transition gap (RTG) shall be inserted between the downlink and uplink and at the end of each frame respectively to allow the BS and the SS to turn around. TTG and RTG shall be at least

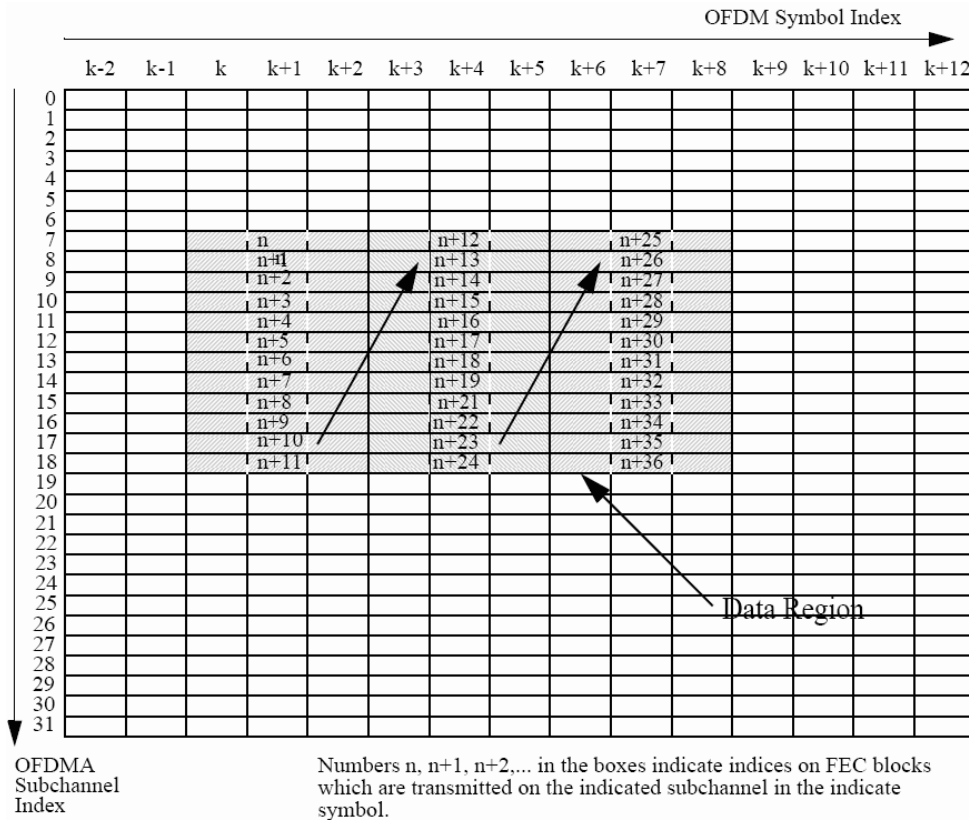


Fig. 2.6: Mapping of FEC blocks to OFDMA subchannels and symbols (from [1]).

5 μ sec and an integer multiple of four samples in duration [1].

From the UL-MAPs, the SSs know their usable subchannels and transmission time. The first symbol is an all-pilot preamble where the SS should send on all its allocated subchannels. The number of symbols of the UL subframe is $3N + 1$, where N is a positive integer, one for the preamble, and the others for data bursts.

2.4 Transmitter - Receiver System Architecture

The UL transmitter is shown in Fig. 2.8. For each SS, the transmitted data are first scrambled, FEC encoded, and then interleaved. After passing through the constellation mapper, the data are mapped to Gray-mapped QPSK, 16-QAM, or 64-QAM up to the option of the modulation types. The framing is used to arrange the coded data, MAPs, preamble

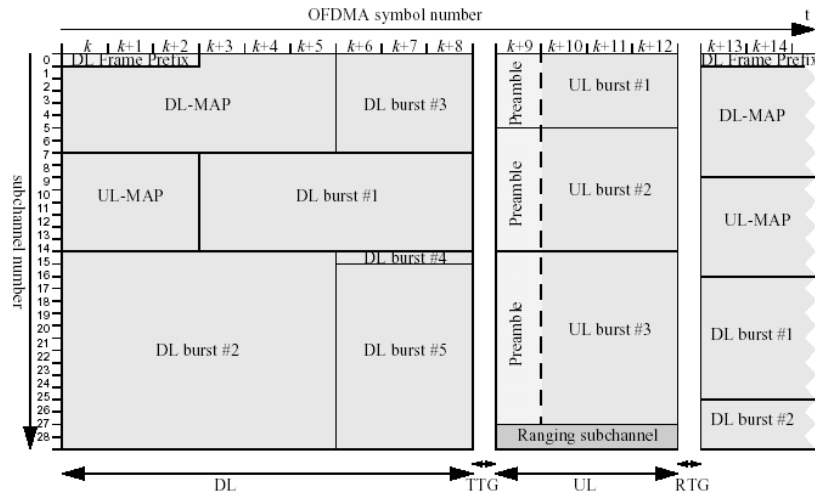


Fig. 2.7: Time plan of one OFDMA frame (from [1]).

and pilots to the corresponding subchannels following the specification of used carrier allocation. After framing, the used carriers and null carriers are allocated properly and fed into the 2048-point IFFT block in parallel. The IFFT results are output sequentially and shaped by the interpolator block, which is composed of a $4\times$ upsampler and a low-pass filter (LPF).

The receiver is shown in Fig. 2.9. It is in some sense a modified reverse of the transmitter. Synchronizer and channel estimator are added. In the following subsections, we introduce the modulation and the TX/RX SRRC filter.

2.4.1 Modulation

Data Modulation

Gray-mapped QPSK and 16-QAM must be supported by any compliant transceiver, whereas the support of 64-QAM is optional. The constellations as shown in Fig. 2.10 shall be normalized by multiplying the constellation points with the indicated factor c (shown in Fig. 2.10) to achieve equal average power. The constellation-mapped data shall be subsequently modulated onto the allocated data carriers.

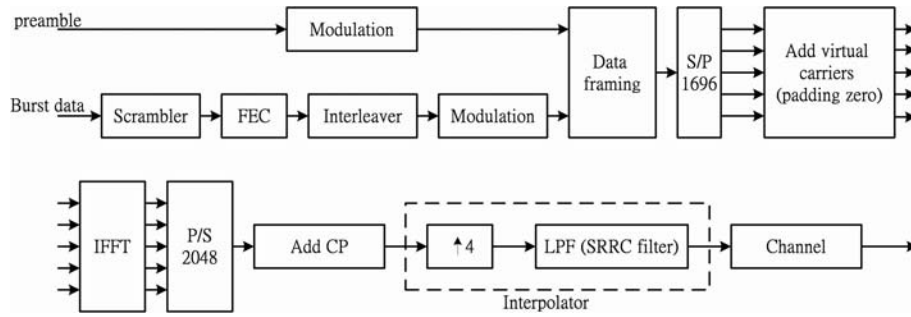


Fig. 2.8: UL transmitter structure.

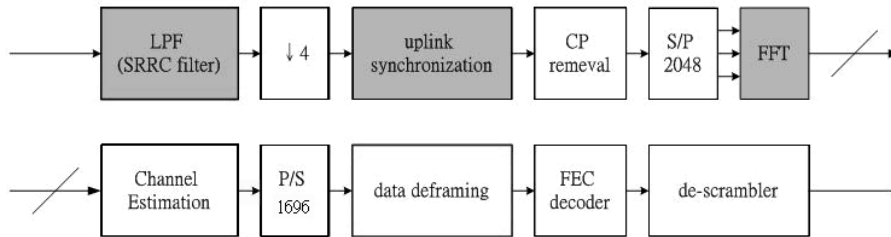


Fig. 2.9: UL receiver structure.

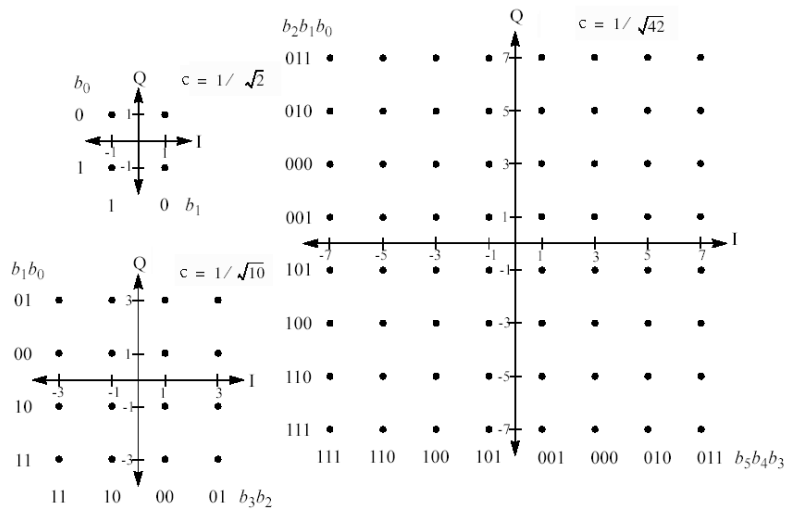


Fig. 2.10: QPSK, 16-QAM, and 64-QAM constellations [1].

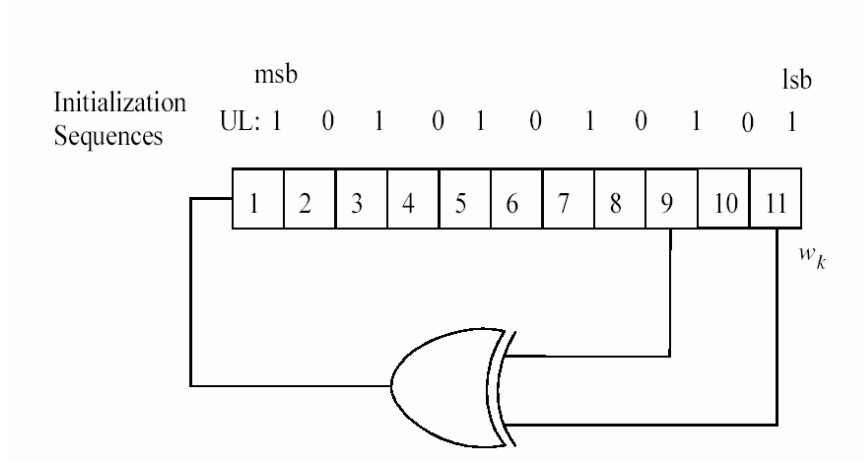


Fig. 2.11: PRBS for generation of data pilots and preamble pilots [1].

Pilot Modulation

There are two types of pilot to be modulated: data pilots and preamble pilots. These two pilots are generated using the PRBS generator in Fig. 2.11 with initialization vector [1 0 1 0 1 0 1 0 1 0 1] for the UL.

1. Data Pilot Modulation

Each pilot shall be transmitted with a boosting of 2.5 dB over the average power of each data tone. The pilot carriers shall be modulated according to the following formulas:

$$\Re\{c_k\} = \frac{8}{3}\left(\frac{1}{2} - w_k\right), \quad \Im\{c_k\} = 0, \quad (2.4.1)$$

where w_k is the sequence produced by the PRBS generator, and k corresponds to the carrier index.

2. Preamble Pilot Modulation

For the first UL OFDMA symbol, it shall be an all-pilot preamble. The pilots shall not be boosted and shall be modulated according to the following formulas:

$$\Re\{c_k\} = 2\left(\frac{1}{2} - w_k\right), \quad \Im\{c_k\} = 0. \quad (2.4.2)$$

2.4.2 TX/RX SRRC filter

We briefly introduce the SRRC filter based on [11] here. To avoid the complexity of an ideal lowpass filter and to simulate path delays at non-integer sample times, an interpolator is added to the transmitter to yield 4-times oversampled transmitter output. The square root raised cosine (SRRC) filter is used as the lowpass interpolation filter. The impulse response of this filter is given by

$$SRRC(t) = \frac{\sin\left(\pi\frac{t}{T_{sample}}(1-\alpha)\right) + 4\alpha\frac{t}{T_{sample}}\cos\left(\pi\frac{t}{T_{sample}}(1+\alpha)\right)}{\pi\frac{t}{T_{sample}}\left(1 - \left(4\alpha\frac{t}{T_{sample}}\right)^2\right)},$$

where α is the roll-off factor. One reason for adopting the SRRC filter is that for this filter the transmitter and receiver filters are matched to each other and there is no inter-sample interference introduced in the receiver when fully synchronized. Finally, the roll-off factor of SRRC filter is 0.155 with 57 taps, which is chosen to satisfy the power mask specified in 802.16a [11].

2.5 UL Synchronization Problems

Before the receiver can demodulate the subcarriers, it has to perform the synchronization task, since the OFDM systems can be extremely sensitive and vulnerable to synchronization errors. There are three major kinds of synchronization tasks:

1. Symbol synchronization [12]

The purpose of it is to find the correct position of the fast Fourier transform (FFT) window. Any misalignment of the FFT window will result in an evolving phase shift in the frequency domain symbols, leading to BER degradation. If the timing errors are so high that the FFT window of the receiver includes samples outside the data and guard intervals of the current OFDMA symbol, then the consecutive OFDMA symbols interfere, severely affecting the system's performance. Fig. 2.12(a) shows the correct FFT window. Fig. 2.12(b) shows an early FFT window that includes

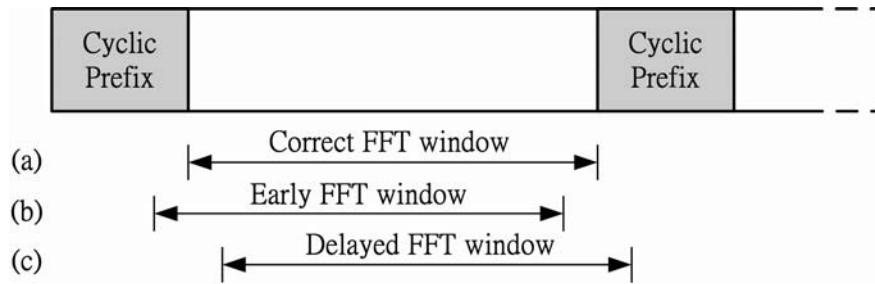


Fig. 2.12: Positioning of the FFT window.

samples of the data segment and the guard interval. Fig. 2.12(c) depicts a delayed FFT window that overlaps with the next OFDMA symbol. The second case will not introduce any interference, but the third is detrimental to the performance.

2. Sampling clock synchronization

The purpose of it is to align the receiver sampling clock frequency to that of the transmitter. The sampling clock errors can cause ICI. In addition, the sampling clock frequency error can result in a drift in the symbol timing and can further worsen the symbol synchronization problems. In this thesis, we will assume that the sample clocks of the users and the base station are identical.

3. Carrier synchronization

Carrier frequency offset can give rise to a shift of all the subcarriers and results in not only ICI but also multiple access interference (MAI). It is caused by the difference in the local oscillators of the transmitter and the receiver, or the Doppler spread introduced by motion. Carrier synchronization is a complex problem in the UL system, since all users share the total number of subcarriers and each user has its own carrier frequency offset.

In our system, the synchronization scheme is subject to the specifications of 802.16a. Thus we assume that after a successful initial synchronization and ranging, the mobile enters the time and frequency grid with a low offset in time and frequency [11]. Hence

no frequency synchronization is done in normal UL transmission. While this assumption may be suitable for fixed BS and SS, it is certainly debatable for multipath fading channels. However, for simplicity we leave it further consideration to future work.

2.6 UL Synchronization

The above discussion of the UL synchronization motivates our doing timing synchronization only. We now introduce the techniques used in our UL synchronization, the detection of symbol start time.

Our synchronization task is to find the first coming symbol. Different users' transmitted signals may not arrive at the same time, but the correlation peak may occur between them, as shown in Fig. 2.13 for an example of three users. If we use the detected peak location as the symbol start time, the corresponding useful time will include a part of the guard interval of the next symbol for the earlier arriving signals. Therefore, we have to find the exact instant of the first arriving signal to avoid ISI.

Since the subchannels are comprised by the subcarriers which are orthogonal to one another, we can assume that the orthogonality property still exists among subchannels unless the received signals from different users are subject to significantly different carrier

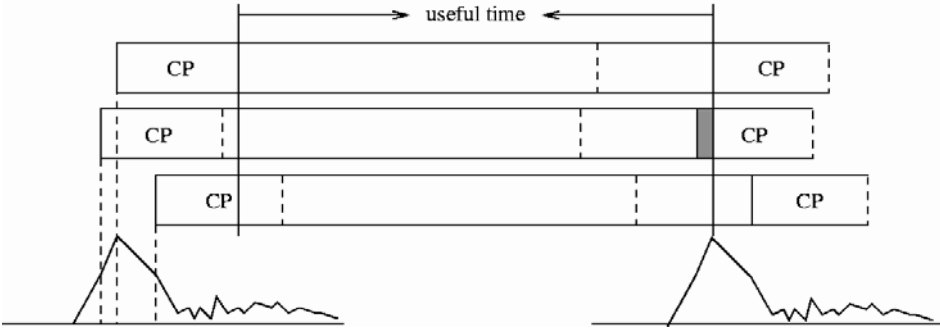


Fig. 2.13: Three UL signals arrive at different times, and the CP correlation peak may occur between them [11].

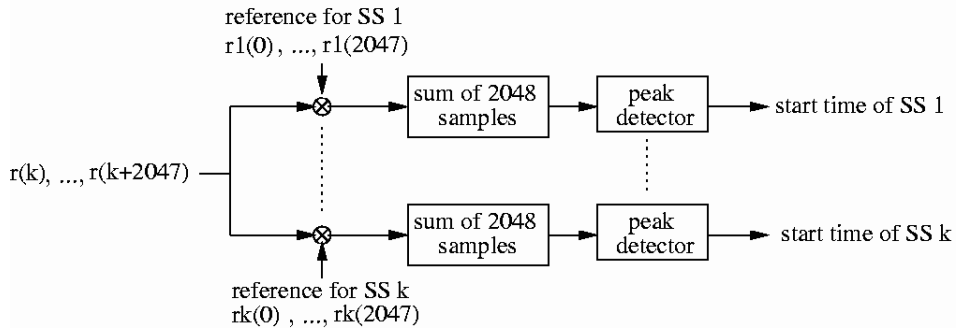


Fig. 2.14: Illustration of UL synchronization in time domain.

offsets. After passing through IFFT, the time domain signals which occupy different sub-channels in the frequency domain are uncorrelated if the channel has zero delay spread. Since the first coming symbol is an all-pilot preamble, the BS knows the exact values of each user's signals. Therefore, the signal transmitted by each SS in the UL preamble is deterministic and the BS can generate the same time domain signals as all SSs by taking IFFT. We show the block diagram depicting how the synchronization works in Fig. 2.14.

The received samples are correlated with the reference data string, which results from passing the preamble into the IFFT block. When the next sample arrives, we recompute the correlation. The start and stop times of the correlation are as illustrated in Fig. 2.15. The start time is decided by when the BS turns to receive signals. Note that this time shall be in the TTG interval. As the user arrival time may vary as much as 50% of the guard interval, we stop the correlation up to 50% of the guard interval earlier than the corresponding detected useful time.

Then, the peak locations of different SSs are compared as follows. We can find the peak location of each correlator which uses a distinct preamble, then we can know the peak locations of different SSs. Finally, we compare all these peaks and get the start location of the first coming signal.

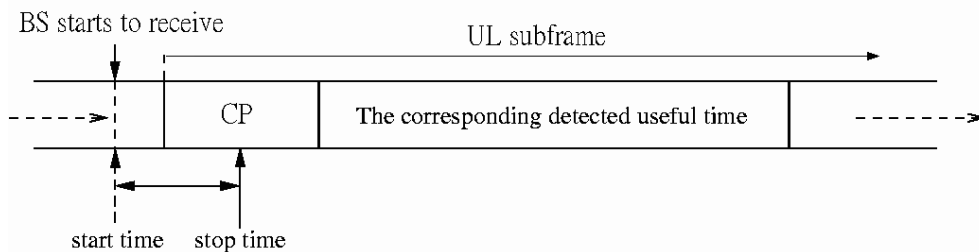


Fig. 2.15: The received samples and the time plan of the UL synchronization.

2.7 UL Synchronization Result

2.7.1 Simulation Parameters and Environments

Table 2.3 specifies the transmission parameters for our simulation. The uplink and downlink use the same frequency bands. The intercarrier spacing is thus 5.58 kHz and the symbol length (without cyclic prefix) is 179.2 μsec . In this section, we select the channel environment defined by ETSI for the evaluation of UMTS radio interface proposals. We employ the multipath ETSI “Vehicular A” channel model given in Table 2.4. The SNR is chosen to be 10 dB in the fading channels. Note that the receiver SNR specified in 802.16a is from 9.4 dB to 24.4 dB, so 10 dB, which is almost the worst condition, is a reasonable value for simulation. The maximum Doppler shifts of our simulation are shown in Table 2.5 for the speed from 0 to 100 km/hr.

Table 2.3: System Parameters Used in Our Study [14]

Number of carriers(N)	2048
Center frequency	6 GHz
Uplink / Downlink bandwidth (BW)	10 MHz
Carrier spacing (Δf)	5.58 kHz
Sampling frequency (f_s)	11.43 MHz
OFDM symbol time(T_s)	201.6 μsec (2304 samples)
Useful time (T_b)	179.2 μsec (2048 samples)
Cyclic prefix time (T_g)	22.4 μsec (256 samples)

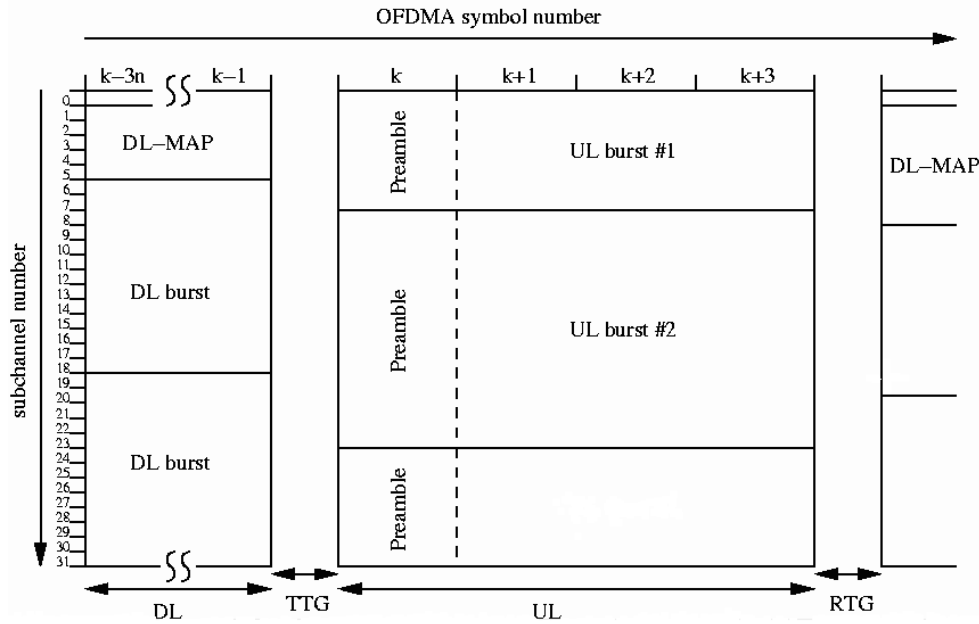


Fig. 2.16: Frame structure used in UL synchronization.

Fig. 2.16 shows the frame structure used in UL synchronization simulation, where SS1 transmits UL burst 1 using 8 subchannels and SS2 transmits UL burst 2 using 16 subchannels. The arriving times of burst 1 and burst 2 differ by 11.25% of the guard time, which is 16 samples. No ranging subchannel is allocated. Note that the start time of the preamble correlation is chosen to be 76 samples earlier than the UL subframe, and the stop time is 128 samples after the starting instant of the UL subframe.

Recall that the TTG is used for BS to turn around (from TX to RX). It is reasonable to assume that the transition instant is approximately at the midpoint of the TTG. Now TTG is 136 samples, and thus we assume that it is 60 samples after the start time of TTG. Figure 2.17 illustrates the transition instant for BS to turn around.

The reason for the stop time is as follows. According to IEEE 802.16a standard, all SSSs shall acquire and adjust their timings such that all uplink OFDM symbols arrive time coincident at the base station to an accuracy of 50% of the minimum guard-interval or better. Therefore, we assume that all SSSs arrive before the stop time, 50% of the guard

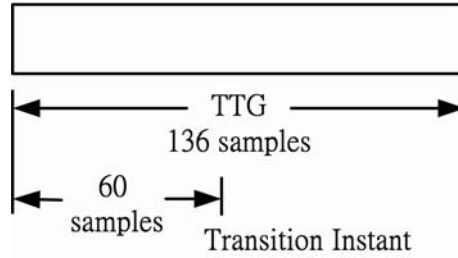


Fig. 2.17: The transition instant for BS to turn around.

Table 2.4: ETSI “Vehicular A” Channel Model in Different Units [23]

tap	relative delay (nsec or sample number)			average power		
	(nsec)	(4 oversampling)	(normal)	(dB)	(normal scale)	(normalized)
1	0	0	0	0	1.0000	0.4850
2	310	14	3 or 4	-1.0	0.7943	0.3852
3	710	32	8	-9.0	0.1259	0.0610
4	1090	50	12 or 13	-10.0	0.1000	0.0485
5	1730	79	20	-15.0	0.0316	0.0153
6	2510	115	29	-20.0	0.0100	0.0049

interval after the start time of the uplink subframe.

2.7.2 UL Synchronization

Fig. 2.18 shows the symbol time synchronization errors of the first coming signal under different Doppler spreads. If the Doppler shift is zero (speed = 0 km/hr), it is shown that we can always detect the correct symbol start time of the first coming signal.

Another interesting result is that when the speed increases, the distribution of the time synchronization errors is closely related to the power-delay profile of the multipath channel. Fig. 2.19 depicts the power-delay profile of the simulated channel with normal sample numbers and with normalized average power (see Table 2.4). Fig. 2.20 shows the resulting time synchronization error distribution. Comparing these two figures, we see that the different time offsets obtained at the synchronizer output almost coincide with the sample number of the multipath delays. Furthermore, the occurrence probabilities at

Table 2.5: Relation Between Speed and Maximum Doppler Shift at Carrier Frequency 6 GHz. Subcarrier Spacing is 5.58 kHz

Speed (km/hr)	Doppler shift (Hz)	$f_d T_s$
0	0	0
20	111	0.0224
40	222	0.0448
60	333	0.0672
80	444	0.0896

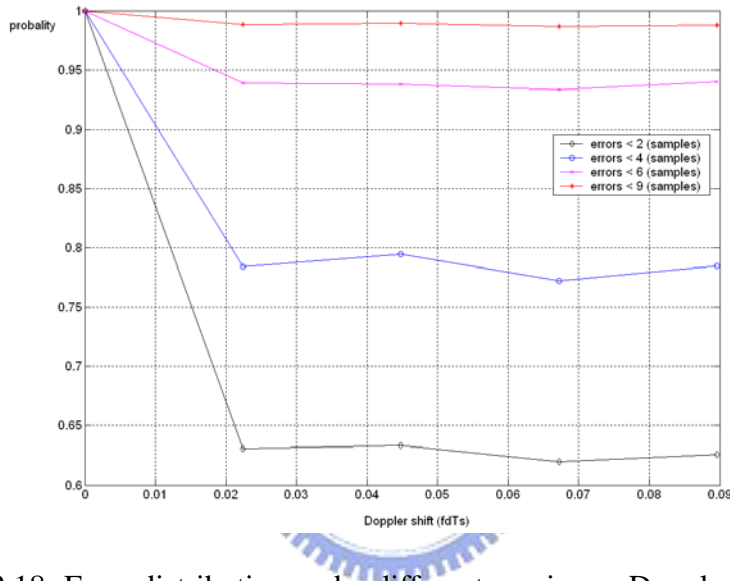


Fig. 2.18: Error distribution under different maximum Doppler shifts.

the different time offsets are proportional to the relative average power of the paths. Note that the Doppler shift has no obvious effects on this synchronization scheme except when it is very small.

As the correlation is done for each SS, we can detect the arriving time of each SS. We find that the timing error distributions of the late arriving SS are almost the same as the result of the first coming SS. No matter when the signal arrives, the synchronization performance has no significant differences. In summary, we can detect the start time of all signals from different SSs and this information could be helpful to channel estimation.

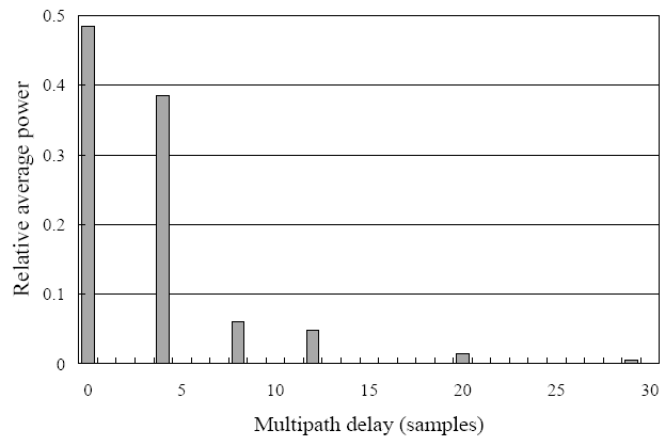


Fig. 2.19: Power-delay profile of the multipath channel [14].

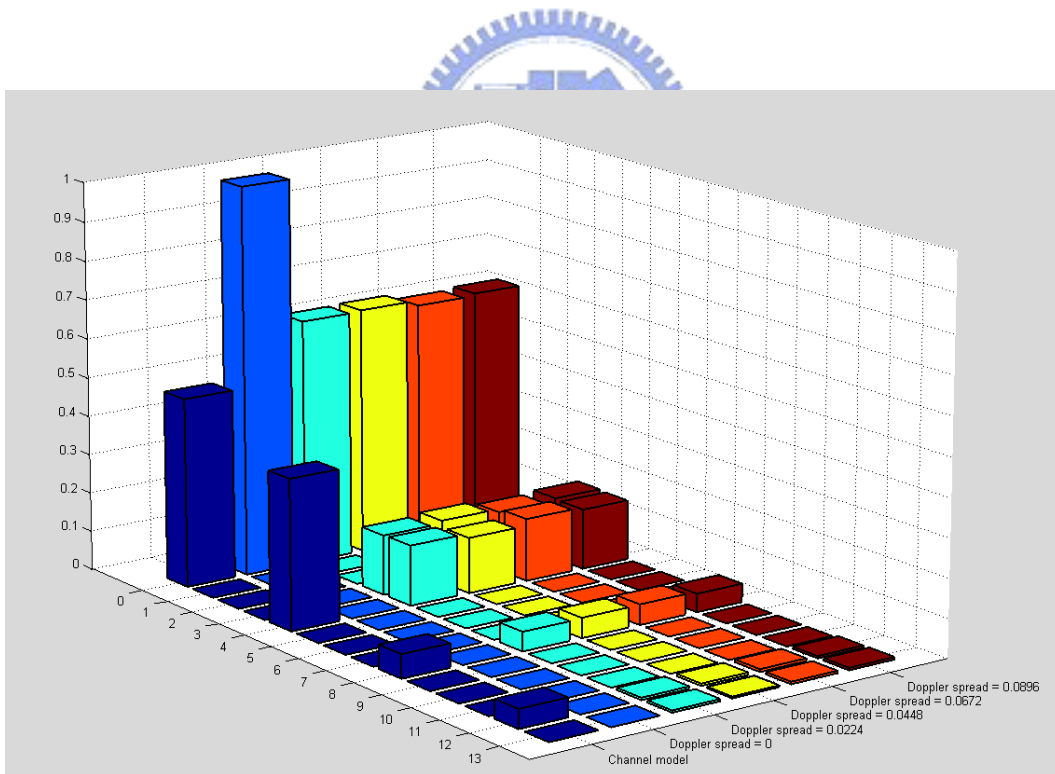


Fig. 2.20: Performance of UL symbol time synchronization: error distribution under different maximum Doppler shifts.

Chapter 3

Introduction to the DSP Implementation Platform

In this chapter, we introduce the DSP platform utilized in our implementation. The platform includes a DSP board, DSP core, and the communication mechanism between the host PC and the DSP target.

3.1 DSP Board [16]

The DSP board, Quixote-II, is a 64-bit cPCI 6U board for advanced signal capture, generation and co-processing. Figure 3.1 shows a picture of the board. Quixote-II associates with one TI (Texas Instruments) TMS320C6416 DSP with a Xilinx's Virtex-II FPGA, providing processing flexibility, efficiency, and delivering performance. The block diagram of Quixote-II is shown in Figure 3.2. On our board, the FPGA is a six-million-gate one.

The board's primary features are as follows:

1. 600 MHz 32-bit fixed-point TMS320C6416 DSP offers processing power of 4800 MIPS.
2. An onboard 32 MB SDRAM for DSP chip, with advanced cache controllers.
3. 64/32-bit 33 MHz PCI interface for busmastering data between the card and the memory.

4. 14-bit 105 MSPS I/Q input channels and output channels for A/D and D/A.

3.2 DSP Chip [18]

The DSP chip, TI's TMS320C6416, employs the "VelociTI" architecture, a variant of the traditional VLIW architecture, which consists of multiple execution units running in parallel, performing multiple instructions during one cycle time. It is a 32-bit fixed-point DSP, with processing speed at 600 MHz, delivering 4800 MIPS.

The C6416 core CPU, which is shown in Fig. 3.3, consists of 64 general-purpose 32-bit registers and eight functional units. These eight functional units contain two multipliers and six arithmetic units. It allows users to develop highly effective RISC-like code for fast development time.

The C6416 uses a two-level cache-based architecture with 16 kB of L1 data cache, 16 kB of L1 program cache, and 1 MB of L2 data/program cache. On-chip peripherals include two multichannel buffered serial ports (McBSPs), two timers, a 16-bit host port interface (HPI), a 32-bit external memory interface (EMIF), a direct memory access (DMA) controller and an enhanced direct memory access (EDMA) controller.

The following gives some sketch of the units just mentioned above:

- The EDMA controller transfers data between the memory without passing through

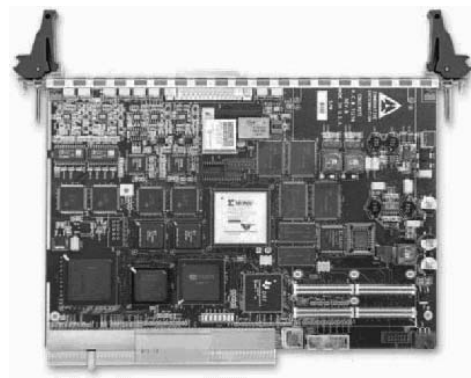


Fig. 3.1: Quixote-II board [24].

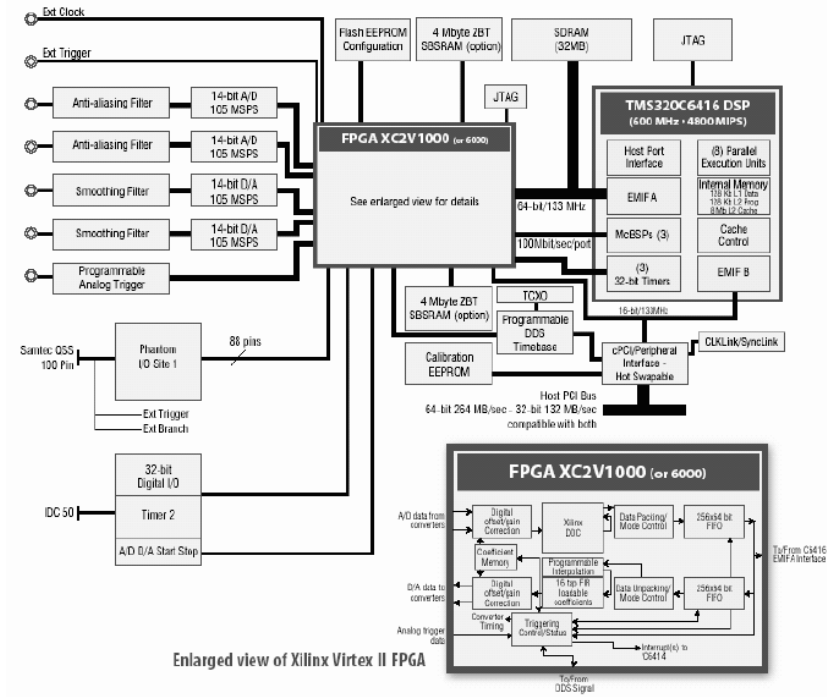
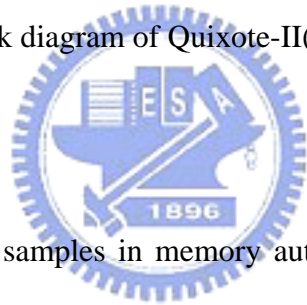


Fig. 3.2: Block diagram of Quixote-II(from [16]).



the DSP core.

- McBSPs can buffer serial samples in memory automatically with the aid of the DMA/ EDMA controller.
- HPI is a parallel port through which a host processor can directly access the CPU's memory space.
- EMIF provides the interface for the DSP core to connect with several external devices, allowing additional data and program space.

The C6416 has two 64-bit internal ports to access internal data memory. It supports double word loads and stores. There are four 32-bit paths for loading/storing data from memory to the register file. C6416 has two register files (A and B), each containing 32 32-bit registers for a total of 64 general-purpose registers. The general-purpose registers can be used for data, data address pointers, or condition registers. The C6416 register

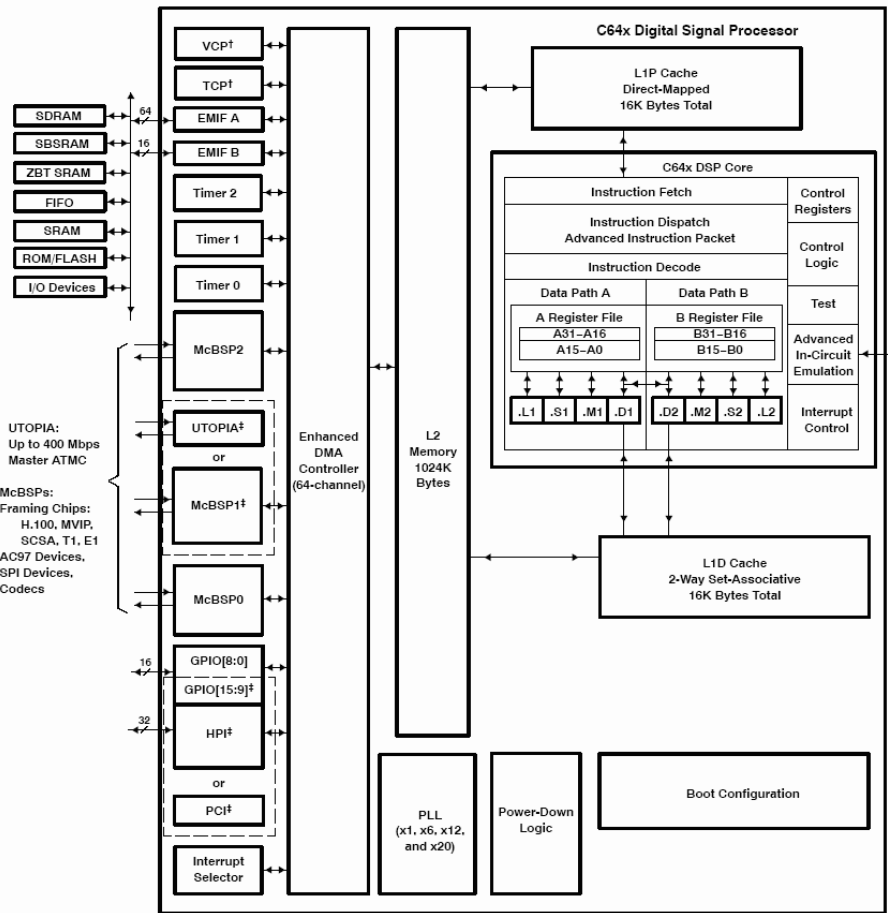


Fig. 3.3: Functional block and CPU (DSP core) diagram [17].

file supports packed 8-bit types and 64-bit fixed-point data types. Packed data types store either four 8-bit values or two 16-bit values in a single 32-bit register, or four 16-bit values in a 64-bit register pair. Note that the C6416 does not directly support floating-point data types.

The eight functional units in the C6416 data paths can be divided into two groups of four; each functional unit in one data path is almost identical to the corresponding unit in the other data path. The two sets of functional units, along with two register files, compose sides A and B of the DSP core. Figure 3.4 illustrates the C6416 DSP CPU. From this figure, we see that the C6416 CPU contains:

- Program fetch unit

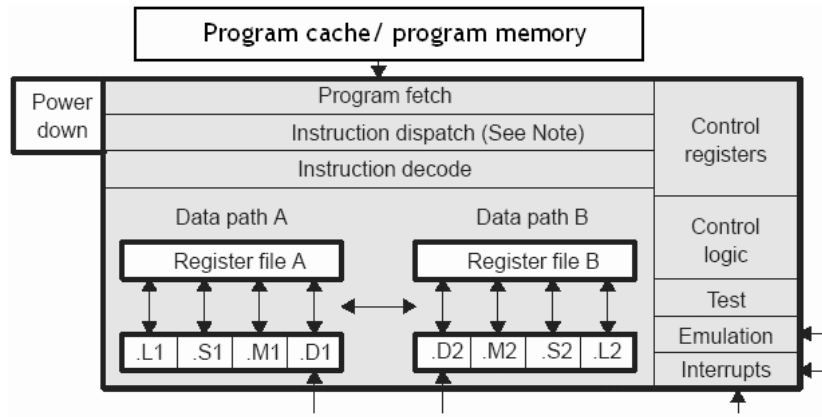


Fig. 3.4: The C64x CPU block diagram [18].

- Instruction dispatch unit, with advanced instruction packing
- Instruction decode unit
- Control registers
- Control logic
- Test, emulation, and interrupt logic



The details of each functional units are given in Tables 3.1 and 3.2. Most data lines in the CPU support 32-bit operands, and some support long (40-bit) and double word (64-bit) operands. Each functional unit has its own 32-bit write port into a general-purpose register file. All units ending in 1 (for example, .L1) write to register file A, and all units ending in 2 write to register file B. Each functional unit has two 32-bit read ports for source operands src1 and src2. Four units (.L1, .L2, .S1, and .S2) have an extra 8-bit-wide port for 40-bit long writes, as well as an 8-bit input for 40-bit long reads. Because each unit has its own 32-bit write port, when performing 32-bit operations all eight units can be used in parallel every cycle.

Table 3.1: Functional Units (.L, .S) and Operations Performed [18]

Functional Unit	Fixed-Point Operations
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits Byte shifts Data packing/unpacking 5-bit constant generation Dual 16-bit arithmetic operations Quad 8-bit arithmetic operations Dual 16-bit min/max operations Quad 8-bit min/max operations
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from control register file (.S2 only) Byte shifts Data packing/unpacking Dual 16-bit compare operations Quad 8-bit compare operations Dual 16-bit shift operations Dual 16-bit saturated arithmetic operations Quad 8-bit saturated arithmetic operations

Table 3.2: Functional Units (.M, .D) and and Operations Performed [18]

Functional Unit	Fixed-Point Operations
.M unit (.M1, .M2)	<p>16 x 16 multiply operations</p> <p>16 x 32 multiply operations</p> <p>Quad 8 x 8 multiply operations</p> <p>Dual 16 x 16 multiply operations</p> <p>Dual 16 x 16 multiply with add/subtract operations</p> <p>Quad 8 x 8 multiply with add operation</p> <p>Bit expansion</p> <p>Bit interleaving/de-interleaving</p> <p>Variable shift operations</p> <p>Rotation</p> <p>Galois Field Multiply</p>
.D unit (.D1, .D2)	<p>32-bit add, subtract, linear and circular address calculation</p> <p>Loads and stores with 5-bit constant offset</p> <p>Loads and stores with 15-bit constant offset (.D2 only)</p> <p>Load and store double words with 5-bit constant</p> <p>Load and store non-aligned words and double words</p> <p>5-bit constant generation</p> <p>32-bit logical operations</p>

3.3 Data Transmission Mechanism [16]

In this section, we introduce the data transmission mechanism that Quixote-II supports. We will make use of one, CPU busmastering, to realize the transmission between the host PC and the DSP target, Quixote-II. From [16], we know that there are three schemes provided by the Quixote baseboard. They are DSP streaming interface, CPU busmastering interface, and packetized message interface. We now introduce them in the following subsections.

3.3.1 DSP Streaming Interface

The Quixote supports using PCI busmastering for the highest data rate streaming between the host and the target. 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. The target DSP code can then take any needed action to cover the interruption. When service resumes, the system will move the backed up data through the system to the application normally.

Figure 3.5 shows the block diagram of DSP streaming 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. This arrangement allows the DSP to process the streams as they move from the application to the hardware.

- Software implementation:

DSP streaming is initiated and started on the host, using the Caliente component, which handles bi-directional streaming of data between the host memory and the

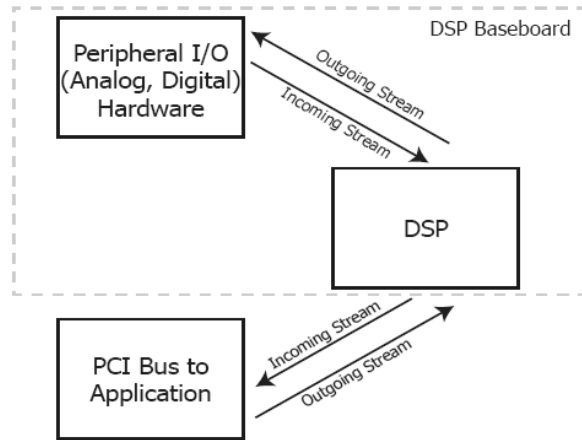


Fig. 3.5: DSP streaming mode [16].

target DSP. On the target, the DSP interface uses one 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.

- Hardware implementation:

The Quixote baseboard has a 32 or 64 bit PCI interface, 33 MHz, compatible with 3V or 5V signalling PCI bus systems. This interface supports both busmastering and “slave” interfaces to the baseboard and supports data burst rates of 264 MB/sec for 64-bit systems, or 132 MB/sec for 32-bit systems.

The baseboard uses busmastering to host memory as the primary method for moving large amounts of data in the system. From the DSP perspective, the busmastering interface is a bi-directional FIFO that manages the interaction with the host memory. The PCI controller is responsible for moving data to and from the host as required by the DSP. Slave accesses, from the host processor to the target DSP, are used to support configuration, control, and communications.

3.3.2 CPU Busmastering Interface

The TI 64x baseboard is capable of using PCI busmastering to move data between target and host memory. This additional busmaster channel can be used to transfer data between host and target applications.

The primary busmaster interface is based on a streaming model where logically data constitute an infinite stream between the source and destination. This model is more efficient because the signalling between the two parties in the transfer can be kept to a minimum and transfers can be buffered for maximum throughput. On the other hand the streaming model can have relatively high latency for a particular piece of data. This is because a data item may remain in internal buffering until subsequent data accumulates to allow for an efficient transfer. The CPU busmaster interface uses a different model: it transfers discrete blocks between the source and destination. Each data buffer is transferred completely to the destination in a single operation. Only if several transfers are requested at once will any delay in beginning transmission occur, as multiple requests have to be serialized through the single hardware system.

The data buffers transferred can be of different sizes. Each requested buffer is interrogated for its size and fully transmitted. At the destination, the destination buffer is re-sized to allow the incoming data to fit. If the buffer given is too small for the data, it will be reallocated to allow the transfer. Reallocating buffers can take some time, for best performance buffers should be pre-sized to be large enough for the largest transfer expected. This will make allocation of buffers at critical times unnecessary.

CPU busmastering uses a simple blocking interface for its sending and receiving functions. The sending function will not return until the transfer has completed and the buffer is ready for reuse. Similarly, the receiving function waits until data have arrived from the data source and transferred into the data buffer before returning. At this point the buffer is ready for use. This blocking allows sequences of transfers managed by a simple sequence of calls to transfer functions. Since the transfer functions are blocking, they are

best avoided in the main user interface thread of a Windows application. The GUI will appear to be frozen until the transfer has completed. For best results, the data transfer functions should be placed in separate threads on the target and host applications. In fact, each direction of transfer should have its own thread, so that the two directions of transfer can interleave as much as possible.

The CPU busmaster interface allows separate channels of data between the target and the host. Using separate channels allows multiple, independent data streams to be maintained between the target and host. At present, only a single channel is supported.

The largest transfer allowed is half the total size of the DMA buffer allocated by the INF file (a kind of files used for software/firmware installation in windows system) when the driver is installed. Half of the memory is dedicated to each direction. The default buffer size in the INF is 0x200000 bytes, so the maximum transfer is 1 MB.

PciTransfer::Send() sends the contents of a Buffer-derived object to the target on the channel Channel. All of the data in the buffer are transferred. There is no means of sending a partial buffer. Only channel 0 is currently supported. The function will not return until the block has been transferred to the host. The use of the base buffer class allows any of the IntBuffer, CharBuffer, FloatBuffer and similar classes to be sent across the interface. The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error. PciTransfer::Recv() waits for data to arrive from the target, then returns the data in the buffer provided. The data must be sent on the same channel as the Channel argument. The Buffer will be re-sized to fit the data transferred from the source. If the buffer is too small, this may involve a reallocation of the data block. The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

3.3.3 Packetized Message Interface

The DSP and host have a lower bandwidth communications link for sending commands or out-of-band information between target and host. These packets can provide the users another way to send commands for many purposes. For example, we can use these packets to tell the target what to do next, or when to do next. These packets provide a very important “bridge” for the host and the DSP.

A set of sixteen mailboxes in each direction to and from the host PC are shared with the DSP to allow for an efficient message mechanism that complements the busmastering interface. These mailboxes have a handshake mechanism that signals the recipient for the availability of data, and a corresponding signalling to the sender when the message was received. Data rate is limited to about 56 kB per second. Higher data rate requirements should use the busmastering interface.

A single bi-directional path can be set up with minimal configuration for applications with simple communication needs. A virtually unlimited number of independent communication channels may be set up to run in parallel, with messages on each channel directed to their own receiver on the other side. Figure 3.6 shows a single bi-directional path between the DSP and the host PC. As we see, this figure is divided into two parts, one is host application, and the other is target application. On the host side, CIIMessage class can encapsulate the packet, that may contain up to 14 32-bit data words plus two 32-bit header words, to be transmitted. On the target, the corresponding class is called IIMessage. Messages sent by the target are collected into CIIMessage objects for delivery to the event handlers dedicated to respond to the messages. For all practical purposes, we can think of the Message System as exchanging IIMessage/CIIMessage objects.

The header portion of the Message Packet contains some system data and some fields that can be used by the application. Table 3.3 gives the header field that CIIMessage can read.

The 14 words of Data are accessible as array. Table 3.4 shows these methods all have

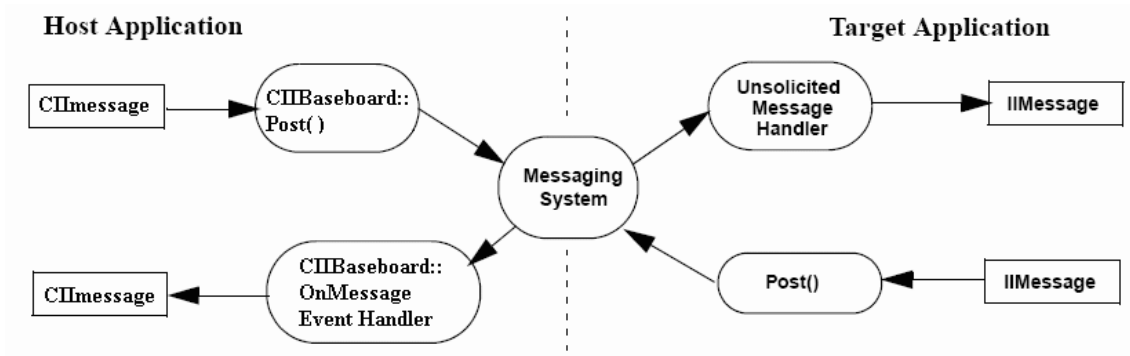


Fig. 3.6: Simple target to host messaging configuration [16].

Table 3.3: CIIMessage Header Field [16]

Channel	Message Channel (may be overwritten by system)
TypeCode	Message or Command Type
MessageId	Message counter or other user data
IsReplyExpected	Set if reply is needed. Free for use in application.

an additional argument giving the index into the data section.

On the target side, the Pismo library supports a very similar class, IIMessage, to contain the message. The header field access and the data section interface are identical to the host side, CIIMessage.

The packetized message system is event driven. When the sender posts a message packet, at the first available opportunity the packet is loaded in the communication registers and an interrupt generated on the receiving side. On the receiver, the interrupt is

Table 3.4: CIIMessage Data Section Interface [16]

Data{}	Access the data region as 32-bit integers (0-13)
AsFloat{}	Access the data region as floating point data. (0-13)
AsShort()	Access the data region as 16-bit integers. (0-27)
AsChar()	Access the data region as 8-bit characters. (0-55)

detected and the message removed and enqueued for later processing. The sender is then acknowledged that the previous packet has been removed and the hardware is free for another transmission. The receiver then analyzes the message and distributes it to the proper handler for processing.

For more information, refer to [16].



Chapter 4

Integration and Optimization of the IEEE 802.16a OFDMA TDD Uplink Transmitter-Receiver System

In previous chapters, the components of the uplink transceiver system have been introduced and the DSP implementation platform has been described. In this chapter, we discuss the major topic of this thesis — the integration and optimization of the specified uplink transceiver system on II's Quixote DSP baseboard, using the TI TMS320C6416 DSP chip. At first, we briefly introduce the entire structure of our system, its transmission mechanism, and the precision of the fixed-point numbers that we use. Secondly, we introduce the DSP code development environment and some features of the TI C6000 family DSP tools for doing compiler level optimization. Then, we discuss optimization of the major blocks in the uplink transceiver, including the TX/RX SRRC filters, the TX IFFT, and others. Finally, we present the improvement after the efforts we have made by showing the simulation profile generated by TI's Code Composer Studio (CCS) built-in profiler.

4.1 Structure of the Implemented System

The structure of the uplink transmitter and receiver system is shown in Fig. 4.1. There are two SSs and one BS. In consequence, the FEC scheme and the channel modulation

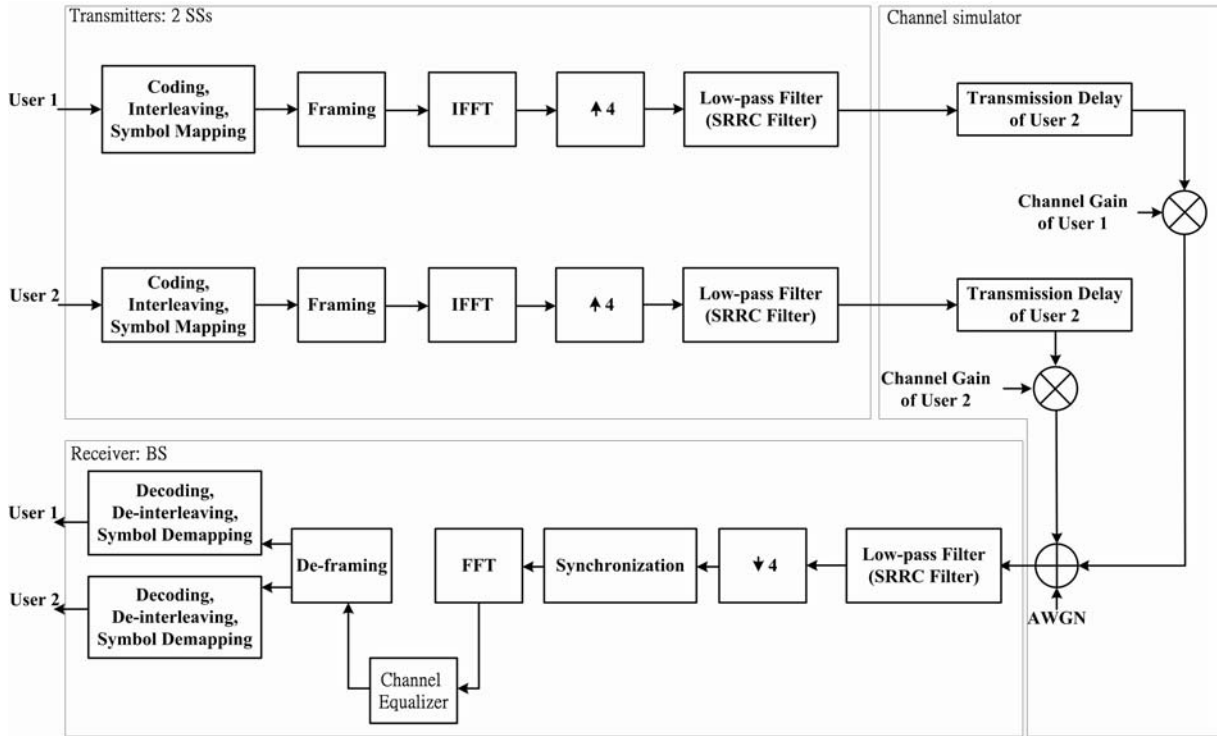


Fig. 4.1: Structure of implemented system.

scheme each ideally needs to use one individual DSP board and be linked by the PCI port on the personal computer (PC), which are illustrated in Figs. 4.2 and 4.3, respectively. In our work, however, we merge the two on only one DSP board, for the reason that the data transmission mechanism across the DSP boards is too complex to realize. The FEC part consists of randomizer/de-randomizer, Reed-Solomon encoder/decoder, convolutional encoder/decoder, and interleaver/de-interleaver. The channel modulation part consists of data modulation/demodulation, data framing/deframing, IFFT/FFT, 4-times upsampling/downsampling, and SRRC filtering.

Due to the unknown system software bug, we are unable to run the channel equalizer() on the DSP baseboard yet. We think memory allocation problem may be the reason for the bugs. Hence, only parts of the functions in the receiver (i.e., RX_SRRC() and RX_sync) are workable now. The implementation of other parts of the receiver on DSP are now leaved to the future work.

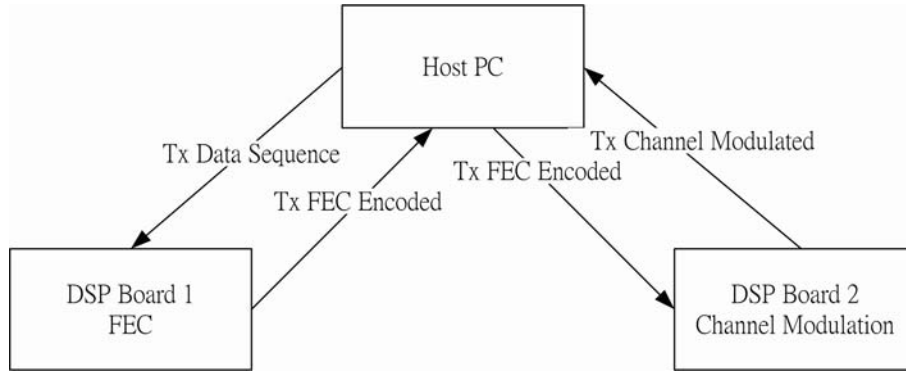


Fig. 4.2: System structure on transmitter side (modified from [15]).

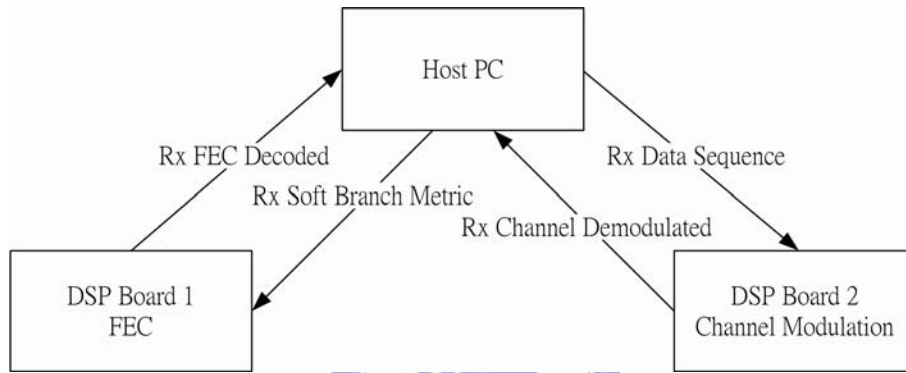


Fig. 4.3: System structure on receiver side (modified from [15]).

4.1.1 CPU Busmastering Interface

The data transmission mechanism between the DSP board and PC employs the CPU busmastering interface described in chapter 3, because this mechanism is relatively easier to implement than the data streaming mode. The size of the transmitting blocks from the transmitter is chosen to be 8200 samples, that is, 2050 samples before the 4-times up-sampling. Here one sample means a complex number that contains the real part and the imaginary part. The reasons why we do not use 2304 samples (one OFDM symbol) as block size are given below. The size of one OFDMA frame in our work consists of three downlink symbols, four uplink symbols, one TTG, and one RTG. Figure 2.16 shows the frame structure. After 4-times upsampling, the SS transmits 9216 samples per OFDM

symbol, and 544 samples in both TTG and RTG. Therefore, the SS transmits 65600 samples in one frame time. We assume that we do not know the precise boundary of the received symbols on the receiver side. To cope with this assumption, we can let our transmitting block size be a constant value. But we must ensure that the output sample time is an integer fraction of the input time, or we will have to take care of complicated timing. Mathematically, we have

$$\text{output sample time} = \frac{\text{input sample time}}{N},$$

where N is a positive integer bigger than 1.

In our work, the input sample time is 65600 samples. The value of N is chosen to be 8, resulting in the output sample time of 8200 samples. There are no particular theoretical reasons to choose 8 as N . We just let the output sample time be close to one OFDM symbol, 9216 samples.

Figure 4.4 shows the organization of our transmitter and receiver. Note that the transmitter will not return until the transfer has completed and the buffer is ready for reuse, and the receiver waits until data has arrived from the data source and transferred into the data buffer before returning. The transmitters (2 SSs) and the receiver (BS) are actually both on the same DSP chip, and the channel simulators for both users are located in the host PC. The reason for excluding the channel simulator from the DSP chip is because the C6416 is a fixed-point DSP and floating-point operations on it are time-consuming. Therefore, we exclude the channel simulator from the DSP chip, for it uses expensive floating-point operations.

Since there are two SSs in the transmitter side, the buffer size of the transmitting block is 16400 samples, that is, 2 times the 8200 samples. The receiver's block size is 8200 samples. In the following section, we first introduce the data format we use to represent one sample.

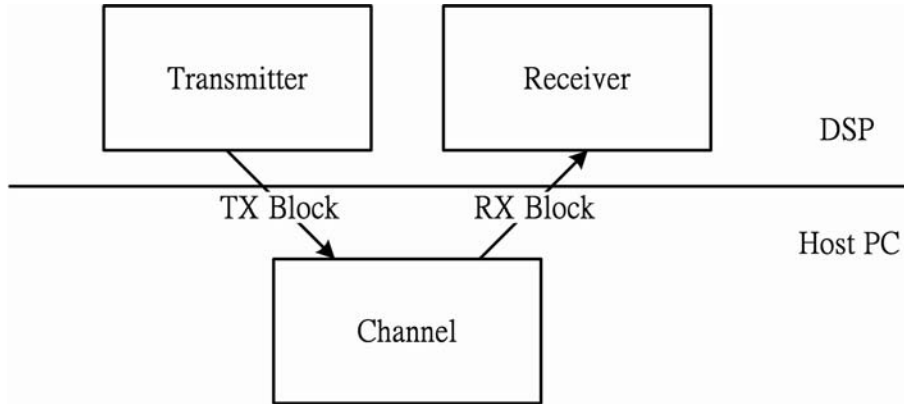


Fig. 4.4: Organization of transmitter and receiver using CPU busmastering interface.

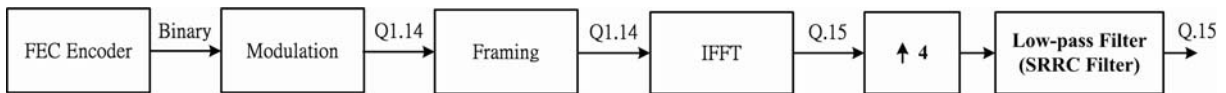


Fig. 4.5: Fixed-point data formats at the transmitter side.

4.2 Fixed-Point Data Formats

For improving the speed and saving the memory, we have to work with fixed-point numbers instead of floating-point numbers. The data formats we use in the transmitter are shown in Fig. 4.5. Since the DSP chip supports 16×16 multiply operations, and [19] suggests that use of the short data type (16 bits) for fixed-point multiplication inputs whenever possible, most of the data types are chosen to be 16-bit in our implementation:

- Data format before IFFT is Q1.14, which is in the range $[-2,2]$.
- Data format after IFFT is Q.15, which is in the range $[-1,1]$.

The Q1.14 format places the sign bit in the leftmost position, followed by 1 integer bit and 14 fractional bits (Table 4.1), and the Q.15 format places the sign bit in the leftmost position, and the remainder 15 bits are fractional ones. We explain the reasons for adopting these representations below.

Table 4.1: Q1.14 Bit Fields

Bits	15	14	13	...	1	0
Value	S	I0	Q13	...	Q1	Q0

After the binary sequence passes through the modulator, the range of data values, at normalized symbol energies as shown in Table 4.2. The widest range occurs in the 64-QAM, it is $[\frac{-7}{\sqrt{42}}, \frac{7}{\sqrt{42}}]$, and the Q.15 can not cover this range. Thus, Q1.14 is the suitable range to use.

Then, the range of data values at the framing block's output is located in $[-\frac{3}{4}, \frac{3}{4}]$, where the values $\pm\frac{3}{4}$ occur in the UL preamble carriers [1]. Again, it can be represented by the data format Q1.14. The fractional part of the fixed-point number in this system is 14 bits; hence the finest fractional resolution is $2^{-14} = 6.10 \times 10^{-5}$.

Now, we focus on the IFFT's output range. The 2048-point IFFT defined as

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)W_N^{-kn}, n = 0, \dots, N - 1, \quad (4.2.1)$$

where $N = 2048$, $W_N = e^{-j(2\pi/N)}$, $X(k)$ is the input sequence, and $x(n)$ is the resulting output sequence. This function is implemented by the 16-bit FFT function DSP_fft16x16r() provided in the TI TMS320C64x DSP library (DSPLIB) [20]. The detailed operation of it can be found in later sections. Because of the factor $\frac{1}{2048}$, the range of the output data values will be smaller than 1. For this reason, the data format after FFT is set to be Q.15. The fractional part of the fixed-point number in this system is 15 bits; hence the finest fractional resolution is $2^{-15} = 3.05 \times 10^{-5}$. Finally, the data field of the SRRC filter's output is Q.15, and it's SNR is 44.90dB comparing to the floating-point results.

In the receiver side, the main consideration of setting fixed-point formats is that the multiplier operations are always 16×16 . We show the data formats in the receiver side in Fig. 4.6. The data formats are:

- The data format before the synchronization is Q.15, which is in the range [-1,1].

Table 4.2: Range of Data Values After Modulation

Modulation	Range
QPSK	$[\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$
16-QAM	$[\frac{-3}{\sqrt{10}}, \frac{3}{\sqrt{10}}]$
64-QAM	$[\frac{-7}{\sqrt{42}}, \frac{7}{\sqrt{42}}]$

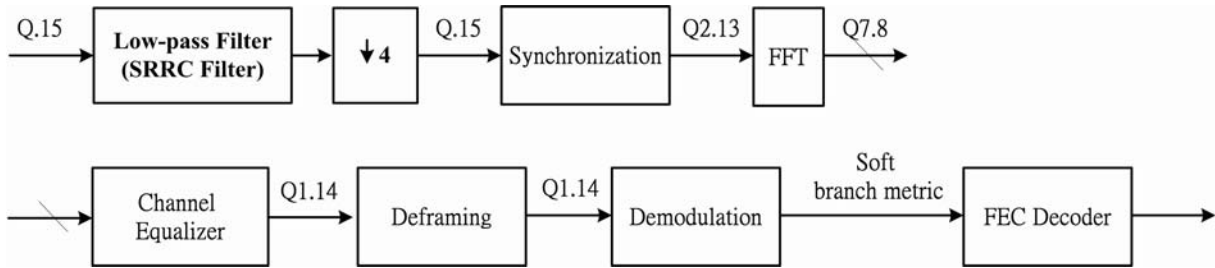


Fig. 4.6: The fixed-point data formats at the receiver side.

- The data format of the FFT input is Q2.13, and the output is Q7.8.
- The data format after the channel equalizer is Q1.14.

In the receiver side, we must take into account the channel gain introduced by the fading channel. Since the fading coefficients implemented are not normalized, it is better for us to give some guard bits to the integer part of the FFT input. Thus, we set the data format of the FFT input to Q2.13. This prevents data overflows in the FFT output. According to [20], the FFT function we use scale the output by 5 bits (i.e., $\gg 5$), to prevent output overflow. As a result, the output of the FFT is Q7.8. The finest fractional resolution is $2^{-8} = 3.91 \times 10^{-3}$.

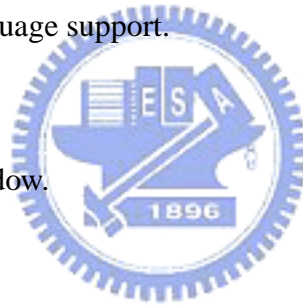
For simplicity, we assume that the receiver knows the frequency response of the channel. We implement a zero forcing equalizer, which is simply an inverse filter which inverts the frequency response of the channel. After the channel equalizer, the output shall be in the range $[-2,2]$ as the transmitter output. Hence, we set the data format to Q1.14.

4.3 TI's Code Development Environment [21]

In the following sections, we introduce the software environments we utilize in our work and how to successfully develop an efficient DSP code as quickly as possible. Then, we introduce some important and useful techniques to improve the program speed performance. The optimization of each block in our work is discussed after introduction of the software environment.

The Code Composer Studio, TI's GUI development tool, is the software platform that we use to develop and debug the projects. Some main features of it are listed below:

- Real-time analysis.
- Source code debugger common interface for both simulator and emulator targets.
 - C/C++ assembly language support.
 - Simple breakpoints.
 - Advanced watch window.
 - Symbol browser.
- DSP/BIOS support.
 - Pre-emptive multi-threading.
 - Interthread communication.
 - Interrupt handling.
- Chip Support Libraries (CSL) to simplify device configuration. CSL provides C-program functions to configure and control on-chip peripherals.
- DSP libraries for optimum DSP functionality. The DSP library includes many C-callable, assembly-optimized, general-purpose signal-processing and image/video processing routines. These routines are typically used in computationally intensive



real-time applications where optimal execution speed is critical. The TMS320C64x digital signal processor library (DSPLIB) provides some routines shown below:

- Adaptive filtering.
- Correlation.
- FFT.
- Filtering and convolution.
- Math.
- Matrix functions.
- Miscellaneous.

In our project, some routines are used in the implementation, such as FFT and filtering. We introduce them in later sections.

4.3.1 Code Development Flow [19]

The recommended code development flow involves utilizing the C6000 code generation tools to aid in optimization rather than forcing the programmer to code by hand in assembly. These advantages allow the compiler to do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation. These features simplify the maintenance of the code, as everything resides in a C framework that is simple to maintain, support, and upgrade. Figure 4.7 illustrates the three phases in the code development flow. Because phase 3 is usually too detailed and time consuming, most of the time we will not go into phase 3 to write linear assembly code unless the software pipelining efficiency is too bad or the resource allocation is too unbalanced. The following techniques can be used to analyze the performance of our specific code regions:

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

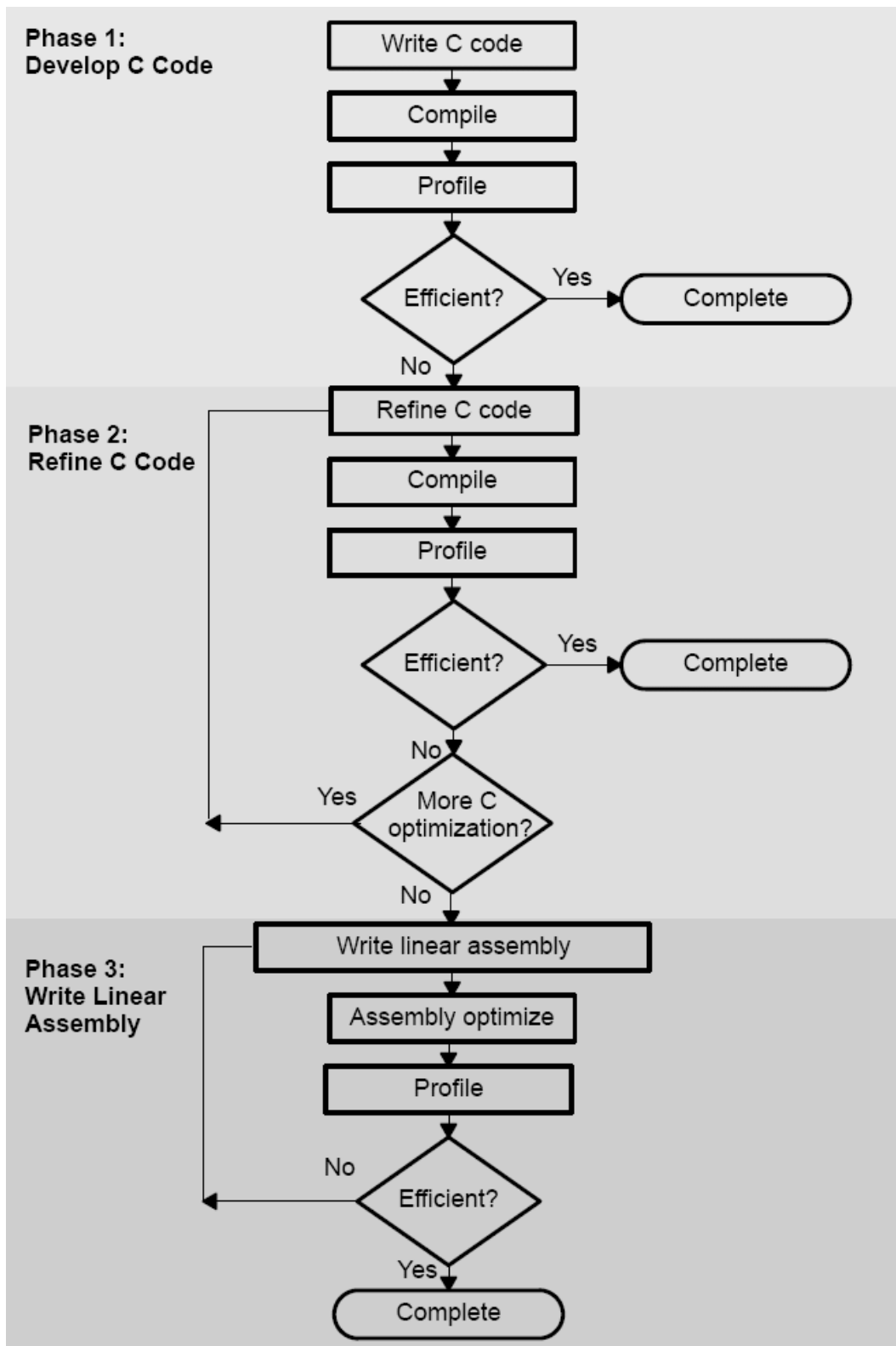


Fig. 4.7: Code development flow of C6000 (from [19]).

- Use the profile mode of the stand-alone simulator. This can be done by compiling our code with the `-mg` option and executing `load6x` with the `-g` option. Then enable the clock and use profile points and the `RUN` command in the Code Composer debugger to track the number of CPU clock cycles consumed by a particular section of code. Use “View Statistics” to view the number of cycles consumed.

Usually, we use the second technique above to analyze our C code performance. The feedback of the optimization result can be obtained with the `-mw` option. It shows some important results of the assembly optimizer of the particular loop. In our analysis, this shall be taken into consideration for improving the computational speed of certain loops in our program.

4.3.2 Compiler Optimization Options [19]

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

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

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

- * Propagates arguments into function bodies when all calls pass the same value in the same argument position.
 - * Identifies file-level variable characteristics.
- -k: Keep the assembly file to analyze the compiler feedback.
 - -pm -op2: In the CCS compiler option, -pm and -op2 are combined into one option.
 - -pm: Gives the compiler global access to the whole program or module and allows it to be more aggressive in ruling out dependencies.
 - -op2: Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler. This improves variable analysis and allowed assumptions.
 - -mh<n>: Allows speculative execution. The appropriate amount of padding, n, must be available in data memory to insure correct execution. This is normally not a problem but must be adhered to.
 - -mw: Produce additional compiler feedback. This option has no performance or code size impact.
 - -mi: Describes the interrupt threshold to the compiler. If we know that no interrupts will occur in our code, the compiler can avoid enabling and disabling interrupts before and after software pipelined loops for a code size and performance improvement. In addition, there is potential for performance improvement where interrupt registers may be utilized in high register pressure loops.

4.3.3 Software Pipelining [22]

Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel. The compiler always attempts to software

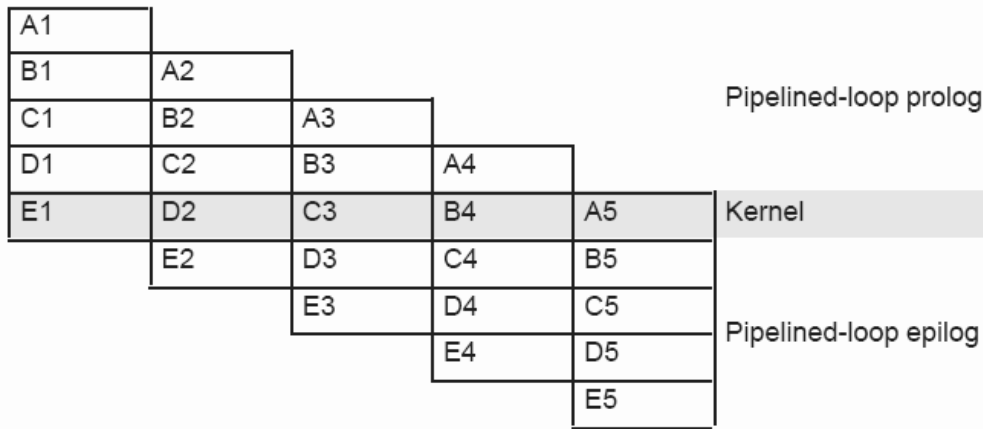


Fig. 4.8: Software pipeline loop (from [18]).

pipeline. Figure 4.8 illustrates a software pipelined loop. The stages of the loop are represented by A, B, C, D, and E. In this figure, a maximum of five iterations of the loop can execute at one time. The shaded area represents the loop kernel. In the loop kernel, all five stages execute in parallel. The area above the kernel is known as the pipelined loop prolog, and the area below the kernel is known as the pipelined loop epilog.

But under the conditions listed below, the compiler will not do software pipelining [19]:

- If a register value lives too long, the code is not software-pipelined.
- If a loop has complex condition code within the body that requires more than five condition registers, the loop is not software pipelined.
- A software-pipelined loop cannot contain function calls, including code that calls the run-time support routines.
- In a sequence of nested loops, the innermost loop is the only one that can be software-pipelined.
- If a loop contains conditional break, it is not software-pipelined.

In our work, we must maximize the number of loops that satisfy the requirements of software pipelining. Software pipelining is a very important technique for optimization, its importance cannot be overemphasized.

4.3.4 Intrinsic [19]

The C6000 compiler provides intrinsics, which are special functions that map directly to inlined C64x instructions, to optimize C/C++ code quickly. All assembly instructions that are not easily expressed in C/C++ code are supported as intrinsics. A table of TMS320C6000 C/C++ compiler intrinsics can be found in [19].

4.4 Performance of the Original Program

Before we enter the detailed description of how to accelerate our implementation, we present the performance of the original program (from [14]). Tables 4.3 and 4.4 show the code size, average cycles per sample of individual function blocks for the transmitter and the receiver, respectively.

In the original program, many blocks contain disqualified software pipelining loops. The compiler feedbacks tell us that many loops contain complex condition code and function calls, such as `fread()` and `fwrite()`. Also, there are some unneeded computation that can be discarded. Detail descriptions are given in later sections.

In our system, the clock frequency of DSP is 600 MHz, and one symbol duration is 201.6 μ s (2304 samples). Therefore, the available execution clock cycles are 120960 in a symbol duration, averaging to 52.5 in a sample duration. To achieve real-time processing speed, one sample must execute less than 52.5 clock cycles. In Tables 4.3 and 4.4, the “multiples of real-time” is defined as the ratio of average cycles count per sample to 52.5. We can see that many blocks cannot operate in real-time rate. In this work, we try to optimize all the blocks as we can.

Table 4.3: Profile of Transmitter Function Blocks

Blocks	Code Size (Bytes)	Avg. Count (Cycles)	Number of Sample Processed	Avg. Cycles per Sample	Multiples of Real-Time
Modulation	616	90252	288	313.38	5.97
Framing	2144	124089	1696	73.17	1.39
IFFT	936	35710	2048	17.44	0.11
TX_SRRC	1624	6199459	2304	2690.74	51.25

Table 4.4: Profile of Receiver Function Blocks

Blocks	Code Size (Bytes)	Avg. Count (Cycles)	Number of Sample Processed	Avg. Cycles per Sample	Multiples of Real-Time
de_framing	2144	124089	1696	73.17	1.39
CP_correlation	756	57	1	57	1.75
Preamble_correlation	1416	8327	1	8327	158.61
SRRC_downsample	564	2302	1	2302	43.85
FFT	276	32247	2048	15.75	0.3

4.5 The Modulation Function

The original version of the modulation software cannot satisfy the requirements of software pipelining due to loop containing control code, as shown in Figure 4.9. Our solution is to reduce the if-else statements and rewrite the modulation schemes into 3 independent files. The first thing we do is construct 3 independent look-up tables that contain the constellation points of their own modulation schemes. Then we extract the modulation schemes from the original function, modulation(). Finally, we let the main function

```

; *-----*
; *   SOFTWARE PIPELINE INFORMATION
; *   Disqualified loop: Loop contains control code
; *-----*

```

Fig. 4.9: Compiler's feedback of the modulation().

```

if (mod_type_user1==0 ||mod_type_user1==1)
{
    modulation_QPSK(TX_IOBuffer1,coded_block_size1,modu_out_user1);
}
if (mod_type_user1==2 ||mod_type_user1==3)
{
    modulation_16QAM(TX_IOBuffer1,coded_block_size1,modu_out_user1);
}
if (mod_type_user1==4 ||mod_type_user1==5)
{
    modulation_64QAM(TX_IOBuffer1,coded_block_size1,modu_out_user1);
}
}

```

Fig. 4.10: A part of C code in the main function.



```

FIXED QAM[4] = {5181,11543,-5181,-11543};
void modulation_16QAM(unsigned char *input, int coded_block_size,short *IQout)
{
    short i,j;
    unsigned int datain;
    unsigned char temp;

    for(j=0;j<coded_block_size/3;j++)
    {
        // load 24 bits into the data_in
        datain=(unsigned int) (input[3*j]);
        datain=(datain<<8)^(unsigned int) (input[3*j+1]);
        datain=(datain<<8)^(unsigned int) (input[3*j+2]);
        datain=datain<<8;

        for(i=0;i<12;i++)
        {
            // Take the rightmost 2 bits of the datain as the input
            temp=(unsigned char) ( (datain&(0xC0000000)) >> 30 );
            // The value of input is then mapped to QAM
            IQout[i+12*j]= QAM[temp];
            datain<<=2;
        }
    }
}

```

Fig. 4.11: C code for modulation_16QAM().

```

/* C code */
// 3 LDBU for loads //
datain=(unsigned int)(input[3*j]);
datain=(datain<<8)^(unsigned int)(input[3*j+1]);
datain=(datain<<8)^(unsigned int)(input[3*j+2]);
=====
/* assembly code */
;* 0      LDBU   .D1T1  *++A7(3),A5      ; |29|
;* 1      NOP
;* 12     LDBU   .D1T1  *+A7(1),A18     ; |29|
;* 13     LDBU   .D1T1  *+A7(2),A8      ; |29|
=====
/* C code */
// 12 STH instructions for stores //
// 12 LDH instructions for loads //
for(i=0;i<12;i++)
{
    temp=(unsigned char)( (datain&(0xC0000000)) >> 30 );
    IQout[i+12*j]= QAM[temp];
    datain<<=2;
}
=====
/* assembly code */
;* 19     LDH    .D1T1  *A16,A3          ; |37|
;*      ||     AND    .L2    6,B7,B7      ; |37|
;*      ||     SHRU   .S2X   A8,5,B21     ; |37|
;*      ||     AND    .S1    6,A9,A9      ; |37|
;* 20     LDH    .D2T1  *B6,A6           ; |37|
;*      ||     SHRU   .S2X   A18,5,B20    ; |37|
;*      ||     AND    .L2    6,B21,B6     ; |37|
;*      ||     ADD    .L1    A19,A9,A9    ; |37|
;* 21     AND    .L2    6,B20,B7         ; |37|
;*      ||     ADD    .S2X   A19,B7,B19   ; |37|
;* 22     LDH    .D2T2  *B19,B9          ; |37|
;*      ||     AND    .L2    6,B17,B17    ; |37|
;*      ||     LDH    .D1T1  *+A19[A3],A8 ; |37|
;*      ||     SHRU   .S2X   A8,3,B19    ; |37|
;* 23     LDH    .D2T1  *B8,A6           ; |37|
;*      ||     ADD    .S2X   A19,B7,B7    ; |37|
;*      ||     AND    .L2    6,B19,B8     ; |37|
;* 24     ADD    .S2X   A19,B8,B8        ; |37|
;*      ||     LDH    .D1T1  *A9,A16      ; |37|
;* 25     ADD    .S2X   A19,B6,B6        ; |37|
;*      ||     LDH    .D1T1  *+A19[A17],A9 ; |37|
;* 26     MV     .S2X   A6,B8            ; |37|
;*      ||     LDH    .D2T2  *B8,B16      ; |37|
;* 27     ADD    .S2X   A19,B17,B6       ; |37|
;*      ||     LDH    .D2T2  *B6,B20      ; |37|
;* 28     LDH    .D2T2  *+B4[B18],B19    ; |37|
;* 29     STH    .D1T1  A3,*+A4(24)      ; |37|
;*      ||     LDH    .D2T2  *B6,B19      ; |37|
;* 30     LDH    .D2T2  *B7,B21          ; |37|
;*      ||     STH    .D1T1  A16,*+A4(20) ; |37|
;* 31     STH    .D2T2  B8,*+B5(24)      ; |37|
;*      ||     STH    .D1T1  A8,*+A4(14)  ; |37|
;* 32     STH    .D1T1  A9,*+A4(22)      ; |37|
;* 33     STH    .D2T2  B19,*+B5(4)      ; |37|
;* 34     STH    .D1T2  B20,*+A4(16)     ; |37|
;*      || [ A0] BDEC   .S1    C950,A0    ; |40|
;* 35     STH    .D2T1  A6,*+B5(2)       ; |37|
;*      ||     STH    .D1T2  B19,*+A4(12) ; |37|
;* 36     NOP
;* 37     STH    .D1T2  B16,*+A4(18)     ; |37|
;* 38     STH    .D2T2  B9,*+B5(8)      ; |37|
;* 39     STH    .D2T2  B21,*+B5(6)     ; |37|
;*      ; BRANCH OCCURS ; |40|

```

Fig. 4.12: A part of the assembly code in the modulation_16QAM().

```

;* SOFTWARE PIPELINE INFORMATION
;*
;* Loop source line           : 31
;* Loop opening brace source line : 32
;* Loop closing brace source line : 63
;* Known Minimum Trip Count    : 12
;* Known Maximum Trip Count    : 12
;* Known Max Trip Count Factor : 12
;* Loop Carried Dependency Bound(^) : 2
;* Unpartitioned Resource Bound : 26
;* Partitioned Resource Bound(*) : 26
;* Resource Partition:
;*
;*           A-side   B-side
;* .L units      0      0
;* .S units      6      13
;* .D units      25     26*
;* .M units      0      0
;* .X cross paths 10     20
;* .T address paths 25   26*
;* Long read paths  0      0
;* Long write paths  0      0
;* Logical ops (.LS)  0      0      (.L or .S unit)
;* Addition ops (.LSD) 26    30     (.L or .S or .D unit)
;* Bound(.L .S .LS)   3      7
;* Bound(.L .S .D .LS .LSD) 19   23
;*
;* Searching for software pipeline schedule at ...
;* ii = 26 Schedule found with 2 iterations in parallel

```

Fig. 4.13: Compiler's feedback of the modulation_QPSK().



```

;* SOFTWARE PIPELINE INFORMATION
;*
;* Loop source line           : 27
;* Loop opening brace source line : 28
;* Loop closing brace source line : 40
;* Known Minimum Trip Count    : 24
;* Known Maximum Trip Count    : 24
;* Known Max Trip Count Factor : 24
;* Loop Carried Dependency Bound(^) : 3
;* Unpartitioned Resource Bound : 14
;* Partitioned Resource Bound(*) : 14
;* Resource Partition:
;*
;*           A-side   B-side
;* .L units      0      0
;* .S units      3      7
;* .D units      14*    13
;* .M units      0      0
;* .X cross paths  0     14*
;* .T address paths 14*   13
;* Long read paths  0      0
;* Long write paths  0      0
;* Logical ops (.LS)  0      0      (.L or .S unit)
;* Addition ops (.LSD)  6     16     (.L or .S or .D unit)
;* Bound(.L .S .LS)   2      4
;* Bound(.L .S .D .LS .LSD) 8     12
;*
;* Searching for software pipeline schedule at ...
;* ii = 14 Schedule found with 3 iterations in parallel

```

Fig. 4.14: Compiler's feedback of the modulation_16QAM().

Table 4.5: Breakdown of Clock Cycles for Three Modulation Functions

One FEC Block	modulation_QPSK	modulation_16QAM	modulation_64QAM
Code Size (bytes)	780	364	280
Max. Cycles	374	412	419
Min. Cycles	374	395	412
Avg. Cycles	374	404	416
Avg. Cycles per Sample	2.59	2.81	2.89

```

;* SOFTWARE PIPELINE INFORMATION
;*
;* Loop source line : 31
;* Loop opening brace source line : 32
;* Loop closing brace source line : 46
;* Known Minimum Trip Count : 36
;* Known Maximum Trip Count : 36
;* Known Max Trip Count Factor : 36
;* Loop Carried Dependency Bound(^) : 2
;* Unpartitioned Resource Bound : 10
;* Partitioned Resource Bound(*) : 10
;* Resource Partition:
;*
;* A-side B-side
;* .L units 0 0
;* .S units 4 5
;* .D units 9 10*
;* .M units 0 0
;* .X cross paths 7 3
;* .T address paths 10* 9
;* Long read paths 0 0
;* Long write paths 0 0
;* Logical ops (.LS) 0 0 (.L or .S unit)
;* Addition ops (.LSD) 10 7 (.L or .S or .D unit)
;* Bound(.L .S .LS) 2 3
;* Bound(.L .S .D .LS .LSD) 8 8
;*
;* Searching for software pipeline schedule at ...
;* ii = 10 Schedule found with 3 iterations in parallel

```

Fig. 4.15: Compiler’s feedback of the modulation_64QAM().

control the program flow, and let the compiler optimize the individual functions, modulation_QPSK(), modulation_16QAM(), and modulation_64QAM(). Figure 4.10 shows the a part of C code in the main function. Figures 4.11 and 4.12 show the C code and its resulting assembly code of modulation_16QAM().

Table 4.5 lists the performance of three different modulation schemes. The size of output data is one FEC block, which has 48×3 samples. Figures 4.13, 4.14 and 4.15 show the compiler’s feedback of the major loop in the codes. The number of iteration intervals (ii) is bounded on the .D functional units. Therefore, if we want to reduce “ii”,

```

void PRBS_pilot(unsigned char wk[212])
{
    unsigned short PRBS, msb_8, PRBS_right_2, i;
    unsigned char temp, j, rev;
    PRBS=0x0555;
    for(i=0;i<212;i++)
    {
        temp=(unsigned char)(PRBS); // reads PRBS
        // bit - reverse for wk
        for ( j=rev=0; j < 8; j++ )
        {
            rev = (rev << 1) | (temp & 1);
            temp >>= 1;
        }
        wk[i]=rev;
        PRBS_right_2 = (PRBS) >> 2;
        msb_8 =(0x00FF)&((PRBS) ^ (PRBS_right_2));
        PRBS = (PRBS >> 8) ^ (msb_8 << 3); //writes PRBS
    }
}

```

Fig. 4.16: C code for original PRBS generator (from [14]).

```

void gen_carrier(int IDcell, unsigned short *carrier_n_s)
{
    unsigned char perbase[32] =
        { 3,18, 2, 8,16,10,11,15,
          26,22, 6, 9,27,20,25, 1,
          29, 7,21, 5,28,31,23,17,
          4,24, 0,13,12,19,14,30};

    int s, n, k;
    for(s = 0; s<Nsubchannels; s++)
    {
        //s: index number of a subchannel
        for(n=0; n<Nsubcarrier; n++)
        {
            k = (n<32) ? 1:2; //ceil(((float)(n+1)/(float)Nsubchannels));
            carrier_n_s[s*Nsubcarrier+n] = Nsubchannels*n + (perbase[(n+ps*s)%Nsubchannels] +
                IDcell*k)%Nsubchannels;
        }
    }
}

```

Fig. 4.17: C code for generating the carrier locations.

we must find a way to deal with data moves more efficiently.

4.6 The Framing and Deframing Functions

In the original code shown in Fig. 4.16, the pseudo random binary sequence (PRBS) generator was for pilot/preamble modulation for the framing function. The polynomial for the PRBS generator is $X^{11} + X^9 + 1$, and the initialization vector is fixed to 0x0555. So we can find that the output is always the same; hence, we can eliminate this function and place the output in the header file for repeated use.


```

/* before optimization */
// UL preamble //
for(s = subchannel_offset; s<(subchannel_offset+No_subchannels); s++)
{
    //s: index number of a subchannel
    for(n=0; n<Nsubcarrier; n++)
    {
        k = (n<32) ? 1:2;
        carrier_n_s = Nsubchannels*n + (perbase[(n+ps*s)%(Nsubchannels)] +
            IDcell*k)%(Nsubchannels);
    }
}




---


/* after optimization */
// UL preamble //
for(s = subchannel_offset; s<(subchannel_offset+No_subchannels); s++)
{
    //s: index number of a subchannel
    for(n=0; n<Nsubcarrier; n++)
    {
        carrier_n_s = carrier_n[s*Nsubcarrier+n];
    }
}

```

Fig. 4.18: Two versions of C program for framing of the preamble.



```

/* before optimization */
// other symbols //
for(n=0,ptr=0;n<Nsubcarrier;n++)// Nsubcarrier = 53
{
    if(n==26 || n==L || n==(L+13) || n==(L+27) || n==(L+40))
    {
        if( ((wk[carrier_n_s/8]<<(carrier_n_s%8))&(0x80))==0x80 )
            frameout[2*carrier_n_s]= -1.33333333*16384;
        else
            frameout[2*carrier_n_s]= 1.33333333*16384;

        frameout[2*carrier_n_s+1]=0;
    }
}



---


/* after optimization */
// other symbols //

unsigned char  L_carrier[Nsubcarrier]; // Nsubcarrier = 53
for(n=0,ptr=0;n<Nsubcarrier;n++)
{
    if(n==26 || n==L || n==(L+13) || n==(L+27) || n==(L+40))
        L_carrier[n] = 1;
    else
        L_carrier[n] = 0;
}
for(n=0,ptr=0;n<Nsubcarrier;n++)
{
    k = (((wk1[carrier_n_s/8]<<(carrier_n_s%8))&(0x80))==0x80) ?-1:1;
    if(L_carrier[n]==1)
    {
        frameout[2*carrier_n_s]= 21845*k;
        frameout[2*carrier_n_s+1]=0;
    }
}

```

Fig. 4.19: Two versions of C program for framing of other symbols.

```

/* C code */
for(n=0,ptr=0;n<Nsubcarrier;n++)//Nsubcarrier = 53
{
  if(n==26 || n==L || n==(L+13) || n==(L+27) || n==(L+40))
    L_carrier[n] = 1; |67|
  else
    L_carrier[n] = 0;
} |71|
=====
/* assembly code */
L62: ; PIPED LOOP KERNEL

      ADD      .D2      1,B7,B7          ; <0,8>
||      OR       .D1X    A3,B9,A0        ; <0,8>
|| [ B0] BDEC     .S2     L62,B0          ; |71| <1,5>
||      MVD      .M1     A6,A3           ; <2,2> Split a long life
||      CMPEQ    .L2X    A16,B5,B9       ; <2,2> ^
||      EXTU     .S1     A17,24,24,A16    ; |71| <2,2> ^
||      CMPEQ    .L1     A16,A5,A6       ; <2,2>

      [!A0] STB     .D2T2  B8,*-B7(1)     ; |67| <0,9>
||      OR       .S2X    B17,A7,B9       ; <1,6>
||      ROTL     .M2     B9,0,B17        ; <2,3> Split a long life
||      OR       .D1X    B16,A8,A7       ; <2,3>
||      CMPEQ    .L1     A16,A9,A8       ; <3,0>

      [ A0] STB     .D2T2  B6,*-B7(1)     ; |67| <0,10>
||      OR       .S1     A6,A7,A7        ; <2,4>
||      CMPEQ    .L1     A16,A4,A6       ; <3,1>
||      ADD      .D1     1,A16,A17       ; |71| <3,1> ^
||      CMPEQ    .L2X    A16,B4,B16      ; <3,1> ^
=====
/* C code */
for(n=0,ptr=0;n<Nsubcarrier;n++)//tung: Nsubcarrier = 53
{
  carrier_n_s = carrier_n[s*Nsubcarrier+n]; |145|
  k = (((wk[carrier_n_s/8]<<(carrier_n_s%8))&(0x80))==0x80) ?-1:1; |146|
  if(L_carrier[n]==1)
  {
    frameout[2*carrier_n_s]= 21845*k; |163|
    frameout[2*carrier_n_s+1]=0; |165|
  }
  else
  {
    frameout[2*carrier_n_s]=readin[ptr+offset];
    frameout[2*carrier_n_s+1]=readin[ptr+1+offset];
    ptr = ptr+2;
  }
} |176|
=====
/* assembly code */
L68: ; PIPED LOOP KERNEL

      CMPEQ    .L2      B24,1,B1          ; <1,8>
||      ADD      .S1X    8,SP,A20         ; |146| <2,0>
||      LDBU     .D2T2   *++B8,B24        ; <2,0>
||      ADD      .L1     1,A20,A23        ; |176| <2,0> ^
||      ADDAH    .D1     A9,A20,A19       ; |145| <2,0>

      [ B0] STH     .D1T1  A3,*+A17(2)     ; |165| <0,17>
|| [!B0] STH     .D2T2   B23,*+B4(2)      ; |165| <0,17>
||      SHL     .S1     A21,A22,A20       ; |146| <0,17>
|| [!B1] ADD     .S2     B7,B17,B22        ; |163| <1,9>
||      MVD      .M1     A20,A18          ; |146| <2,1> Split a long life
||      ADD      .L1     A5,A19,A19       ; |145| <2,1>

      [ B0] ADDAW   .D1     A4,A7,A16       ; |163| <0,18>
|| [ B2] BDEC     .S2     L68,B2          ; |176| <0,18>
||      AND     .L1     A8,A20,A0         ; |146| <0,18>
|| [!B1] ADD     .L2     2,B7,B7          ; |173| <1,10>
|| [!B1] ADDAH   .D2     B5,B22,B20       ; |163| <1,10>
||      EXTU     .S1     A23,24,24,A20    ; |176| <2,2> ^

```

Fig. 4.20: The resulting assembly code for the revised code.

```

; *-----*
; * SOFTWARE PIPELINE INFORMATION          First Loop
; *
; * Loop source line                      : 64
; * Loop opening brace source line       : 65
; * Loop closing brace source line       : 71
; * Known Minimum Trip Count             : 53
; * Known Maximum Trip Count             : 53
; * Known Max Trip Count Factor          : 53
; * Loop Carried Dependency Bound(^)     : 2
; * Unpartitioned Resource Bound         : 3
; * Partitioned Resource Bound(*)        : 3
; * Resource Partition:
; *
; *           A-side   B-side
; * .L units      3*     2
; * .S units      1     1
; * .D units      0     2
; * .M units      0     0
; * .X cross paths 2     3*
; * .T address paths 0     2
; * Long read paths 0     0
; * Long write paths 0     0
; * Logical ops (.LS) 0     0      (.L or .S unit)
; * Addition ops (.LSD) 4     2      (.L or .S or .D unit)
; * Bound(.L .S .LS) 2     2
; * Bound(.L .S .D .LS .LSD) 3*     3*
; *
; * Searching for software pipeline schedule at ...
; *      ii = 3  Schedule found with 4 iterations in parallel
; *-----*
; * SOFTWARE PIPELINE INFORMATION          Second Loop
; *
; * Loop source line                      : 135
; * Loop opening brace source line       : 136
; * Loop closing brace source line       : 177
; * Known Minimum Trip Count             : 53
; * Known Maximum Trip Count             : 53
; * Known Max Trip Count Factor          : 53
; * Loop Carried Dependency Bound(^)     : 2
; * Unpartitioned Resource Bound         : 8
; * Partitioned Resource Bound(*)        : 8
; * Resource Partition:
; *
; *           A-side   B-side
; * .L units      0     1
; * .S units      3     1
; * .D units      8*    7
; * .M units      0     1
; * .X cross paths 1     2
; * .T address paths 3     6
; * Long read paths 0     0
; * Long write paths 0     0
; * Logical ops (.LS) 0     0      (.L or .S unit)
; * Addition ops (.LSD) 5     6      (.L or .S or .D unit)
; * Bound(.L .S .LS) 2     1
; * Bound(.L .S .D .LS .LSD) 6     5
; *
; * Searching for software pipeline schedule at ...
; *      ii = 8  Register is live too long
; *      ii = 8  Schedule found with 3 iterations in parallel

```

Fig. 4.21: Compiler feedback of the optimized code for other symbols.

In the original code for framing, the carrier locations for each user are always re-computed, while these locations are fixed when the transmission parameters are given. Therefore, we can implement an structure that computes the carrier locations of all the subchannels in the beginning of the transmission. Its output is a sequence with a length of 1696 (the number of the data carriers in one OFDM symbol) that contains the carrier locations of subchannels 0 – 31 in order. Figure 4.17 shows the C code that computes the carrier locations. Figure 4.18 shows the original version and the optimized version. This technique can also be used in the deframing program.

Since the framing structure deals with the carrier allocation of the preamble symbol and other UL symbols, we also consider the optimization of the part for other UL symbols. The original program are too complex for the compiler to software pipeline. Therefore, our goal is to reduce the complex code in that part of the program. Figure 4.19 shows the original code and the optimized code. This modification lets the compiler do software pipelining, and the resulting assembly code is shown in Fig. 4.20. And their feedbacks are shown in Figure 4.21. Tables 4.6 and 4.7 compare the original version and the optimized version for framing and deframing, respectively. The original version here is not the same as the code in [14], where the major modification is eliminating the use of fread() and fwrite(). Note that the number of minimum cycles is for preamble and the number of the maximum cycles is for other symbols in both tables. The revised codes are 4.52 and

Table 4.6: Breakdown of Clock Cycles for framing()

4 × OFDMA Symbols 8 Subchannels	Original	Revised
Number of execution	4	4
Code Size (bytes)	1568	1788
Max. Cycles	24472	4828
Min. Cycles	5552	3012
Avg. Cycles	19737	4366
Total Cycles	78950	17466
Cycles per Sample	46.55	10.29

Table 4.7: Breakdown of Clock Cycles for deframing()

4 × OFDMA Symbols 8 Subchannels	Original	Revised
Number of execution	4	4
Code Size (bytes)	696	784
Max. Cycles	17882	6131
Min. Cycles	31	32
Avg. Cycles	13414	4601
Total Cycles	53656	18407
Cycles per Sample	28.91	10.85

2.66 times faster than the original ones, respectively. The first optimized loop in revised version shown in Fig. 4.19 can still be improved, and we leave it to future work.

4.7 The IFFT and FFT Functions

According to IEEE 802.16a, the length of IFFT/FFT is 2048. In the original program [14], it employs the 32-bit fixed-point data type for inputs, outputs and twiddle factors. In order to reduce the computation complexity due to lots of 32×32 multiply operations, we use the 16×16 -bit IFFT function in the TI C64x DSPLIB to replace the original one.

As mentioned in Section 4.3, DSPLIB includes many C-callable, assembly-optimized, general-purpose signal processing routines. These routines are typically used in computation-intensive real-time applications where optimal execution speed is critical. By using these routines, we can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language.

Complex forward mixed radix 16×16 -bit FFT with rounding (DSP_fft16x16r) computes a complex forward mixed radix FFT with scaling, rounding and digit reversal. Input data $x[]$, output data $y[]$ and coefficients $w[]$ are 16-bit. The output is returned in the separate array $y[]$ in normal order. Each complex value is stored as interleaved 16-bit real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors). These twiddle factors are generated by using the program `tw_fft16x16` provided

by TI.

Scaling by 2 (i.e., $\gg 1$) takes place at each radix-4 stage except the last one. A radix-4 stage could give a maximum bit-growth of 2 bits, which would require scaling by 4. To completely prevent overflows, the input data must be scaled by $2^{(BT-BS)}$, where BT (i.e., total number of bit growth) = $\log_2(N)$, BS (i.e., 2's exponent of scaling) = $\text{ceil}[\log_4(N)-1]$, and N is the length of the FFT. All shifts are rounded to reduce truncation noise power by 3 dB.

Note that the DSPLIB does not provide a 16×16 -bit IFFT routine. The IFFT function we perform is just a reuse of the DSP_fft16x16r routine. The IFFT function we implement follows the equation below, with input $x[]$, output $y[]$, and twiddle factor W_N :

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} y[k] W_N^{-kn} = \frac{1}{N} \sum_{k=0}^{N-1} y[k] (W_N^{kn})^* = \frac{1}{N} \left(\sum_{k=0}^{N-1} y[k]^* W_N^{kn} \right)^*, \quad (4.7.1)$$

where $n = 0, \dots, N - 1$. We first conjugate the input, then perform FFT, and conjugate the output again. This IFFT uses the same twiddle factor as the DSP_fft16x16r routine. In the following subsection, we discuss an issue that takes some effort to address in implementing the IFFT.

4.7.1 Analysis of the Output Performance

Since the DSP_fft16x16r routine computes fixed-point numbers, it is recommended by [20] that when using the FFT function, the input data must be scaled by 2^{BT-BS} to completely prevent output overflow. In our case, $BT = \log_2(2048) = 11$, and $BS = \text{ceil}[\log_4(2048)-1] = 5$, so we need to shift right the input by 6 bits. There is a tradeoff between output performance and probability of output overflow. Remember that all shifts can result in quantization errors, the more bits we scale, the more errors we have. As a result, we must try to avoid directly scaling the input by 6 bits (i.e., $\gg 6$).

The IFFT function we implement can be depicted as in Figure 4.22. We note that the number of scaling bits we utilize is not a constant value. The total shift-right bits are 5.

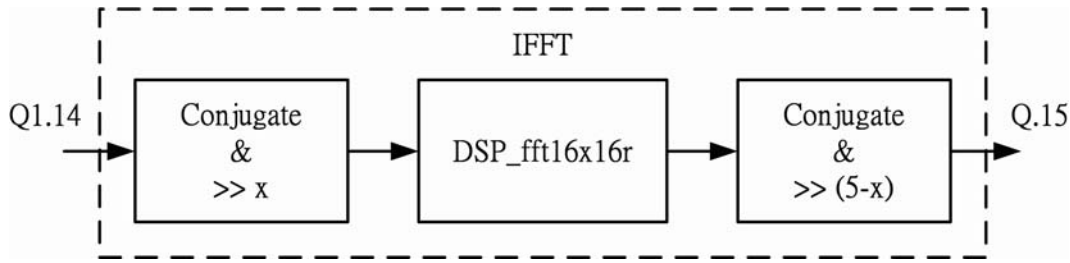


Fig. 4.22: Block diagram of the IFFT function.

This is because by the IFFT factor, $\frac{1}{N} = \frac{1}{2048}$, we need to shift right the output by 11 bits, but since there are 5 “built-in” shift-right bits introduced by DSP_fft16x16r, we only need to shift right 6 bits. Then we shift left the output by 1 bit to obtain the data format Q.15. The reason for adopting a variable number of scaling bits is that different SSs may use a different amount of subchannels, so it is not suitable to set a constant value for all users. We find that, not surprisingly, the more subchannels the SS uses, the more scaling bits it needs.

We simulate use of different number of subchannels. In each simulation, 10000 OFDM symbols are generated. We evaluate the performance of the IFFT output by means of the normalized SNR defined as follows:

- The SNR measures the differences between the output of the IFFT and the output of the original DSP_ifft32x32 routine used in [14].
- Since it is possible to have output overflows, when we calculate the SNR, we discard all the outputs that have overflowed.
- “Normalized” means that the the SNR is divided by the number of subchannels that the SS uses.

From Fig. 4.23, we see that the more subchannels we use, the higher the probability of output overflow is. When $x = 1$, the figure shows that when using more than 20 subchannels, the output always overflows.

Table 4.8: Computational Complexity for FFT algorithm

Complexity	No. of Real Multiplications	No. of Real Additions
Radix-4 FFT	$\frac{9}{8}N \log_2 N - 3N + 3$	$\frac{25}{8}N \log_2 N - 3N + 3$

If the number of subchannels used is fixed, the probability of output overflow decreases with increasing x . The normalized SNR is about 50.6 dB at $x = 1$, 47.6 dB at $x = 2$, and 44.5 dB at $x = 3$. Scaling the input by one more bit can result in 3 dB loss in normalized SNR. Figure 4.24 shows the performance of IFFT when using 64-QAM for modulation.

4.7.2 Complexity Analysis

Theoretically, there are 4 radix-4 and 1 radix-2 stages in DSP_fft16x16r. Table 4.8 gives the computational complexity for radix-4 FFT algorithm. We can find that the number of real multiplications theoretically needed in DSP_fft16x16r are

$$2 \times \left(\underbrace{8451}_{\text{from Table 4.8}} + \underbrace{1024 \times \frac{3}{4}}_{\substack{\text{no. of complex twiddle factors} \\ 1896}} \times \underbrace{2}_{\text{from complex muls}} \right) = 19974,$$

and the real additions needed are

$$2 \times \left(\underbrace{28931}_{\text{from Table 4.8}} + 1024 \times \frac{3}{4} \times \underbrace{4}_{\text{from complex muls}} \right) + \underbrace{2048 \times 2}_{2048 \text{ radix-2 FFT}} = 68102.$$

Since the C6416 DSP chip has 4 16×16 multipliers and 6 32×32 adders executing in parallel, the minimum number of clock cycles needed is given by

$$\max \left[\underbrace{19974}_{\text{multiplications}} \times \frac{1}{4}, \underbrace{68102}_{\text{additions}} \times \frac{1}{6} \right] = 11350.$$

Practically, the time DSP_fft16x16r needed is 15511 clock cycles. We list the complexity and efficiency of DSP_fft16x16 and DSP_fft32x32 (the latter used in [14]) in Table 4.9.

Note that the IFFT implementation includes lots of data moves. Since the former stage is framing, whose output is composed of 1696-point data carriers, we need to move them

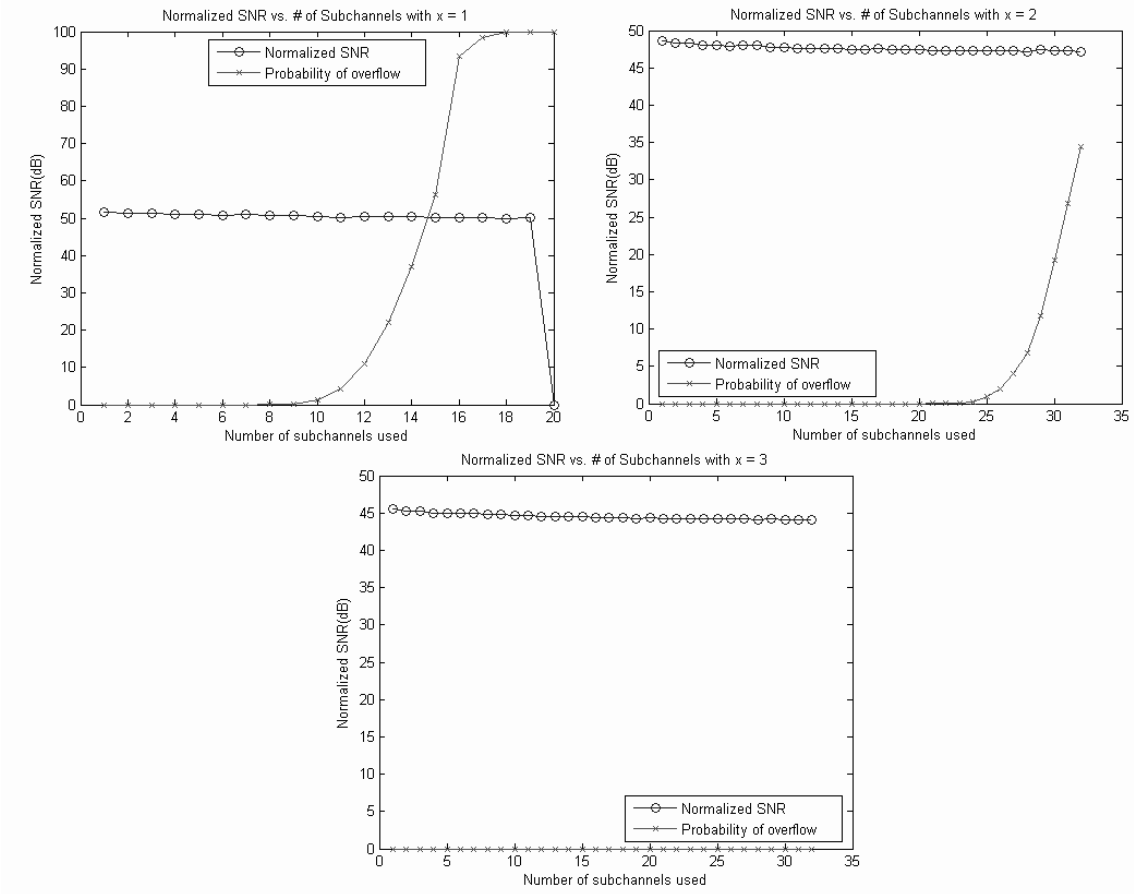


Fig. 4.23: Performance of IFFT when the modulation is 16-QAM.

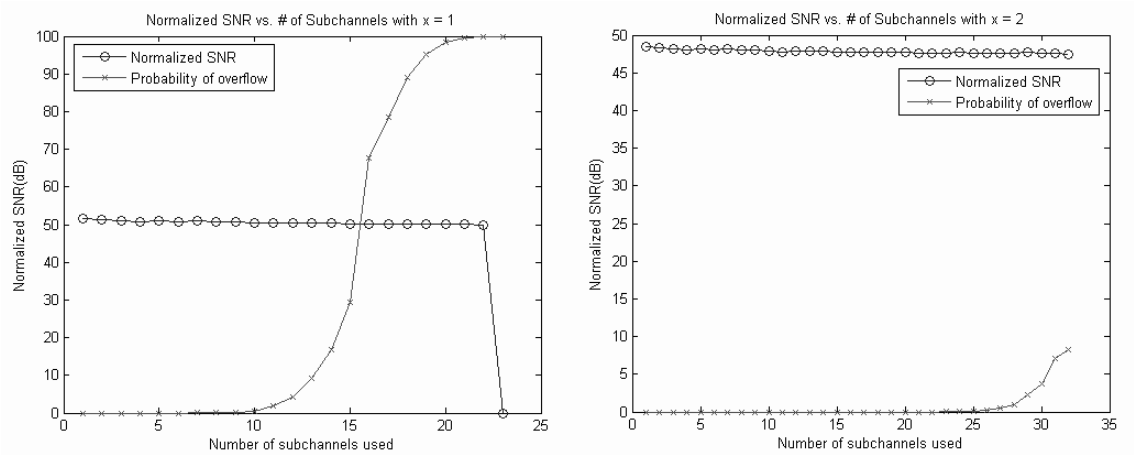


Fig. 4.24: Performance of IFFT when the modulation is 64-QAM.

into their locations in a 2048-point OFDM symbol. Table 4.10 shows a comparison of the IFFT performance for the original version (from [14]) and the revised version.

We give a part of assembly code in the pipeline kernel for DSP_fft16x16r in Fig. 4.25. From this figure, we see that there can be eight instructions executing in parallel. Also we find that this code uses the DOTP2 instruction to execute 16×16 multiplications. This confirms that it can do $4 \cdot 16 \times 16$ multiplications per cycle. Another point is that it uses packet data transmission, through the PACKH2 and PACKLH2 instructions. These instructions are helpful in reducing the number of memory accesses. Thus, the clock cycles can be considerably reduced as the C64x is more efficient in 16-bit multiplies.

```

* ===== PIPE LOOP KERNEL ===== *
LOOP_Y:
PACKH2 .S2   B_x_l1_1,      B_x_l1_0,      B_xl1_1_0      :[24,1]
|| ADD     .L2   B_rnd,      B_x_l2_3,      B_x_l2_3      :[24,1]
|| ADD     .L1   A_rnd,      A_x_h2_3,      A_x_h2_3      :[24,1]
|| DOTP2   .M2X  B_co30_si30,  A_xt2_0_yt2_0, B_x_l2_0      :[14,2]
|| DOTP2   .M1   A_xt1_0_yt1_0, A_co10_si10,  A_x_h2_0      :[14,2]
|| PACKLH2.S1   A_yt2_1_xt1_1, A_yt1_1_xt2_1, A_xt1_1_yt1_1 :[14,2]
|| SUB     .D1   A_fft_jmp,  A_j,          A_ifj         :[ 4,3]
|| LDDW    .D2T2 *B_x[B_l1],  B_xl1_3_xl1_2:B_xl1_1_xl1_0 :[ 4,3]

PACKH2 .L2   B_x_l2_3,      B_x_l2_2,      B_xl2_3_2     :[25,1]
|| PACKH2 .S2   B_x_l2_1,      B_x_l2_0,      B_xl2_1_0     :[25,1]
|| PACKH2 .S1   A_x_h2_3,      A_x_h2_2,      A_xh2_3_2     :[25,1]
|| DOTPN2 .M1   A_yt1_0_xt1_0, A_co10_si10,  A_x_h2_1      :[15,2]
|| PACKLH2.L1  A_yt1_1_xt2_1,  A_yt2_1_xt1_1, A_xt2_1_yt2_1 :[15,2]
|| SUB2    .D2   B_xh1_0_xh0_0, B_xh21_0_xh20_0, B_yt0_0_xt0_0 :[15,2]
|| AVG2    .M2   B_xh21_0_xh20_0, B_xh1_0_xh0_0, B_x_1_x_0     :[15,2]
|| [[!A_p0]STDW .D1T2 B_xl1_3_2:B_xl1_1_0,  *A_x_[A_h2] :[25,1]

ADD     .S1   A_rnd,      A_x_h2_1,      A_x_h2_1      :[26,1]
|| ADD     .L2   B_rnd,      B_x_l1_2,      B_x_l1_2      :[16,2]
|| DOTPN2 .M2X  B_co30_si30,  A_yt2_0_xt2_0, B_x_l2_1      :[16,2]
|| ROTL    .M1   A_xt1_1_yt1_1, 16,          A_yt1_1_xt1_1 :[16,2]
|| PACKLH2.S2   B_yt0_0_xt0_0, B_yt0_0_xt0_0, B_xt0_0_yt0_0 :[16,2]
|| ADD     .L1   A_j,      3,          A_j          :[ 6,3]
|| LDDW    .D1T2 *A_w1[A_j],  B_co21_si21:B_co20_si20 :[ 6,3]
|| LDDW    .D2T1 *B_w0[B_j],  A_co11_si11:A_co10_si10 :[ 6,3]

```

Fig. 4.25: A part of the assembly code in DSP_fft16x16r.

Table 4.9: Complexity and Efficiency of DSP_fft16x16r and DSP_fft32x32

	Needed Number of Clock Cycles	Equivalent Number of Clock Cycles	Efficiency
DSP_fft16x16r	11350	15511	73.18%
DSP_fft32x32	11350	28811	39.40%

Table 4.10: Breakdown of Clock Cycles for IFFT()

Number of Frames = 1 Frame Size = 4×OFDMA Symbols	Original Version	Refined Version
Code size (bytes)	936	1180
Number of execution	4	4
Max. cycles	35710	24149
Min. cycles	35710	24149
Avg. cycles	35710	24149
Total cycles	142840	96596

4.8 Transmission Filtering

In order to provide the ability to simulate path delays at non-integer sample times, an interpolator is induced in the transmitter to yield 4-times oversampled transmitter output. In our system, we adopt the 57-taps SRRC filter with $\alpha = 0.155$. We implement a polyphase system, shown in Fig. 4.26. This implementation would involve applying filter coefficients only to input values that are nonzero. In our work, $L = 4$. When computing an output value at the boundary of a sequence, a portion of the convolution or correlation kernel is usually off the edge of sequence, as illustrated in Fig. 4.27. We assume the values outside the data sequence to be 0, that is, zero padding. Thus, we can avoid using many if-else statements to handle the boundary values when doing convolution.

Table 4.11 shows a comparison of the performance of the original version (from [14]) and the revised version. Note that the refined version can achieve real-time processing, which requires less than 120960 cycles for 2304 samples.

We give a part of C code for convolution the input with $E_0(z)$ and $E_2(z)$ in Fig. 4.28, its compiler's feedback in Fig. 4.29, and a part of the assembly code in the pipeline kernel in Fig. 4.30. From the feedback, we see that the loop can run 8 iterations in parallel with each iteration completed in 5 cycles. From the assembly code, we see that the DOPT2, PCKLH2, and LDNDW instructions are used. The first two we have introduced in last section, the last one is for 8 bytes memory accesses. We know that this is the widest

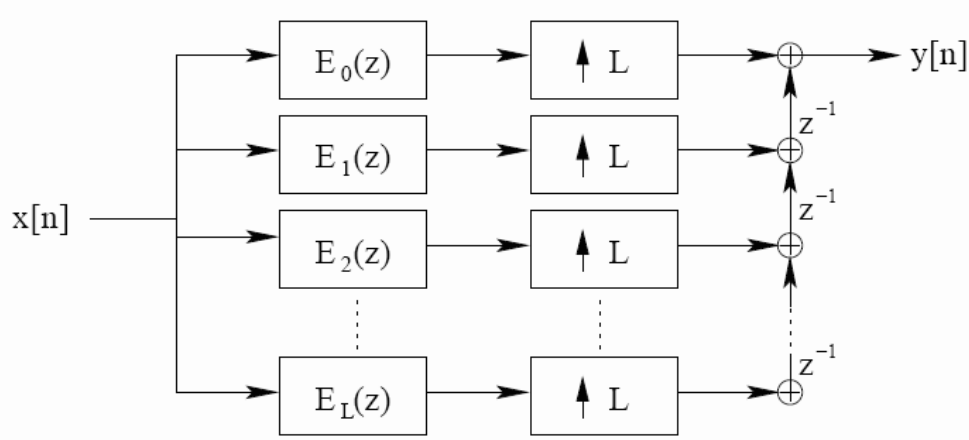


Fig. 4.26: Implementation of interpolation filter with polyphase decomposition [11].

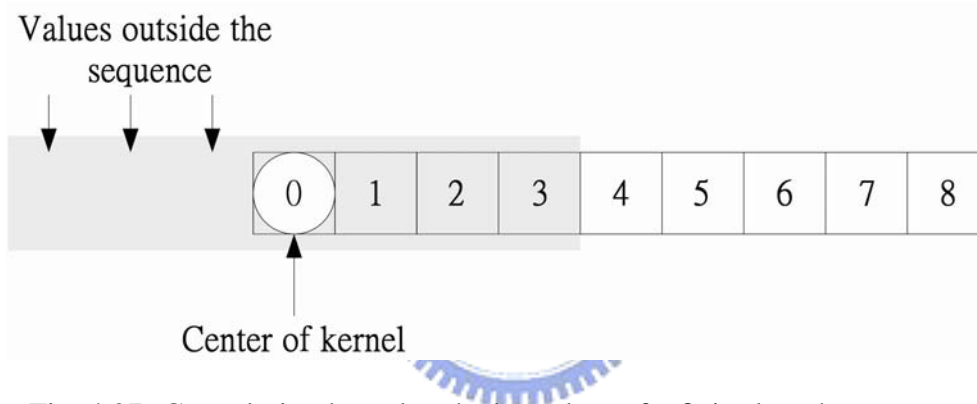


Fig. 4.27: Convolution kernel at the boundary of a finite-length sequence.

memory access provided by the C6416 DSP chip. Although there are not 8 instructions executing in parallel every cycle, the improvement of the performance is impressive.

We also consider replacing the 4 decomposition filters in our revised code by 4 `DSP_fir_gen()` routines from DSPLIB. `DSP_fir_gen()` does real FIR filtering. It operates on 16-bit data with a 32-bit accumulate. From [20], one `DSP_fir_gen()` needs 15675 clock cycles. Since there are 8 real decomposition filters, the total cycles needed are 125400.

After profiling the code, we obtain the performance as listed in Table 4.12. In this table, we do not show the Max and Min cycles, since they are all equal to the average cycles. The inclusive cycles are the cycles for one pass over the profile area, including


```

for(n=7;n<2311;n++)
{
    srFR.buffer_0 = LPF_coefficient_0[14]*temp[n-14+8];
    srFR.buffer_2 = 0;

    for(i=0;i<14;i++)
    {
        srFR.buffer_0 = srFR.buffer_0 + (LPF_coefficient_0[i]*temp[n-i+8]);
        srFR.buffer_2 = srFR.buffer_2 + (LPF_coefficient_2[i]*temp[n-i+8]);
    }
    _symbol[8*(n-7)] = srFR.buffer_0>>15;
    _symbol[8*(n-7)+4] = srFR.buffer_2>>15;
}

```

Fig. 4.28: C code for convolution with $E_0(z)$ and $E_2(z)$.



```

;-----
;*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*   Loop source line           : 107
;*   Loop opening brace source line : 108
;*   Loop closing brace source line : 121
;*   Known Minimum Trip Count    : 2304
;*   Known Maximum Trip Count    : 2304
;*   Known Max Trip Count Factor : 2304
;*   Loop Carried Dependency Bound(^) : 4
;*   Unpartitioned Resource Bound : 8
;*   Partitioned Resource Bound(*)  : 8
;*   Resource Partition:
;*
;*           A-side   B-side
;*   .L units           0       0
;*   .S units           1       2
;*   .D units           7       2
;*   .M units           8*      7
;*   .X cross paths     8*      8*
;*   .T address paths   7       8*
;*   Long read paths    0       0
;*   Long write paths   0       0
;*   Logical ops (.LS)   3       8       (.L or .S unit)
;*   Addition ops (.LSD) 8       8       (.L or .S or .D unit)
;*   Bound(.L .S .LS)   2       5
;*   Bound(.L .S .D .LS .LSD) 7       7
;*
;*   Searching for software pipeline schedule at ...
;*       ii = 8   Schedule found with 5 iterations in parallel

```

Fig. 4.29: Compiler's feedback for convolution with $E_0(z)$ and $E_2(z)$.

```

/* C code */
for(n=7;n<2311;n++)                                     |107|

/* assembly code */
L89:      ; PIPED LOOP EPILOG AND PROLOG
          SUB   .D1   A4,2,A22
||        MVK   .L1   0x1,A2          ; init prolog collapse predicate
||        MV    .D2   B6,B18
||        MVK   .S1   0x900,A0       ; |107|
||        MV    .L2X  A2,B8
||        MVC   .S2   CSR,B2

/* C code */
for(i=0;i<14;i++)
{
  srFR.buffer_0 = srFR.buffer_0 + (LPF_coefficient_0[i]*temp[n-i+8]); |115|
  srFR.buffer_2 = srFR.buffer_2 + (LPF_coefficient_2[i]*temp[n-i+8]); |116|
}
symbol[8*(n-7)] = srFR.buffer_0>>15;                               |118|
symbol[8*(n-7)+4] = srFR.buffer_2>>15;                             |119|

/* assembly code */
L90:      ; PIPED LOOP KERNEL

          ADD   .S2X  A19,B28,B21    ; |116| <0,26>
||        ADD   .S1X  A4,B29,A4      ; |116| <1,18>
||        ADD   .D2   B5,B21,B28     ; |115| <1,18>
||        MPY   .M1   A9,A24,A21     ; |115| <1,18>
||        DOTP2 .M2   B16,B4,B5      ; |116| <2,10>
||        LDNDW .D1T2  *+A22(2),B5:B4 ; |116| <3,2>

          ADD   .S2X  A19,B28,B21    ; |116| <0,26>
||        ADD   .S1X  A4,B29,A4      ; |116| <1,18>
||        ADD   .D2   B5,B21,B28     ; |115| <1,18>
||        MPY   .M1   A9,A24,A21     ; |115| <1,18>
||        DOTP2 .M2   B16,B4,B5      ; |116| <2,10>
||        LDNDW .D1T2  *+A22(2),B5:B4 ; |116| <3,2>

          SHR   .S2   B21,15,B27     ; |119| <0,27>
||        ADD   .D2X  B25,A18,B26    ; |115| <0,27>
||        MPY2  .M1X  A6,B26,A5:A4   ; |115| <1,19>
||        DOTP2 .M2   B18,B27,B30    ; |116| <1,19>
||        LDNDW .D1T2  *+A22(10),B7:B6 ; |116| <3,3>

[ A0]    BDEC   .S1   L90,A0          ; |121| <0,28>
||        ADD   .D2X  A17,B26,B4     ; |115| <0,28>
||        ADD   .L1X  B17,A4,A28     ; |116| <1,20>
||        DOTP2 .M1   A8,A28,A25     ; |115| <1,20>
||        PACKLH2 .S2  B5,B5,B26     ; |115| <2,12>
||        DOTP2 .M2   B9,B31,B17     ; |116| <2,12>
||        PACKLH2 .L2  B4,B4,B5      ; |115| <2,12>
|| [!A2] LDNDW  .D1T1  *A5,A27       ; |115| <3,4>

          DOTP2 .M1X  A16,B30,A4     ; |116| <2,13>
||        PACKLH2 .S2  B6,B6,B31     ; |115| <2,13>
||        DOTP2 .M2   B20,B5,B5      ; |115| <2,13>
||        ADD   .D2X  B0,A26,B21     ; |115| <2,13>
||        LDNDW .D1T2  *-A22(6),B7:B6 ; |115| <3,5>
||        PACKLH2 .S1  A5,A5,A26     ; |115| <3,5>
||        PACKLH2 .L1  A4,A4,A4      ; |115| <3,5>

```

Fig. 4.30: A part of assembly code for convolution with $E_0(z)$ and $E_2(z)$.

Table 4.11: Breakdown of Clock Cycles for TX_SRRC()

Number of Frames = 1 Frame size = 4×OFDMA Symbols	Original version	Revised version
Code size (bytes)	1624	2576
Number of execution	4	4
Max. cycles	6199459	72209
Min. cycles	6199459	72209
Avg. cycles	6199459	72209
Total cycles	24797836	288836

Table 4.12: Breakdown of Clock Cycles for Modified Code using DSP_fir_gen()

Code Size (bytes)	Inclusive Average	Exclusive Average
1856	146372	20956

the execution time (cycle count) of any subroutines called from within the profile area. The exclusive cycles are the cycles spent executing the profile area, excluding the execution time (cycle count) of any subroutines called from within the profile area. After subtracting the inclusive cycles from the exclusive cycles, we can get the cycles spent on the subroutines only. This results in 125416 clock cycles, very close to our calculated, 125400. Table 4.13 summarizes the performance of our optimized versions. The revised version without using DSP_fir_gen() performs the best and achieves real-time processing. This version is 85.85 times faster than the original version.

4.8.1 Complexity Analysis

Each OFDMA symbol has 2304 samples. The SRRC filter is 57-tap, so the number of real multiplications is

$$2304 \times 57 \times \underbrace{2}_{I\text{- and }Q\text{- channels}} = 262656.$$

And the number of real additions is

$$2304 \times 56 \times \underbrace{2}_{I\text{- and }Q\text{- channels}} = 258048.$$

Table 4.13: Breakdown of Clock Cycles for TX_SRRRC()

Number of Frames = 1 Frame size = 4×OFDMA Symbols	Original Version	Revised Version	Revised version with DSP_fir_gen()
Code size (bytes)	1624	2576	1856
Number of execution	4	4	4
Max. cycles	6199459	72209	146372
Min. cycles	6199459	72209	146372
Avg. cycles	6199459	72209	146372
Total cycles	24797836	288836	585488

Table 4.14: Complexity and Efficiency of SRRRC Filter

SRRRC filter	Ideal	Practical	Efficiency
Clock cycles	65664	72209	90.94%

The DSP chip can support four 16×16 multiplications and six 32×32 additions per cycle.

Hence the total cycles we need are at least

$$\max\left[\underbrace{262656}_{\text{multiplications}} \times \frac{1}{4}, \underbrace{258048}_{\text{additions}} \times \frac{1}{6} \right] = 65664.$$

Table 4.14 lists the complexity and efficiency of our revised version without using the DSP_fir_gen() routine.

4.9 The Uplink Synchronization Function

The main operation in the uplink synchronization is preamble correlation. We correlate the received signal with the SS's preamble symbols. Figure 4.31 shows a part of the C code for preamble correlation in sync(). In this loop, Buffer_length equals 2048. This loop performs 2048-point complex multiplication and generates one sample per iteration. Although this loop satisfies the requirements of software pipelining, it can be more efficient if we make some improvements on it. Figure 4.32 shows the code after optimization. The optimization techniques that we utilize are listed below:


```

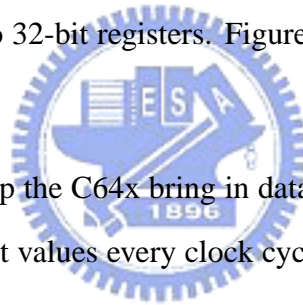
for(i=0;i<Buffer_Length;i++)
{
    CP_real = CP_real    + ref1[i+OFFSET]      * sync_buffer_1_real[i]
                    + ref1[i+2048+OFFSET] * sync_buffer_1_imag[i];
    CP_imag = CP_imag    + ref1[i+2048+OFFSET] * sync_buffer_1_real[i]
                    - ref1[i+OFFSET]      * sync_buffer_1_imag[i];
}

```

Fig. 4.31: C code in sync() before optimization.

- **intrinsics**

- `_hi`: Returns the high 32 bits of a double as an integer.
- `_lo`: Returns the low 32 bits of a double as an integer.
- `_amemd8_const`: Allows aligned loads of 8 bytes to memory.
- `_dotp2`: Returns the dot product between two pairs of signed packed 16-bit values residing in two 32-bit registers. Figure 4.33 illustrates how the `_dotp2` intrinsic operates.



We use the intrinsics to help the C64x bring in data as 64-bit values. Our aim is to let C64x access eight 16-bit values every clock cycle to be able to do four 16×16 multiplies every clock cycle.

- **Loop unrolling**: It expands small loops so that all iterations of the loop appear, and increase the number of instructions available to execute in parallel. There is a speed-space tradeoff. We choose to unroll the loop by 8 times.

Figures 4.34 and 4.35 show the compiler's feedback for the loop. We can see that before optimization, each sample takes 2 cycles ($ii=2$). After optimization, each sample only takes 1 cycle ($ii=8$, and loop-unrolling 8x). Figure 4.36 shows the assembly code in the optimized loop kernel. We see that the instructions used are just the way we want the DSP to do. DOTP2 instructions do 16×16 multiplications and LDW instructions do 8 bytes memory accesses.

```

for(i=0;i<2048;i=i+8)
{
    mul_unit_1 = _hi(_amemd8_const(&ref1[i+2048+OFFSET]));
    mul_unit_2 = _lo(_amemd8_const(&ref1[i+2048+OFFSET]));
    mul_unit_3 = _hi(_amemd8_const(&sync_buffer_1_imag[i]));
    mul_unit_4 = _lo(_amemd8_const(&sync_buffer_1_imag[i]));
    mul_unit_5 = _hi(_amemd8_const(&sync_buffer_1_real[i]));
    mul_unit_6 = _lo(_amemd8_const(&sync_buffer_1_real[i]));
    mul_unit_7 = _hi(_amemd8_const(&ref1[i+OFFSET]));
    mul_unit_8 = _lo(_amemd8_const(&ref1[i+OFFSET]));
    mul_unit_11 = _hi(_amemd8_const(&ref1[i+4+2048+OFFSET]));
    mul_unit_12 = _lo(_amemd8_const(&ref1[i+4+2048+OFFSET]));
    mul_unit_13 = _hi(_amemd8_const(&sync_buffer_1_imag[i+4]));
    mul_unit_14 = _lo(_amemd8_const(&sync_buffer_1_imag[i+4]));
    mul_unit_15 = _hi(_amemd8_const(&sync_buffer_1_real[i+4]));
    mul_unit_16 = _lo(_amemd8_const(&sync_buffer_1_real[i+4]));
    mul_unit_17 = _hi(_amemd8_const(&ref1[i+4+OFFSET]));
    mul_unit_18 = _lo(_amemd8_const(&ref1[i+4+OFFSET]));

    CP_real = CP_real + _dotp2(mul_unit_1 , mul_unit_3) + _dotp2(mul_unit_2 , mul_unit_4)
    + _dotp2(mul_unit_7 , mul_unit_5) + _dotp2(mul_unit_8 , mul_unit_6)
    + _dotp2(mul_unit_11 , mul_unit_13) + _dotp2(mul_unit_12 , mul_unit_14)
    + _dotp2(mul_unit_17 , mul_unit_15) + _dotp2(mul_unit_18 , mul_unit_16);
    CP_imag = CP_imag + _dotp2(mul_unit_1 , mul_unit_5) + _dotp2(mul_unit_2 , mul_unit_6)
    - _dotp2(mul_unit_7 , mul_unit_3) - _dotp2(mul_unit_8 , mul_unit_4)
    + _dotp2(mul_unit_11 , mul_unit_15) + _dotp2(mul_unit_12 , mul_unit_16)
    - _dotp2(mul_unit_17 , mul_unit_13) - _dotp2(mul_unit_18 , mul_unit_14);
}

```

Fig. 4.32: C code in sync() after optimization.

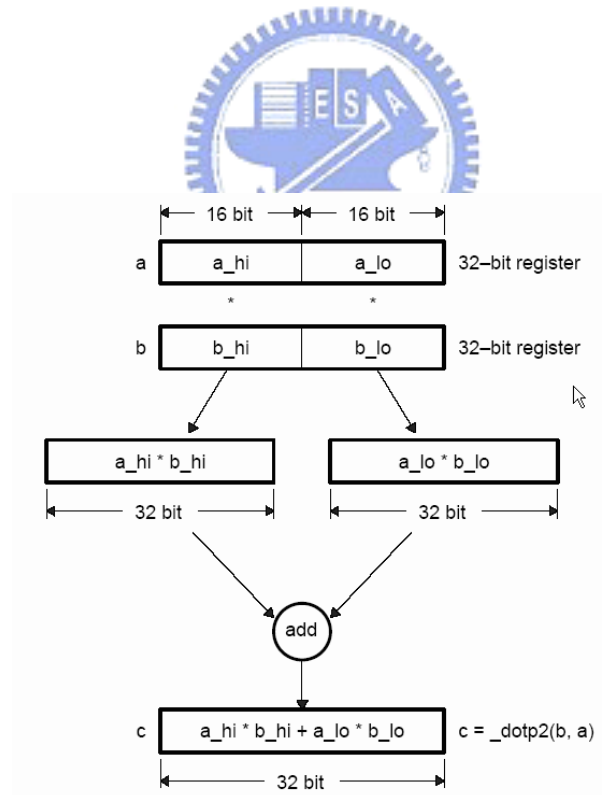


Fig. 4.33: Graphical illustration of $c = \text{_dotp2}(b, a)$ [19].

Table 4.15: Breakdown of Clock Cycles for sync()

Number of Frames = 1 Frame size = 4×OFDMA Symbols	Original Version	Revised Version
Code size (bytes)	716	1020
Number of execution	205	205
Max. cycles	8331	4770
Min. cycles	8331	4770
Avg. cycles	8331	4770
Total cycles	1707855	977850

Table 4.15 compares the performance of the original code and the revised code. The revised version is 1.75 times faster than the original one.

4.9.1 Complexity Analysis

Figures 2.14 and 2.15 have shown the structure of preamble correlation and the number of executions, respectively. We only do uplink synchronization at the first symbol of one frame.

For each user, the real multiplications we need are

$$2048 \times \underbrace{4}_{\text{complex multiplication}} \times \underbrace{205}_{\text{range}} = 1679360.$$

The real additions we need are

$$2048 \times \underbrace{2}_{\text{from complex multiplication}} \times \underbrace{205}_{\text{range}} + 2047 \times \underbrace{2}_{\text{complex addition}} \times \underbrace{205}_{\text{range}} = 1678950.$$

In our work, the number of users is 2. We need to do preamble correlation for each user, then we can get the peak locations (i.e., arriving timing positions) of them. After knowing their peak locations, we start comparing and decide which SS is first coming. The total multiplications per frame are

$$1679360 \times 2 = 3358720.$$

The total additions per frame are

$$1678950 \times 2 = 3357900.$$

```

;* SOFTWARE PIPELINE INFORMATION
;*
;* Loop source line           : 68
;* Loop opening brace source line : 69
;* Loop closing brace source line : 74
;* Known Minimum Trip Count    : 2048
;* Known Maximum Trip Count    : 2048
;* Known Max Trip Count Factor : 2048
;* Loop Carried Dependency Bound(^) : 1
;* Unpartitioned Resource Bound : 2
;* Partitioned Resource Bound(*) : 2
;* Resource Partition:
;*
;*           A-side   B-side
;* .L units      0      0
;* .S units      1      0
;* .D units      2*    2*
;* .M units      2*    2*
;* .X cross paths 1      1
;* .T address paths 2*   2*
;* Long read paths 0      0
;* Long write paths 0     0
;* Logical ops (.LS) 0     0      (.L or .S unit)
;* Addition ops (.LSD) 2    2      (.L or .S or .D unit)
;* Bound(.L .S .LS) 1     0
;* Bound(.L .S .D .LS .LSD) 2*  2*
;*
;* Searching for software pipeline schedule at ...
;*      ii = 2  Schedule found with 5 iterations in parallel

```

Fig. 4.34: Compiler's feedback of the code shown in Fig. 4.31.



```

;* SOFTWARE PIPELINE INFORMATION
;*
;* Loop source line           : 38
;* Loop opening brace source line : 39
;* Loop closing brace source line : 65
;* Known Minimum Trip Count    : 256
;* Known Maximum Trip Count    : 256
;* Known Max Trip Count Factor : 256
;* Loop Carried Dependency Bound(^) : 1
;* Unpartitioned Resource Bound : 8
;* Partitioned Resource Bound(*) : 8
;* Resource Partition:
;*
;*           A-side   B-side
;* .L units      0      0
;* .S units      1      0
;* .D units      4      4
;* .M units      8*    8*
;* .X cross paths 8*    8*
;* .T address paths 4     4
;* Long read paths 0      0
;* Long write paths 0     0
;* Logical ops (.LS) 0     0      (.L or .S unit)
;* Addition ops (.LSD) 8    8      (.L or .S or .D unit)
;* Bound(.L .S .LS) 1     0
;* Bound(.L .S .D .LS .LSD) 5    4
;*
;* Searching for software pipeline schedule at ...
;*      ii = 8  Schedule found with 3 iterations in parallel

```

Fig. 4.35: Compiler's feedback of the code shown in Fig. 4.32.

```

/* C code */
for(i=0;i<2048;i=i+8) |38|
{
  CP_real = CP_real + _dotp2(mul_unit_1 , mul_unit_3) + _dotp2(mul_unit_2 , mul_unit_4) |51|
              + _dotp2(mul_unit_7 , mul_unit_5) + _dotp2(mul_unit_8 , mul_unit_6)
              + _dotp2(mul_unit_11 , mul_unit_13) + _dotp2(mul_unit_12 , mul_unit_14)
              + _dotp2(mul_unit_17 , mul_unit_15) + _dotp2(mul_unit_18 , mul_unit_16);
  CP_imag = CP_imag + _dotp2(mul_unit_1 , mul_unit_5) + _dotp2(mul_unit_2 , mul_unit_6) |61|
                 - _dotp2(mul_unit_7 , mul_unit_3) - _dotp2(mul_unit_8 , mul_unit_4)
                 + _dotp2(mul_unit_11 , mul_unit_15) + _dotp2(mul_unit_12 , mul_unit_16)
                 - _dotp2(mul_unit_17 , mul_unit_13) - _dotp2(mul_unit_18 , mul_unit_14);
}
=====
/* assembly code */
L12:      ; PIPED LOOP PROLOG

          ZERO   .S2   B23           ; |38|
||        ZERO   .L2   B19           ; |38|
||        ZERO   .D1   A3            ; |38|
||        MVKLE  .S1   _sync_buffer_1_real,A21
||        LDDW   .D2T2 *B18++(16),B7:B6 ; |57| (P) <0,0> ^
||        MV     .L1X  B8,A19

          ZERO   .S2   B22           ; |38|
||        ZERO   .L2   B25           ; |38|
||        ZERO   .L1   A18           ; |38|
||        MVKHE  .S1   _sync_buffer_1_real,A21
||        LDDW   .D1T1 *A19++(16),A9:A8 ; |57| (P) <0,1> ^
||        LDDW   .D2T2 *-B18(8),B17:B16 ; |57| (P) <0,1> ^
L13:      ; PIPED LOOP KERNEL
[!A1]    ADD     .S2   B6,B26,B26     ; |57| <0,10> ^
|| [!A1]    ADD     .S1   A9,A22,A22   ; |57| <0,10> ^
||          DOTP2  .M2X  B17,A7,B4    ; |61| <0,10> ^
||          DOTP2  .M1X  B8,A8,A4    ; |61| <0,10> ^
||          LDDW   .D1T1 *A21++(16),A17:A16 ; |57| <1,2> ^
||          LDDW   .D2T2 *B20++(16),B9:B8 ; |57| <1,2> ^

[!A1]    ADD     .S2   B6,B23,B23     ; |61| <0,11> ^
|| [!A1]    ADD     .S1   A8,A26,A26   ; |61| <0,11> ^
||          DOTP2  .M2X  B5,A7,B4    ; |57| <0,11> ^
||          DOTP2  .M1X  B16,A6,A4   ; |61| <0,11> ^
||          LDDW   .D1T1 *-A19(8),A5:A4 ; |57| <1,3> ^
||          LDDW   .D2T2 *-B20(8),B5:B4 ; |57| <1,3> ^

[!A1]    ADD     .L1   A7,A20,A20     ; |57| <0,12> ^
|| [!A1]    SUB     .D2   B21,B5,B21   ; |61| <0,12> ^
||          DOTP2  .M2X  B9,A17,B6    ; |57| <0,12> ^
|| [ A0]    BDEC   .S1   L13,A0       ; |65| <0,12> ^
||          DOTP2  .M1X  B4,A6,A6    ; |57| <0,12> ^
||          LDDW   .D1T1 *-A21(8),A7:A6 ; |57| <1,4> ^

```

Fig. 4.36: A part of the assembly code in sync().

Hence, the total clock cycles we need are at least

$$\max\left[\underbrace{3358720}_{\text{multiplications}} \times \frac{1}{4}, \underbrace{3357900}_{\text{additions}} \times \frac{1}{6}\right] = 839680.$$

We list the complexity and Efficiency of sync() in Table 4.16. Note that the complexity of sync() depends on the number of correlators so that it is linearly proportional to the number of SSs.

Table 4.16: Complexity and Efficiency of sync()

SRRC filter	Ideal	Practical	Efficiency
Clock cycles	839680	977850	85.87%

4.10 Conclusion in Optimization

In this section, we give a conclusion in optimization. Tables 4.17 and 4.18 give the performance of the optimized DSP code in the transmitter and the receiver, respectively. In our system, the clock frequency of the DSP is 600 MHz, and one symbol duration is 201.6 μ s (2304 samples). Therefore, the execution clock cycles is 120960 in a symbol duration and average 52.5 in a sample duration. To achieve real-time processing speed, one sample must consume no more than 52.5 clock cycles. The “multiples of real-time” is defined as the consumed cycles per sample divided by 52.5.

In Table 4.17, we show the transmitter function. We have only done optimizations to the functions from [14]. They are modulation, framing, TX_SRRC, and IFFT. The FEC functions [15] are also integrated to our system, but only QPSK is functioning at the moment. The FEC decoder for the 16-QAM is not workable, so the 16QAM option does not work normally yet. Except for some of the FEC functions, the other functions we have optimized can operate in real-time individually. The performance of modulation and framing are limited to the number of data moves. If we want to further accelerate them, we may consider rewriting them in linear assembly. The computational efficiencies of TX_SRRC and IFFT are 90.94% and 73.18%, respectively, as already discussed previously. The enc_main_first and enc_main_second are for FEC encoding, and they are for coding rates $\frac{1}{2}$ and $\frac{3}{4}$ in QPSK, respectively. They need the computing power of at least 5 C6416 chips to achieve real-time, at this moment.

Table 4.18 gives the performance of the functions in the receiver. The deframing and deinter_QPSK can achieve real-time rate. In the deframing function, there are still many data moves instructions. So rewriting it in linear assembly is also a way to improve it. The

Table 4.17: Profile of 802.16a UL Transmitter Function Blocks

Function	Code Size (Bytes)	Avg. Cycles per sample	Improvements (%)	Multiples of Real-Time	Fraction of Total
modulation_QPSK	780	2.6	61.63	0.045	4.87%
modulation_16QAM	364	2.84	95.38	0.054	
modulation_64QAM	280	2.88	96.11	0.055	
framing	1812	11.04	77.89	0.21	19.39%
TX_SRRC	2576	31.34	99.89	0.60	55.03%
IFFT	1180	11.79	46.15	0.22	20.70%
enc_main_first	1732	164.87	none	3.14	Excluded
enc_main_second	2124	230.10	none	4.38	
inter_QPSK	244	26.13	none	0.50	

Table 4.18: Profile of 802.16a UL Receiver Function Blocks

Function	Code Size (Bytes)	Avg. Cycles per Sample	Multiples of Real-Time	Fraction of Total
de_framing	784	10.85	0.21	1.82%
sync	1020	4770	90.86	Excluded
channel equalizer	1424	178.18	3.39	41.63%
RX_SRRC	400	242	4.61	56.55%
dec_main_0	2340	883.92	16.84	Excluded
dec_main_1	2244	1203.66	22.93	
deinter_QPSK	216	17.67	0.34	

sync function cannot achieve real-time yet, since one sample needs 4770 cycles. It needs at least 91 C6416 chips to process in real-time rate. But it only needed at the start of a frame. And the computational efficiency of it is 85.87%. If the number of SSs is increased, the number of cycles needed will also be increased. The RX_SRRC is optimized in [25]. But it needs at least 5 C6416 chips to achieve real-time rate. The channel equalizer needs at least 4 C6416 chips to achieve real-time rate. We can do some modifications to it for further improvements. The dec_main_0 and dec_main_1 are used for FEC decoder, and they are for coding rates $\frac{1}{2}$ and $\frac{3}{4}$ in QPSK, respectively. They need at least 23 C6416 chips for real-time processing.

In this thesis, we do not analyze the SNR performance of the SRRC function and the sync function, neither. This shall be done in the future work.



Chapter 5

Conclusion and Future work

5.1 Conclusion

In this thesis, the implementation of TDD OFDMA uplink system on TI's C6416 digital signal processor has been introduced. The implementation was based on the codes from [14] and [15], which dealt with uplink synchronization and FEC encoder/decoder, respectively. We rewrote and integrated those codes into a version that was friendly in block transmission mechanism, which is the method for communication between the host PC and the DSP baseboard. But due to the unknown system software bug, we are unable to run the channel equalizer() on the DSP baseboard yet.

Another part of this thesis was to introduce the optimization techniques in order to accelerate the blocks in our work. We have optimized the modulation, framing/deframing, TX_SRRRC, FFT/IFFT, and sync blocks. Most of them can achieve real-time rate, but the sync cannot. The computational efficiency was also discussed in this thesis. We computed the ideal complexity needed, and compared it with the practical complexity. The results can remind us that if there still has rooms for further improvements. The efficiencies of TX_SRRRC, FFT, and sync are 90.94%, 73.18%, and 85.87%, respectively.

The UL synchronization we proposed use the preamble correlation to obtain the symbol arriving time instant. Since the values of the preamble of each SS are known by the BS, it can be used as the reference to correlate the received signals. By using this method, we can find the precise timing of the first coming SS. The timing errors are in some degree

to correlated to the channel model. The results are similar to [11].

5.2 Potential Future work

In the realized system, we find that there are several possible extensions that would enhance the capability and performance.

- The implementation of overall functions in the receiver on DSP should be workable. The memory allocation problem should be solved.
- The ideal channel equalizer shall be replaced by other practical algorithms.
- The BS shall compute the BER and SNR of each user and tune their used subchannels to approach the real conditions.
- Strictly speaking, the CPU busmastering interface is not efficient for the DSP baseboard communicating with the host PC. We shall consider the streaming interface, another mechanism provided by the Quixote.
- Since there are still many blocks not satisfying the real-time requirement, we can try to partition them on FPGA or other DSP boards.
- Since the communications among DSP baseboards are not applicable in our system, we can not process couples of DSP boards in parallel. If this functionality can be applied, we can consider connecting the source encoder/decoder to our system.
- The integration of other modulation choice (i.e., 16-QAM) in FEC encoder/decoder are not workable yet.
- Analyze the SNR performance of the sync function.

Bibliography

- [1] IEEE Std 802.16a-2003, *IEEE Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Fixed Broadband Wireless Access Systems — Amendment 2: Medium Access Control Modifications and Additional Physical Layer Specifications for 2–11 GHz*. New York: IEEE, April 2003.
- [2] H. Sari and G. Karam, “Orthogonal frequency-division multiple access and its application to CATV networks,” *Eur. Trans. Telecommun.*, vol. 9, pp. 507–516, Dec. 1998.
- [3] <http://standards.ieee.org/announcements/80216abwa.html>.
- [4] A. Ghosh, D. R. Wolter, J. G. Andrews, and R. Chen, “Broadband wireless access with WiMax/802.16: current performance benchmarks and future potential,” *IEEE Commun. Mag.*, vol. 43, pp. 129–136, Feb. 2005.
- [5] IEEE Std 802.16.-2004, *IEEE Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Fixed Broadband Wireless Access Systems*. New York: IEEE, June 2004.
- [6] R. van Nee and R. Prasad, *OFDM for Wireless Multimedia Communications*. Boston: Artech House, 2000.
- [7] S. B. Weinstein and P. M. Elbert, “Data transmission by frequency-division multiplexing using the discrete Fourier transform,” *IEEE Trans. Commun. Technol.*, vol. COM-19, pp. 628–634, Oct. 1971.

- [8] WiMAX Forum, "IEEE 802.16a standard and WiMAX igniting broadband wireless access," White Paper, Sep. 2003. Available at <http://www.wimaxforum.org/news/downloads/WiMAXWhitepaper.pdf>.
- [9] http://66.91.152.34:8080/library/proceedings/ptc2003/program/private/wednesday/w13/w134_fotheringham.pdf
- [10] C. Eklund, R. B. Marks, K. L. Stanwood, and S. Wang, "IEEE standard 802.16: a technical overview of the WirelessMAN air interface for broadband wireless access," *IEEE Commun. Mag.*, vol. 40, pp. 98–107, June 2002.
- [11] M.-T. Lin, "Fixed and mobile wireless communication based on IEEE 802.16a TDD OFDMA: Transmission filtering and synchronization," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2003.
- [12] T. Keller, L. Piazzo, P. Mandarini, and L. Hanzo, "Orthogonal frequency division multiplex synchronization techniques for frequency-selective fading channels," *IEEE T. Select. Areas Commun.*, vol. 19, no. 6, pp. 999–1008, June 2001.
- [13] B. Yang, K. B. Letaief, R. S. Cheng, and Z. Cao, "Timing recovery for OFDM transmission," *IEEE T. Select. Areas Commun.*, vol. 18, no. 11, pp. 2278–2291, Nov. 2000.
- [14] H.-C. Lin, "Study and DSP implementation of IEEE 802.6a TDD OFDMA up-link synchronization," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2004.
- [15] Y.-T. Lee, "DSP implementation and optimization of the forward error correction scheme in IEEE 802.16a standard," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2004.

- [16] Innovative Integration, *Quixote User's Manual*. Dec. 2003.
- [17] Texas Instruments, *TMS320C6414T, TMS320C6415T, TMS320C6416T Fixed-Point Digital Signal Processors*. Literature no. SPRS226A, Mar. 2004.
- [18] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*. Literature no. SPRU189F, Oct. 2000.
- [19] Texas Instruments, *TMS320C6000 Programmer's Guide*. Literature no. SPRU198G, Aug. 2002.
- [20] Texas Instruments, *TMS320C64x DSP Library Programmer's Reference*. Literature no. SPRU565B, Oct. 2003.
- [21] Texas Instruments, *Code Composer Studio User's Guide*. Literature no. SPRU328B, Feb. 2000.
- [22] Texas Instrument, *TMS320C6000 Optimizing Compiler User Guild*. Literature no. SPRU187K, Oct. 2002.
- [23] P. Dent, G. E. Bottomley, and T. Croft, "Jakes' fading model revisited," *Electron. Lett.* vol. 29, no. 13, pp. 1162–1163, June 1993.
- [24] <http://www.innovative-dsp.com/products/quixote.htm>.
- [25] Y.-S. Chen, "DSP software implementation and integration of IEEE 802.16a TDD OFDMA downlink transceiver system," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2005.

作者簡歷

董景中，民國七十年一月出生於台南市。民國九十二年六月畢業於國立清華大學工程與系統科學系，並於同年九月進入國立交通大學電子研究所就讀，從事通訊系統方面相關研究。民國九十四年六月取得碩士學位，碩士論文題目為『IEEE 802.16a 分時雙工正交分頻多重進接上行傳收系統在數位訊號處理器平台上之整合及最佳化』。研究範圍與興趣包括：信號處理、通訊系統及隨機程序。

