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 802.16e OFDMA 上行及下行通道估測技術之探討與數位訊號處理器實現

Research in and DSP Implementation of Channel Estimation Techniques for IEEE 802.16e OFDMA Uplink and Downlink

研 究 生：王依翎

指導教授：林大衛 博士

中 華 民 國 九 十 六 年 六 月

IEEE 802.16e OFDMA 上行及下行通道估測技術之探討

與數位訊號處理器實現

Research in and DSP Implementation of Channel Estimation

Techniques for IEEE 802.16e OFDMA Uplink and Downlink

研究生：王依翎 Student: Yi-Ling Wang

指導教授：林大衛 博士 Advisor: Dr. David W. Lin

國立交通大學

電子工程學系 電子研究所碩士班

碩士論文

A Thesis
Submitted to Department of Electronics Engineering & Institute of Electronics
College of Electrical and Computer Engineering
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of Master
in
Electronics Engineering
June 2007
Hsinchu, Taiwan, Republic of China

中華民國九十六年六月

IEEE 802.16e OFDMA 上行及下行通道估測技術之探討與數位訊號處理器實現

研究生：王依翎　　　　　　　　指導教授：林大衛 博士

國立交通大學

電子工程學系　電子研究所碩士班

摘要

正交分頻多重進接(OFDMA)技術近來在行動環境中廣受注目且已經應用在許多數位通訊應用中。採用 OFDMA 一個最主要的原因是其抗頻率選擇性衰變及窄頻干擾的能力。我們聚焦在 IEEE 802.16e OFDMA 上行及下行傳輸的通道估測部分。我們並在 Sundance 公司的版上裝置德州儀器公司的 TMS320C6416 數位信號處理器來實現通道估測的機制。

通道估測大致可以分成三個階段。首先我們使用最小平方差的估測器來估計在導訊上的通道頻率響應，這是為了硬體的計算方便。其次我們在頻率域上使用線性內插法來得到在資料載波上的通道響應。最後我們使用平均時間技巧在時域上來增進其效能。我們先在 AWGN 通道上驗證我們的模擬模型，然後再放置於多重路徑的 SUI-2 和 SUI-3 通道上模擬。

在上行傳輸，我們提出了瓦線性內插法;而在下行傳輸，我們有提出了 2 點、4 點以及進階 4 點群線性內插法。為了增進程式在數位訊號處理器上的執行效率，我們先將原始的浮點運算 C 程式版本修改為實數運算的程式版本，接著再

考慮數位訊號處理器的特性來修改之前的程式

在本篇論文中，我們首先簡介 IEEE 802.16e OFDMA 上行及下行的標準機制和 DSP 的實現環境。接著，我們分別在各傳輸情形下介紹所用的通道估測方法並探討其估測效能及數位訊號處理器實現方面的實驗結果。

Research in and DSP Implementation of Channel Estimation Techniques for IEEE 802.16e OFDMA Uplink and Downlink

Student：Yi-Ling Wang　　　　Advisor：Dr. David W. Lin

Department of Electronics Engineering

Institute of Electronics

National Chiao Tung University

Abstract

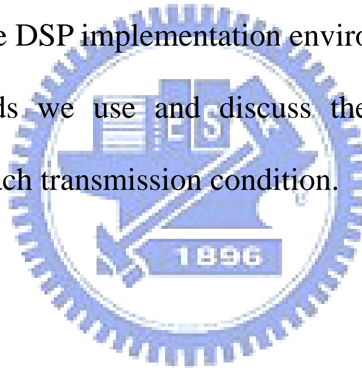
OFDMA (orthogonal frequency division multiple access) technique has drawn much interest recently in the mobile transmission environment and been successfully applied to a wide variety of digital communications applications over the past several years. One of the main reason to use OFDMA is its robustness against frequency selective fading and narrowband interference. We focus on the OFDMA uplink and downlink channel estimation based on IEEE 802.16e. We also implement these channel estimation schemes on Texas Instruments' TMS320C6416 digital signal processor (DSP) housed on Sundance board.

The channel estimation schemes can be separated into three steps. First, we use LS estimator on pilot subcarriers because of its low computational complexity. Second, we estimate the channel response on data subcarriers using linear

interpolation in the frequency domain. Finally we try time averaging technique to improve the performance in the time domain. We verify our simulation model on AWGN channel and then do the simulation on SUI-2 and SUI-3 multipath channels.

In uplink transmission, we propose the tile linear interpolation and as for downlink, we use the 2-point, 4-point and advanced 4-point cluster linear interpolation. In order to increase the efficiency on DSP, we rewrite the floating-point C program to fixed-point version and further refine our codes by considering the features of the DSP chip.

In this thesis, we first introduce the standard of the IEEE 802.16e OFDMA uplink and downlink and the DSP implementation environment . Then we describe the channel estimation methods we use and discuss the performance and the DSP implementation results in each transmission condition.



誌謝

這篇論文能夠順利完成，要感謝的人很多，首先要最感謝我的指導教授林大衛老師，感謝他兩年來在教學上對我的指導與包容，在遭遇困難時老師總是能細心地給予適當的方向去解決問題，能成為老師的學生真的是我前世修來的福氣及畢生最大的榮幸。

此外，由衷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。特別感謝洪崑健學長、吳俊榮學長對我在學業研究上的不吝指導與建議，還有耀鈞、柏昇、政達、介遠、順成、、等同學，謝謝他們這兩年來對我的照顧以及幫助。

最後更要感謝我的父母親，家人對我的支持、鼓勵是我求學路上精神的最大慰藉，對他們的感謝是筆墨難以形容的。

最後由衷感謝所有幫助關懷過我的人。

王依翎

民國九十六年七月 於新竹

Contents

1 Introduction **1**

2 Introduction to IEEE802.16e OFDMA **3**

2.1 Overview of OFDMA [3], [4] 3

2.1.1 Cyclic Prefix 4

2.1.2 Discrete Time Baseband Equivalent System Model 5

2.2 Basic OFDMA Symbol Structure in IEEE 802.16e 6

2.2.1 OFDMA Basic Terms 7

2.2.2 Frequency Domain Description 7

2.2.3 Primitive Parameters 8

2.2.4 Derived Parameters 9

2.2.5 Frame Structure 9

2.3 Uplink Transmission in IEEE 802.16e OFDMA 10

2.3.1 Data Mapping Rules 10

2.3.2 Carrier Allocations 11

2.3.3 Pilot Modulation 15

2.3.4 Data Modulation 15

2.4 Downlink Transmission in IEEE 802.16e OFDMA 16

2.4.1 Data Mapping Rules 16

2.4.2 Preamble Structure and Modulation 17

2.4.3 Subcarrier Allocations 19

2.4.4 Pilot Modulation 22

2.4.5 Data Modulation 22

3 The DSP Hardware and Associated Software Development Environment 23

3.1 The TMS320C6416 DSP [7] 23

3.1.1 TMS320c64x Features 23

3.1.2 Central Processing Unit 25

3.1.3 Memory Architecture 29

3.2 The Code Composer Studio Development Tools [9], [10] 33

3.3 Code Optimization Methods [12] 35

3.3.1 Compiler Optimization Options [9], [10] 37

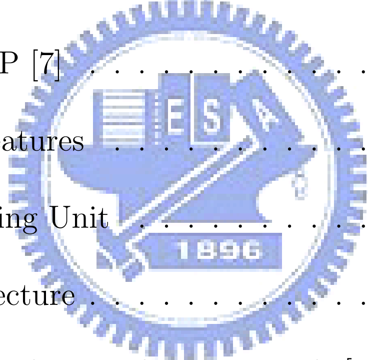
3.3.2 Using Intrinsics 39

4 Uplink Channel Estimation and DSP Implementation 41

4.1 Channel Estimation Techniques 41

4.1.1 The Least-Squares (LS) Estimator 42

4.1.2 Linear Interpolation 43



4.1.3 Time Averaging 43

4.1.4 Application to IEEE 802.16e OFDMA Uplink 45

4.2 Simulation Parameters and Channel Model 46

4.2.1 OFDMA Uplink System Parameters 46

4.2.2 Simulation Channel Model 47

4.3 Simulation Results 47

4.3.1 Simulation Flow 47

4.3.2 Validation of Simulation Model 49

4.3.3 Floating-point Simulation 50

4.4 DSP Implementation 60

4.4.1 Fixed-Point Data Formats 60

4.4.2 Fixed-Point Simulation 61

4.4.3 DSP Computational Load 63

4.5 Appendix 64

5 Downlink Channel Estimation and DSP Implementation 71

5.1 System Parameters and Channel Model 71

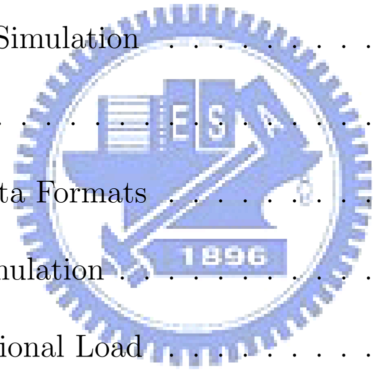
5.2 Channel Estimation Methods 71

5.2.1 Two-Point Cluster Linear Interpolation 72

5.2.2 Four-Point Cluster Linear Interpolation 74

5.2.3 Advanced Four-Point Cluster Linear Interpolation 76

5.3 Simulation Results 78



5.3.1 Simulation Flow 78

5.3.2 Validation with AWGN Channel 78

5.3.3 Floating-Point Simulation 79

5.3.4 Cluster Analysis 91

5.4 DSP Implementation 95

5.4.1 Fixed-Point Data Formats 95

5.4.2 Fixed-Point Simulation 95

5.4.3 DSP Simulation Loading 98

5.5 Appendix 99

6 Conclusion and Future Work 111

6.1 Conclusion 111

6.2 Potential Future Work 112

Bibliography 113

List of Figures

2.1 Discrete-time model of the baseband OFDMA system (from[3]). 4

2.2 OFDMA symbol time structure (from [5]). 5

2.3 Discrete-time baseband equivalent of an OFDMA system with M users (from [4]). 6

2.4 Example of the data region which defines the OFDMA allocation (from [5]). 8

2.5 OFDMA frequency description (from [5]). 8

2.6 Example of an OFDMA frame (with only mandatory zone) in TDD mode (from [6]). 10

2.7 Example of mapping OFDMA slots to subchannels and symbols in the uplink (from [6]). 12

2.8 Description of an uplink tile (from [5]). 12

2.9 PRBS generator for pilot modulation (from [5] and [6]). 15

2.10 QPSK, 16-QAM, and 64-QAM constellations (from [5]). 16

2.11 Example of mapping OFDMA slots to subchannels and symbols in the downlink in PUSC mode (from [6]). 17

2.12 Downlink transmission basic structure (from [5]). 18

2.13 Cluster structure (from [6]). 19

3.1 The DSP on the Sundance board 24

3.2 Block diagram of the TMS320C6416 DSP [7]. 26

3.3 Pipeline phases of TMS320C6416 DSP [7]. 27

3.4 TMS320C64x CPU data paths [7]. 32

3.5 Code development flow for TI C6000 DSP [12]. 36

4.1 Tile structure. 45

4.2 Block diagram of the simulated system. 49

4.3 The SER curve for uncoded QPSK resulting from simulation matches the theoretical one. 50

4.4 Tile linear interpolation with different exponential weighting in AWGN with QPSK. (a) MSE. (b) SER. 52

4.5 Tile linear interpolation with different exponential weighting in SUI-2 with velocity v=60 km/hr with QPSK. (a) MSE. (b) SER. 53

4.6 Tile linear interpolation of exponential weighting 0.9 with different velocities in SUI-2 with QPSK. (a) MSE. (b) SER. 54

4.7 Tile linear interpolation with different modulations in AWGN. (a) MSE. (b) SER. 55

4.8 Tile linear interpolation compared with theory adding data MSE in AWGN: (a),(b) QPSK. (c),(d) 16QAM. (e),(f) 64QAM. 56

4.9 Tile linear interpolation with different velocity and different modulations in SUI-2. (a),(b) QPSK. (c),(d) 16QAM. (e),(f) 64QAM. 57

4.10 Tile linear interpolation with different velocities in SUI-3 with QPSK. (a) MSE. (b)SER. 58

4.11 Tile linear interpolation with different used subchannels. (a),(b) AWGN. (c),(d) SUI-2. 59

4.12 Fixed-point data format in our design. 60

4.13 Fixed-point formats in channel estimation of our design. 60

4.14 Performance of fixed-point computation of tile linear interpolation (10 used subchannels) compared to floating-point computation (a),(b) AWGN. (c),(d) SUI-2. 62

4.15 *FIXED.H*. 64

4.16 Function *channel_estimation_FIXED*. 65

4.17 Function *pilot_extraction_FIXED*. 65

4.18 Function *interpolation_FIXED*. 66

4.19 Assembly code of function *channel_estimation_FIXED*. 67

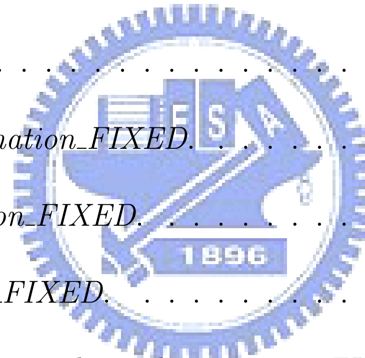
4.20 Assembly code of function *interpolation_FIXED*. 68

4.21 Software pipelining information of function *channel_estimation_FIXED*. . . . 69

5.1 Structure of cluster organization in time. 73

5.2 (a) 2-point cluster linear interpolation illustration, bold line is our estimation of linear interpolation (b) pilot positions are different in even and odd symbols 73

5.3 (a) Pilots in previous symbol taken as reference. (b) Four pilot points in cluster. (c) Four-point cluster linear interpolation illustration. Bold line is our estimation by linear interpolation. 75



5.4 Advanced four-point cluster linear interpolation. (a) First data symbol. (b) Second to $(n-1)$th data symbols. (c) Last (nth) data symbol. 76

5.5 Downlink transmission simulation flow. (a) Preamble. (b) Data symbols. . . 79

5.6 The SER curve for uncoded QPSK resulting from simulation matches the theoretical one. 80

5.7 Two-point cluster linear interpolation with different exponential weighting with QPSK. (a),(b) In AWGN. (c),(d) In SUI-2 with velocity v=60 km/hr. . 82

5.8 Three methods of cluster cluster linear interpolation with different velocity in SUI-2 of QPSK. (a),(b) Two-point with exponential weighting w=0.9. (c),(d) Two-point. (e),(f) Four-point. 83

5.9 Comparison of all methods we use, including two-point, two-point with exponential weighting w=0.9, four-point and advanced four-point cluster linear interpolation in AWGN. (a) MSE. (b) SER. 84

5.10 Comparison of all methods we use, including two-point , two-point with exponential weighting w=0.9, four-point and advanced four-point cluster linear interpolation in SUI-2 with velocity v=60 km/hr. (a) MSE. (b) SER. 85

5.11 Advanced four-point linear interpolation with different modulation in AWGN. (a) MSE. (b) SER. 86

5.12 Advanced four-point cluster linear interpolation compared with theory adding data MSE in AWGN. (a),(b) QPSK. (c),(d) 16QAM. (e),(f) 64QAM. 87

5.13 Advanced four-point cluster linear interpolation with different velocities and different modulations in SUI-2. (a),(b) QPSK. (c),(d) 16QAM. (e),(f) 64QAM. 88

5.14 Advanced four-point cluster linear interpolation with different velocities in SUI-3 with QPSK. (a) MSE. (b) SER. 89

5.15 Advanced four-point cluster linear interpolation considering preamble effect. (a),(b) AWGN. (c),(d) SUI-2. . . . 90

5.16 Advanced four-point cluster linear interpolation with no preamble in AWGN. (a) MSE. (b) SER. . . . 91

5.17 (a)MSE and (b)SER over used subcarriers in AWGN at 10 dB SNR. . . . 92

5.18 Average cluster performance in AWGN at 10 dB SNR. (a) MSE. (b) SER. . 93

5.19 Average cluster performance in AWGN at 10 dB SNR. (a) MSE. (b) SER. . 93

5.20 Average cluster performance in AWGN at 10 dB SNR. (a),(b) Even symbols. (c),(d) Odd Symbols. . . . 94

5.21 Fixed-point preamble transmission formats in our design. . . . 95

5.22 Fixed-point data transmission formats in our design. . . . 96

5.23 Fixed-point data formats in preamble estimation of our design. . . . 96

5.24 Fixed-point data formats in channel estimation of our design. . . . 96

5.25 Fixed-point computation of advanced four-point cluster linear interpolation in AWGN. (a) MSE. (b) SER. . . . 97

5.26 Fixed-point computation of advanced four-point cluster linear interpolation in SUI-2. (a) MSE. (b) SER. . . . 97

5.27 *FIXED.H.* . . . 99

5.28 Function *preamble_estimation_FIXED.* . . . 100

5.29 Function *channel_estimation_FIXED.* . . . 101

5.30 Function *pilot_extraction_FIXED.* . . . 101

5.31 Function *interpolation_FIXED.* . . . 102

5.32 Function *interpolation_FIXED* (cont.). 103

5.33 Function *interpolation_FIXED* (cont.). 104

5.34 Assembly code of function *preamble_estimation_FIXED*. 105

5.35 Assembly code of function *channel_estimation_FIXED*. 106

5.36 Assembly code of function *pilot_extraction_FIXED*. 107

5.37 Assembly code of function *interpolation_FIXED*. 108

5.38 Software pipelining information of function *preamble_estimation_FIXED*. . . 109

5.39 Software pipelining information of function *channel_estimation_FIXED*. . . . 110

List of Tables

2.1 OFDMA Uplink Subcarrier Allocations [5], [6] 13

2.2 OFDMA Downlink Subcarrier Allocation under PUSC [5], [6] 20

3.1 Execution Stage Length Description for Each Instruction Type [7] 28

3.2 Functional Units and Operations Performed [7] 30

3.3 Functional Units and Operations Performed (Continued) [7] 31

4.1 OFDMA Uplink Parameters 46

4.2 Channel Profiles of SUI-2 and SUI-3 [16] 48

4.3 OFDMA Uplink DSP Loading 63

4.4 OFDMA Uplink Efficiency Performance Comparison 70

5.1 OFDMA Downlink Parameters 72

5.2 OFDMA DL DSP Loading for Channel Estimation in 2048-FFT, BW: 20 MHz 98

Chapter 1

Introduction

Orthogonal frequency division multiple access (OFDMA) has emerged as one of the prime multiple access schemes for broadband wireless networks (e.g., IEEE 802.16 Mobile WiMAX, DVB-RCA, etc.). As a special case of multicarrier multiple access schemes, OFDMA exclusively assigns each subchannel to only one user, eliminating the intra-cell interference (ICI). For fixed or portable applications where the frequency selective channels are slowly varying, an intrinsic advantage of OFDMA is its capability to exploit the so-called multiuser diversity embedded in multipath channels. Furthermore, OFDMA has the merit of easy decoding at the receiver side due to the absence of ICI. Other advantages of OFDMA include finer granularity and better link budget [1]. OFDMA can be easily generated using an inverse fast Fourier transform (IFFT) and received using a fast Fourier transform (FFT).

The IEEE 802.16 standard committee has developed a group of standards for wireless metropolitan area networks (MANs). OFDMA is used in the 2 to 11 GHz Fixed Wireless Access (FWA) systems. IEEE 802.16 has developed the IEEE Standard 802.16-2004 for broadband wireless access systems, which provides a variety of services to fixed outdoor as well as nomadic indoor users. The 802.16e is designed to support terminal mobility, and currently it aims to serve terminals with a speed of 120 km/hr [2].

This thesis focuses on the channel estimation part for WirelessMAN-OFDMA in both uplink and downlink transmission, and it is organized as follows. First, in chapter 2, we introduce some OFDMA basics in the IEEE 802.16e OFDMA uplink and downlink standard. In chapter 3, we describe the implementation platform, which consists of Texas Instrument's TMS320C6416 digital signal processor (DSP) on a Sundance Carrier board. In chapter 4, the various channel estimation techniques are introduced and we discuss the performance of channel estimation methods in uplink transmission and some DSP implementation issues. In chapter 5, we propose several methods for downlink, compare the performance of each method and also give some DSP implementation issues. At last, we mention the conclusion and give some potential future work in chapter 6.

Chapter 2

Introduction to IEEE802.16e OFDMA

We first give the basic concept of the OFDMA techniques for multicarrier modulation. The downlink and uplink specifications of IEEE 802.16e are introduced afterward.

2.1 Overview of OFDMA [3], [4]

Orthogonal frequency-division multiple-access (OFDMA) is being considered to be the multiple access scheme for future wireless systems, e.g., WiMAX or fourth-generation (4G) broadband wireless networks. In an OFDMA system, several users simultaneously transmit their data by modulating an exclusive set of orthogonal subcarriers, thus each user's signal can be separated easily in the frequency domain. One typical structure is the subband OFDMA, which divides all available subcarriers into a number of subbands. Each user is allowed to use one available subband for the data transmission. Pilot symbols are employed for the estimation of channel state information (CSI) within the subband. Furthermore, robustness to narrowband interference and dynamic channel assignment are other two advantages of OFDMA systems. Figure 2.1 shows an OFDMA network in which active users simultaneously communicate with the base station (BS).

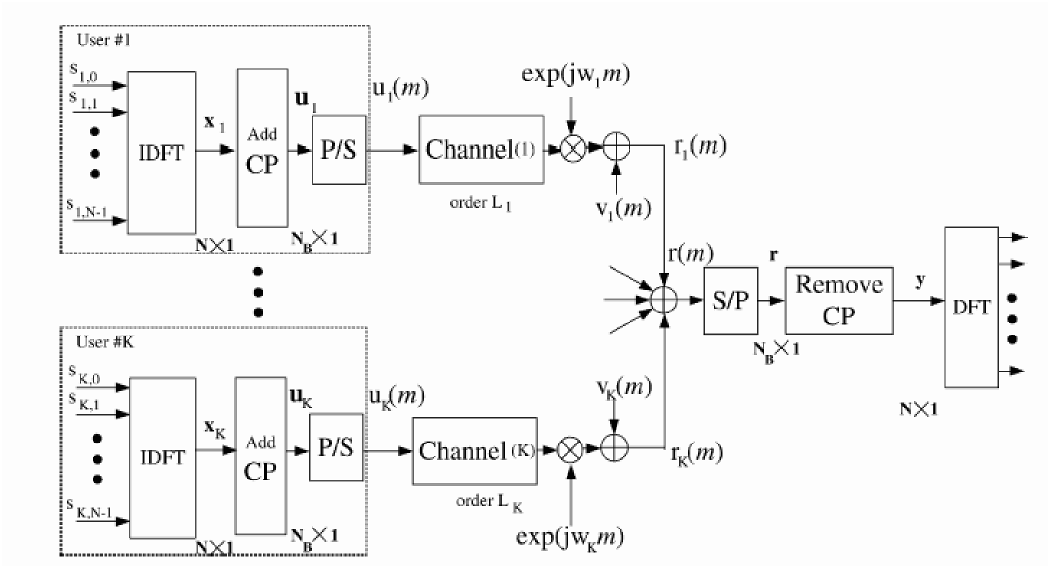


Figure 2.1: Discrete-time model of the baseband OFDMA system (from[3]).

2.1.1 Cyclic Prefix

Cyclic prefix (CP) is used to overcome the intersymbol and interchannel interference problems. The multiuser channel is assumed to be substantially invariant within one-block (or -symbol) duration. The symbol timing mismatch is assumed to be smaller than the CP duration. In this scenario, users do not interfere each other in the frequency domain.

A CP is a copy of the last part of the OFDMA symbol (see Fig. 2.2). A copy of the last T_g of the useful symbol period, termed CP, is used to collect multipath while maintaining the orthogonality of the tones. However, the transmitter energy increases with the length of the guard time while the receiver energy remains the same (the cyclic extension is discarded), so there is a 10 log(1-$T_g/(T_b+T_g)$)/log(10) dB loss in E_b/N_0.

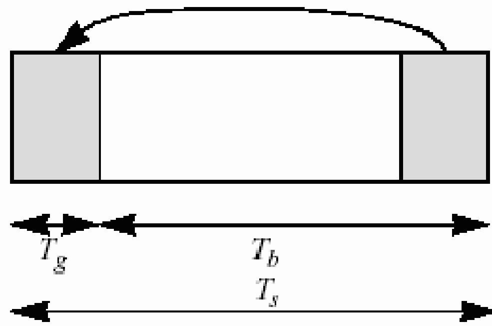


Figure 2.2: OFDMA symbol time structure (from [5]).

2.1.2 Discrete Time Baseband Equivalent System Model

The material in this subsection is mainly taken from [4]. If we consider an OFDMA system with M active users sharing a bandwidth of $B = \frac{1}{T}$ Hz (T is the sampling period) as shown in Fig. 2.3. The system consists of K subcarriers of which K_u are useful subcarriers (excluding guard bands and DC subcarrier). The users are allocated non-overlapping subcarriers in the spectrum depending on their needs.

The discrete time baseband channel consists of L multipath components and has the form

$$h(l) = \sum_{m=0}^{L-1} h_m \delta(l - l_m) \tag{2.1}$$

where h_m is a zero-mean complex Gaussian random variable with $E[h_i h_j^*] = 0$ for $i \neq j$. In frequency domain

$$H = Fh \tag{2.2}$$

where $H = [H_0, H_1, ..., H_{K-1}]^T$, $h = [h_0, ..., h_{L-1}, 0, ..., 0]^T$ and F is K-point DFT matrix. The impulse response length l_{L-1} is upper bounded by the length of CP (L_{cp}).

The received signal in frequency domain is given by

$$Y_n = \sum_{i=1}^{M} X_{i,n} H_{i,n} + V_n \tag{2.3}$$

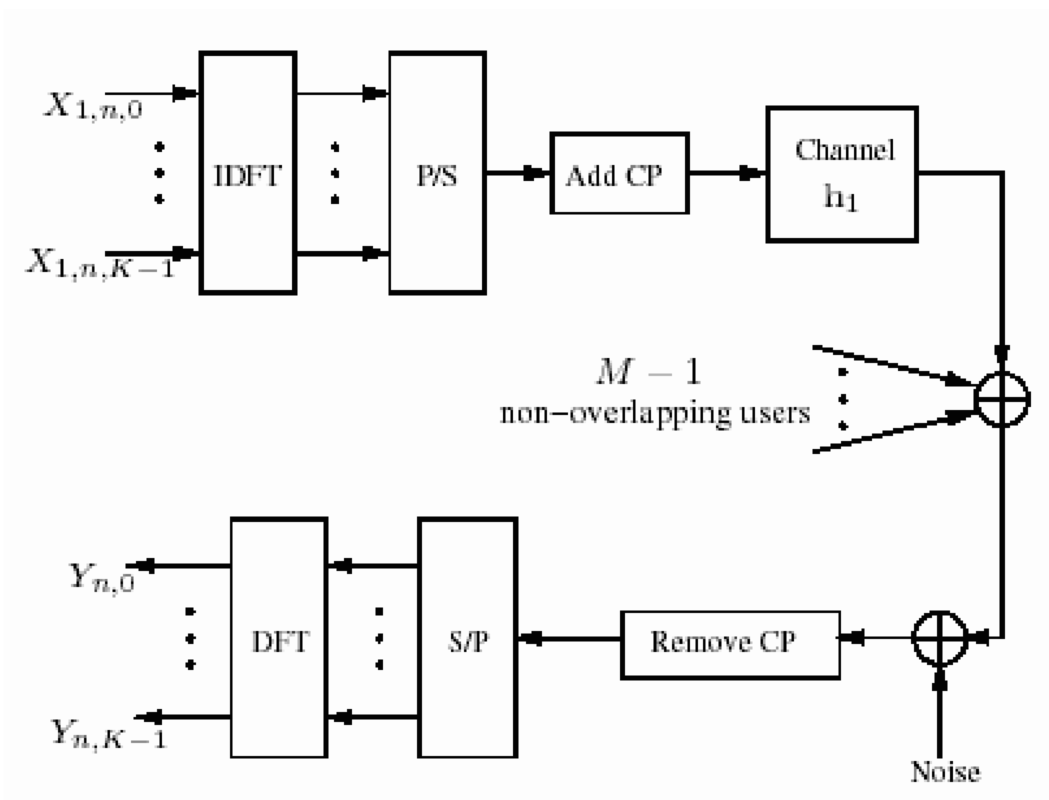


Figure 2.3: Discrete-time baseband equivalent of an OFDMA system with M users (from [4]).

where $X_{i,n} = \text{diag}(X_{i,n,0}, ..., X_{i,n,K-1})$ is $K \times K$ diagonal data matrix and $H_{i,n}$ is the $K \times 1$ channel vector (2.2) corresponding to the ith user in nth symbol. The noise vector V_n is distributed as $\mathcal{CN}(0, \sigma^2 I_K)$.

2.2 Basic OFDMA Symbol Structure in IEEE 802.16e

The WirelessMAN-OFDMA PHY, based on OFDM modulation, is designed for nonline-of-sight (NLOS) operation in frequency bands below 11 GHz. For licensed bands, channel bandwidths allowed shall be limited to the regulatory provisioned bandwidth divided by any power of 2 no less than 1.0 MHz. The material is mainly taken from [5] and [6].

2.2.1 OFDMA Basic Terms

We introduce some basic terms appeared in OFDMA PHY. These definitions would help us to understand the concepts of subcarrier allocation and transmission of IEEE 802.16e OFDMA.

- Slot: A slot in the OFDMA PHY is a two-dimensional entity spanning both a time and a subchannel dimension. It is the minimum possible data allocation unit. For downlink (DL) PUSC (Partial Usage of SubChannels), one slot is one subchannel by two OFDMA symbols. For uplink (UL), one slot is one subchannel by three OFDMA symbols.

- Data Region: In OFDMA, a data region is a two-dimensional allocation of a group of contiguous subchannels, in a group of contiguous OFDMA symbols. All the allocations refer to logical subchannels. A two dimensional allocation may be visualized as a rectangle, such as the 4×3 rectangle shown in Fig. 2.4.

- Segment: A segment is a subdivision of the set of available OFDMA subchannels (that may include all available subchannels). One segment is used for deploying a single instance of the MAC.

2.2.2 Frequency Domain Description

An OFDMA symbol (see Fig. 2.5) is made up of subcarriers, the number of which determines the FFT size used. There are several subcarrier types:

- Data subcarriers: For data transmission.

- Pilot subcarriers: For various estimation purposes.

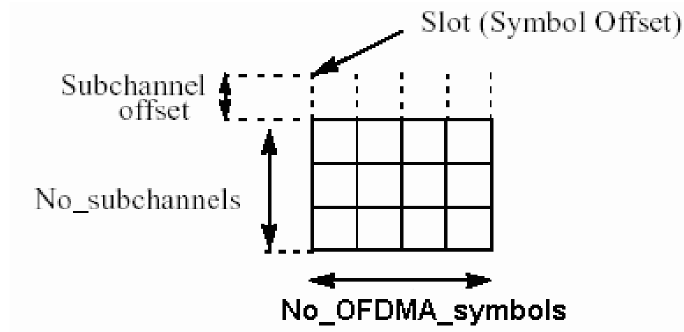


Figure 2.4: Example of the data region which defines the OFDMA allocation (from [5]).

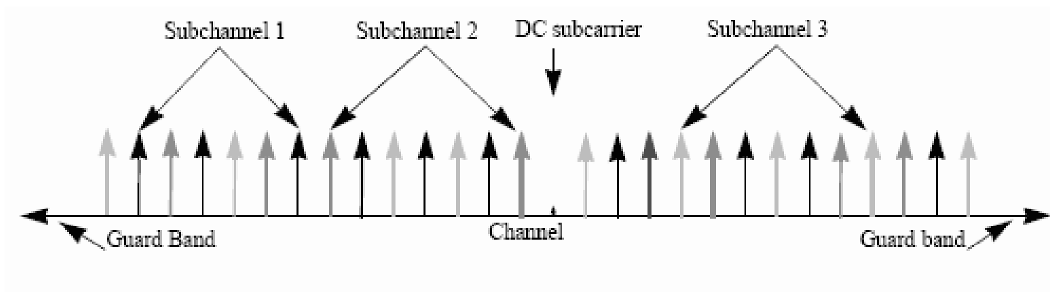


Figure 2.5: OFDMA frequency description (from [5]).

- Null subcarriers: No transmission at all, for guard bands and DC subcarrier.

2.2.3 Primitive Parameters

Four primitive parameters characterize the OFDMA symbols:

- BW: The nominal channel bandwidth.
- N_{used}: Number of used subcarriers (which includes the DC subcarrier).
- n: Sampling factor. This parameter, in conjunction with BW and N_{used}, determines the subcarrier spacing and the useful symbol time. Its value is set as follows: For channel bandwidths that are a multiple of 1.75 MHz n = 8/7, else for channel bandwidths

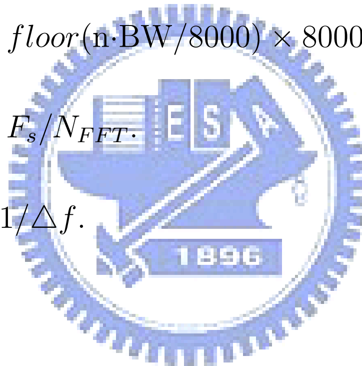
that are a multiple of any of 1.25, 1.5, 2 or 2.75 MHz n = 28/25, else for channel bandwidths not otherwise specified n = 8/7.

- G: This is the ratio of CP time to "useful" time, i.e., T_{cp}/T_s. The following values shall be supported: 1/32, 1/16, 1/8, and 1/4.

2.2.4 Derived Parameters

The following parameters are defined in terms of the primitive parameters.

- N_{FFT}: Smallest power of two greater than N_{used}.
- Sampling frequency: $F_s = floor(\text{n}\cdot\text{BW}/8000) \times 8000$.
- Subcarrier spacing: $\triangle f = F_s/N_{FFT}$.
- Useful symbol time: $T_b = 1/\triangle f$.
- CP time: $T_g = G \times T_b$.
- OFDMA symbol time: $T_s = T_b + T_g$.
- Sampling time: T_b/N_{FFT}.



2.2.5 Frame Structure

When implementing a time-division duplex (TDD) system, the frame structure is built from base station (BS) and subscriber station (SS) transmissions. Each frame in the DL transmission begins with a preamble followed by a DL transmission period and an UL transmission period. In each frame, the TTG and RTG shall be inserted between the downlink and uplink and at the end of each frame, respectively, to allow the BS to turn around. Fig. 2.6 shows an example of an OFDMA frame with only mandatory zone in TDD mode.

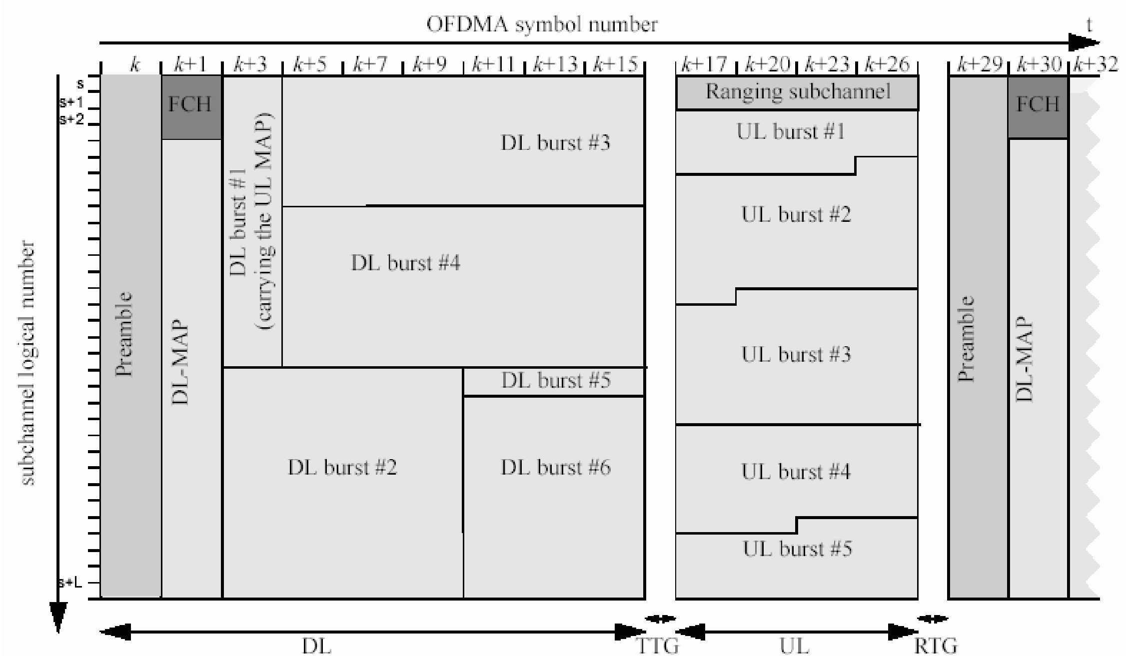


Figure 2.6: Example of an OFDMA frame (with only mandatory zone) in TDD mode (from [6]).

2.3 Uplink Transmission in IEEE 802.16e OFDMA

In this section we briefly introduce the specification of IEEE 802.16e OFDMA uplink transmission. The material is mainly taken from [5] and [6].

2.3.1 Data Mapping Rules

The UL mapping consists of two steps. In the first step, the OFDMA slots allocated to each burst are selected. In the second step, the allocated slots are mapped.

Step 1 : Allocate OFDMA slots to bursts.

1) Segment the data into blocks sized to fit into one OFDMA slot.

2) Each slot shall span one or more subchannels in the subchannel axis and one or more

OFDMA symbols in the time axis (see Fig. 2.7 for an example). Map the slots such that the lowest numbered slot occupies the lowest numbered subchannel in the lowest numbered OFDMA symbol.

3) Continue the mapping such that the OFDMA symbol index is increased. When the edge of the UL zone is reached, continue the mapping from the lowest numbered OFDMA symbol in the next available subchannel.

4) An UL allocation is created by selecting an integer number of contiguous slots, according to the ordering of steps 1 to 3. This results in the general Burst structure shown by the gray area in Fig. 2.7.

Step 2 : Map OFDMA slots within the UL allocation.

1) Map the slots such that the lowest numbered slot occupies the lowest numbered subchannel in the lowest numbered OFDMA symbol.

2) Continue the mapping such that the Subchannel index is increased. When the last subchannel is reached, continue the mapping from the lowest numbered subchannel in the next OFDMA symbol that belongs to the UL allocation. The resulting order is shown by the arrows in Fig. 2.7.

Fig. 2.7 illustrates the order of OFDMA slots mapping to subchannels and OFDMA symbols.

2.3.2 Carrier Allocations

The uplink supports 70 subchannels for 2048-FFT PUSC permutation. Each transmission uses 48 data carriers as the minimal block of processing. Each new transmission for the uplink commences with the parameters as given in Table 2.1.

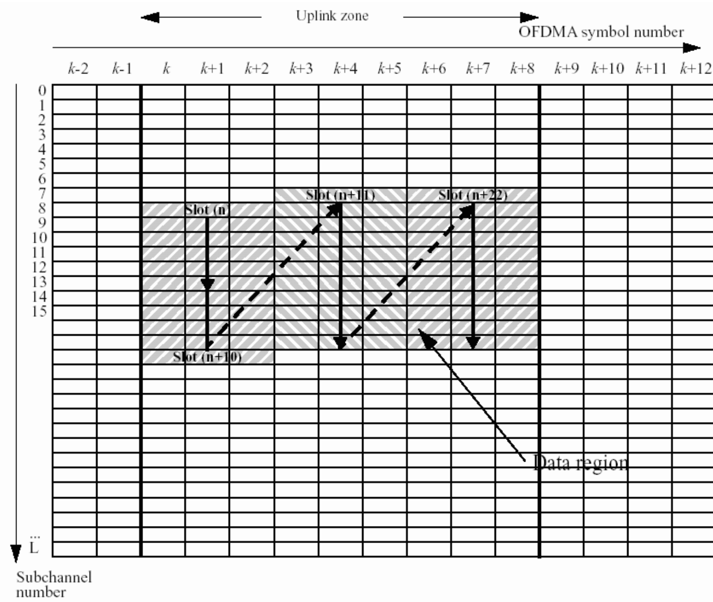


Figure 2.7: Example of mapping OFDMA slots to subchannels and symbols in the uplink (from [6]).

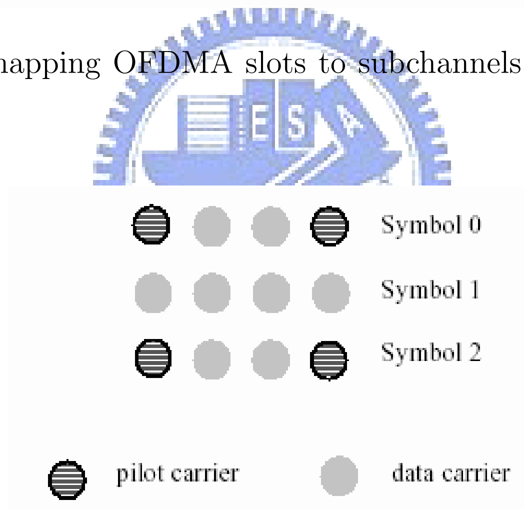


Figure 2.8: Description of an uplink tile (from [5]).

A slot in the uplink is composed of three OFDMA symbols and one subchannel. Within each slot, there are 48 data subcarriers and 24 pilot subcarriers. The subchannel is constructed from six uplink tiles, each having four successive active subcarriers with the configuration as illustrated in Fig. 2.8.

The usable subcarriers in the allocated frequency band shall be divided into N_{tiles} physical

Table 2.1: OFDMA Uplink Subcarrier Allocations [5], [6]

Parameter	Value	Notes
Number of DC subcarriers	1	Index 1024 (counting from 0)
N_{used}	1681	Number of all subcarriers used within a symbol
Guard subcarriers: Left, Right	184,183	
TilePermutation		Used to allocate tiles to subchannels 6, 48, 58, 57, 50, 1, 13, 26, 46, 44, 30, 3, 27, 53, 22, 18, 61, 7, 55, 36, 45, 37, 52, 15, 40, 2, 20, 4, 34, 31, 10, 5, 41, 9, 69, 63, 21, 11, 12, 19, 68, 56, 43, 23, 25, 39, 66, 42, 16, 47, 51, 8, 62, 14, 33, 24, 32, 17, 54, 29, 67, 49, 65, 35, 38, 59, 64, 28, 60, 0
$N_{subchannels}$	70	
$N_{subcarriers}$	48	
N_{tiles}	420	
Number of subcarriers per tile	4	Number of all subcarriers within a tile
Tiles per subchannel	6	

tiles with parameters from Table 2.1. The allocation of physical tiles to logical tiles in subchannels is performed according to:

$$Tiles(s,n) = N_{subchannels} \cdot n + (Pt[(s+n)\ mod\ N_{subchannels}] + UL_PermBase)\mathrm{mod}\ N_{subchannels}$$

where:

- $Tiles(s,n)$ is the physical tile index in the FFT with tiles being ordered consecutively from the most negative to the most positive used subcarrier (0 is the starting tile index),
- n is the tile index 0..5 in a subchannel,
- Pt is the tile permutation,

- s is the subchannel number in the range $0...N_{subchannels} - 1$,
- $UL_PermBase$ is an integer value in the range 0..69, which is assigned by a management entity, and
- $N_{subchannels}$ is the number of subchannels for the FFT size given in Table 2.1.

After mapping the physical tiles to logical tiles for each subchannel, the data subcarriers per slot are enumerated by the following process:

1) After allocating the pilot carriers within each tile, indexing of the data subcarriers within each slot is performed starting from the first symbol at the lowest indexed subcarrier of the lowest indexed tile and continuing in an ascending manner through the subcarriers in the same symbol, then going to the next symbol at the lowest indexed data subcarrier, and so on. Data subcarriers shall be indexed from 0 to 47.

2) The mapping of data onto the subcarriers will follow the equation below. This equation calculates the subcarrier index (as assigned in item 1) to which the data constellation point is to be mapped:

$$Subcarrier(n, s) = (n + 13 \cdot s) \ mod \ N_{subcarriers}$$

   where:

   - $Subcarrier(n, s)$ is the permutated subcarrier index corresponding to data subcarrier n is subchannel s,
   - n is a running index 0..47, indicating the data constellation point,
   - s is the subchannel number, and
   - $N_{subcarriers}$ is the number of subcarriers per slot.

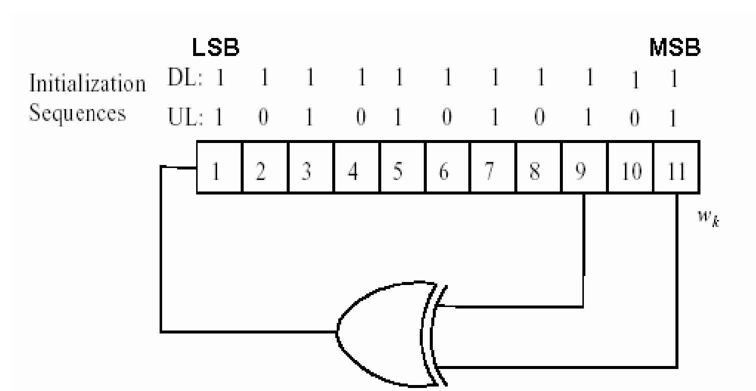


Figure 2.9: PRBS generator for pilot modulation (from [5] and [6]).

2.3.3 Pilot Modulation

The PRBS (pseudo-random binary sequence) generator depicted in Fig. 2.9 is used to produce a sequence, w_k. The value of the pilot modulation, on subcarrier k, shall be derived from w_k.

For the mandatory tile structure in the uplink, pilot subcarriers shall be inserted into each data burst in order to constitute the symbol and they shall be modulated according to their subcarrier location within the OFDMA symbol. The pilot subcarriers shall be modulated according to

$$\Re\{c_k\} = 2\Big(\frac{1}{2} - w_k\Big), \quad \Im\{c_k\} = 0. \tag{2.4}$$

2.3.4 Data Modulation

As shown in Fig. 2.10, the data bits are entered serially to the constellation mapper. Gray-mapped QPSK and Gray-mapped 16QAM shall be supported, whereas the support of 64QAM (also Gray-mapped) is optional.

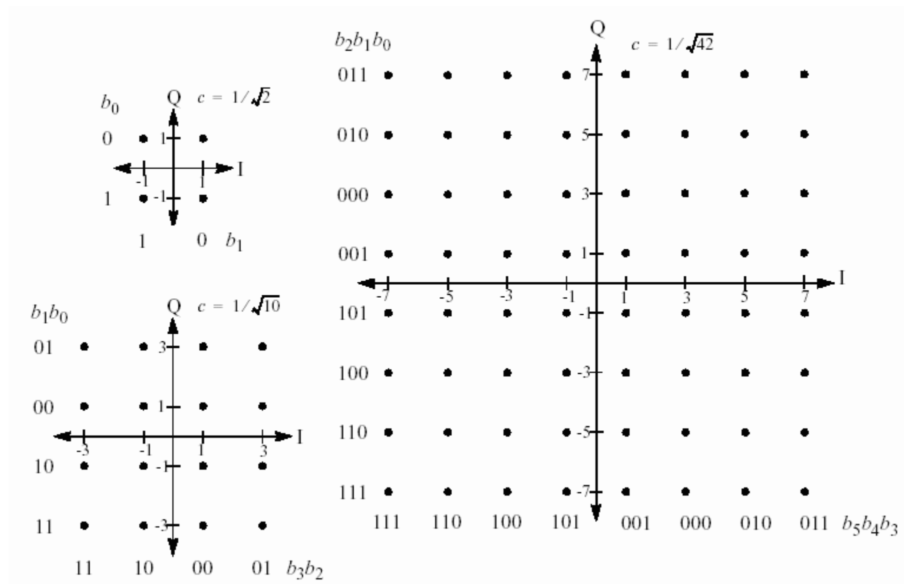


Figure 2.10: QPSK, 16-QAM, and 64-QAM constellations (from [5]).

2.4 Downlink Transmission in IEEE 802.16e OFDMA

This section briefly introduces the specifications of IEEE 802.16e OFDMA PUSC downlink transmission. The material is mainly taken from [5] and [6].

2.4.1 Data Mapping Rules

The downlink data mapping rules are as follows:

1. Segment the data after the modulation block into blocks sized to fit into one OFDMA slot.

2. Each slot shall span one subchannel in the subchannel axis and one or more OFDMA symbols in the time axis, as per the slot definition mentioned before. Map the slots such that the lowest numbered slot occupies the lowest numbered subchannel in the lowest numbered OFDMA symbol.

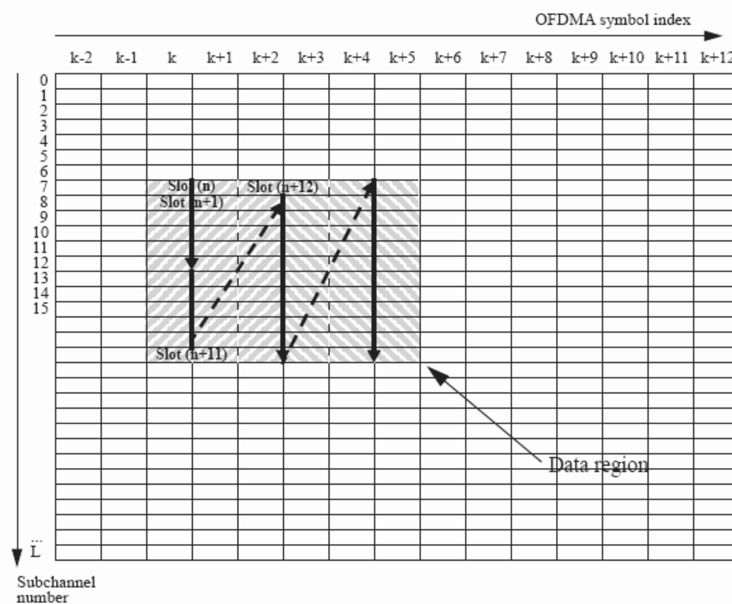


Figure 2.11: Example of mapping OFDMA slots to subchannels and symbols in the downlink in PUSC mode (from [6]).

3. Continue the mapping such that the OFDMA subchannel index is increased. When the edge of the Data Region is reached, continue the mapping from the lowest numbered OFDMA subchannel in the next available symbol.

Figure 2.11 illustrates the order of OFDMA slots mapping to subchannels and OFDMA symbols.

2.4.2 Preamble Structure and Modulation

The first symbol of the downlink transmission is the preamble. Fig. 2.12 shows a downlink transmission period. There are three types of preamble carrier-sets, those are defined by allocation of different subcarriers for each one of them. The subcarriers are modulated using a boosted BPSK modulation with a specific pseudo-noise (PN) code. The PN series modulating the pilots in the preamble can be found in [5, pp. 553–562].

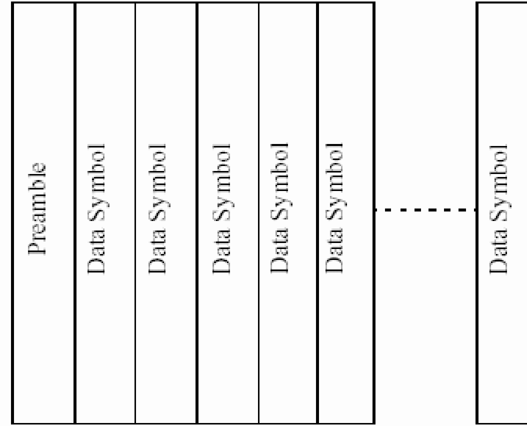


Figure 2.12: Downlink transmission basic structure (from [5]).

The preamble carrier-sets are defined as

$$PreambleCarrierSet_n = n + 3 \cdot k, \tag{2.5}$$

where:

- $PreambleCarrierSet_n$ specifies all subcarriers allocated to the specific preamble,
- n is the number of the preamble carrier-set indexed 0, 1, 2, and
- k is a running index 0,...,567.

Each segment uses one type of preamble out of the three sets in the following manner: For the preamble symbol, there will be 172 guard band subcarriers on the left side and the right side of the spectrum. Segment i uses preamble carrier-set i, where $i = 0, 1, 2$. The DC subcarrier will not be modulated at all and the appropriate PN will be discarded. Therefore, DC subcarrier shall always be zeroed.

The pilots in downlink preamble shall be modulated as

$$\begin{aligned} \Re\{PreamblePilotsModulated\} &= 4 \cdot \sqrt{2} \cdot (\frac{1}{2} - w_k), \\ \Im\{PreamblePilotsModulated\} &= 0. \end{aligned} \tag{2.6}$$

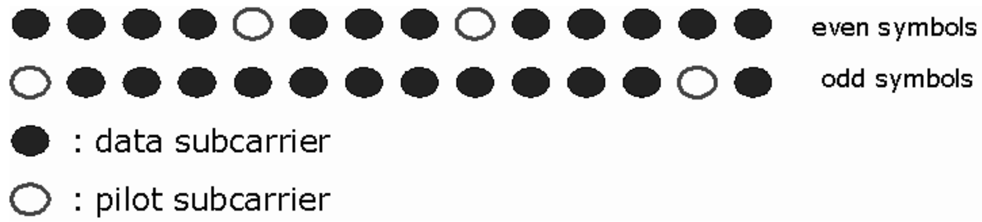


Figure 2.13: Cluster structure (from [6]).

2.4.3 Subcarrier Allocations

The OFDMA symbol structure is constructed using pilots, data and zero subcarriers. The symbol is first divided into basic clusters and zero carriers are allocated. The pilot tones are allocated first; what remains are data subcarriers, which are divided into subchannels that are used exclusively for data. Pilots and data carriers are allocated within each cluster.

Figure 2.13 shows the cluster structure with subcarriers from left to right in order of increasing subcarrier index. For the purpose of determining PUSC pilot location, odd and even symbols are counted from the beginning of the current zone. The first symbol in the zone is even. The preamble shall not be counted as part of the first zone. Table 2.2 summarizes the parameters of the OFDMA PUSC symbol structure.

The allocation of subcarriers to subchannels is performed using the following procedure:

1) Divide the subcarriers into a number ($N_{clusters}$) of physical clusters containing 14 adjacent subcarriers each (starting from carrier 0).

2) Renumber the physical clusters into logical clusters using the following formula:

$$
\begin{aligned}
&LogicalCluster \\
&= \begin{cases} RenumberingSequence(PhysicalCluster), & \text{first DL zone,} \\ RenumberingSequence\big((PhysicalCluster+ \\ \qquad 13 \cdot DL_PermBase) \bmod N_{clusters}\big), & \text{otherwise.} \end{cases}
\end{aligned}
$$

Table 2.2: OFDMA Downlink Subcarrier Allocation under PUSC [5], [6]

Parameter	Value	Comments
Number of DC subcarriers	1	Index 1024 (counting from 0)
Number of guard subcarriers, left	184	
Number of guard subcarriers, right	183	
Number of used subcarriers (N_{used})	1681	Number of all subcarriers used within a symbol, including all possible allocated pilots and the DC carrier
Number of subcarriers per cluster	14	
Number of clusters	120	
Renumbering sequence	1	Used to renumber clusters before allocation to subchannels: 6,108,37,81,31,100,42,116,32,107,30,93,54,78, 10,75,50,111,58,106,23,105,16,117,39,95,7, 115,25,119,53,71,22,98,28,79,17,63,27,72,29, 86,5,101,49,104,9,68,1,73,36,74,43,62,20,84, 52,64,34,60,66,48,97,21,91,40,102,56,92,47, 90,33,114,18,70,15,110,51,118,46,83,45,76,57, 99,35,67,55,85,59,113,11,82,38,88,19,77,3,87, 12,89,26,65,41,109,44,69,8,61,13,96,14,103,2, 80,24,112,4,94,0
Number of data subcarriers in each symbol per subchannel	24	
Number of subchannels	60	
Basic permutation sequence 12 (for 12 subchannels)	12	6,9,4,8,10,11,5,2,7,3,1,0
Basic permutation sequence 8 (for 8 subchannels)	8	7,4,0,2,1,5,3,6

3) Dividing the clusters into six major groups. Group 0 includes clusters 0–23, group 1 clusters 24–39, group 2 clusters 40–63, group 3 clusters 64–79, group 4 clusters 80–103 and group 5 clusters 104–119. These groups may be allocated to segments. If a segment is being used, then at least one group shall be allocated to it. (By default group 0 is allocated to segment 0, group 2 to segment 1, and group 4 to segment 2) .

4) Allocate subcarriers to subchannel in each major group separately for each OFDMA symbol by first allocating the pilot subcarriers within each cluster and then taking all remaining data subcarriers within the symbol. The exact partitioning into subchannels is according to the equation below, called a permutation formula:

$$subcarrier(k,s) = N_{subchannels} \cdot n_k + \{p_s[n_k \ mod \ N_{subchannels}] + DL_PermBase\} \mathrm{mod} \ N_{subchannels}$$

where:

- $subcarrier(k,s)$ is the subcarrier index of subcarrier k in subchannel s,
- s is the index number of a subchannel, from the set $[0...N_{subchannels}-1]$,
- $n_k = (k + 13 \cdot s) \mathrm{mod} \ N_{subcarriers}$, where k is the subcarrier-in-subchannel index from the set $[0...N_{subcarriers}-1]$,
- $N_{subchannels}$ is the number of subchannels (for PUSC use number of subchannels in the currently partitioned group),
- $p_s[j]$ is the series obtained by rotating basic permutation sequence cyclically to the left s times,
- $N_{subcarriers}$ is the number of data subcarriers allocated to a subchannel in each OFDMA symbol, and
- $DL_PermBase$ is an integer from 0 to 31.

2.4.4 Pilot Modulation

Pilot subcarriers shall be inserted into each data burst in order to constitute the symbol. The PRBS (pseudo-random binary sequence) generator depicted in Fig. 2.9 shall be used to produce a sequence, w_k.

Each pilot shall be transmitted with a boosting of 2.5 dB over the average non-boosted power of each data tone. The pilot subcarriers shall be modulated according to

$$\Re\{c_k\} = \frac{8}{3}\left(\frac{1}{2} - w_k\right), \quad \Im\{c_k\} = 0. \tag{2.7}$$

2.4.5 Data Modulation

As shown in Fig. 2.10, for downlink transmission, gray-mapped QPSK and Gray-mapped 16QAM shall be supported, whereas the support of 64QAM (also Gray-mapped) is optional.

Chapter 3

The DSP Hardware and Associated Software Development Environment

1896

DSP implementation is the final goal of our work. The DSP on the Sundance board is TMS320C6416 made by Texas Instruments(see Fig.3.1). In this chapter, we introduce the architectures of the DSP chip.

3.1 The TMS320C6416 DSP [7]

3.1.1 TMS320c64x Features

The TMS320C64x DSPs are the highest-performance fixed-point DSP generation of the TMS320C6000 DSP devices, with a performance of up to 600 million instructions per second (MIPS) and an efficient C compiler. The TMS320C64x device is based on the second-generation high-performance, very-long-instruction-word (VLIW) architecture developed by Texas Instruments (TI). The C6416 device has two high-performance embedded coprocessors, Viterbi Decoder Coprocessor (VCP) and Turbo Decoder Coprocessor (TCP) that significantly speed up channel-decoding operations on-chip. But they do not apply to the work reported in this thesis.

The C64x core CPU consists of 64 general-purpose 32-bits registers and 8 function units,

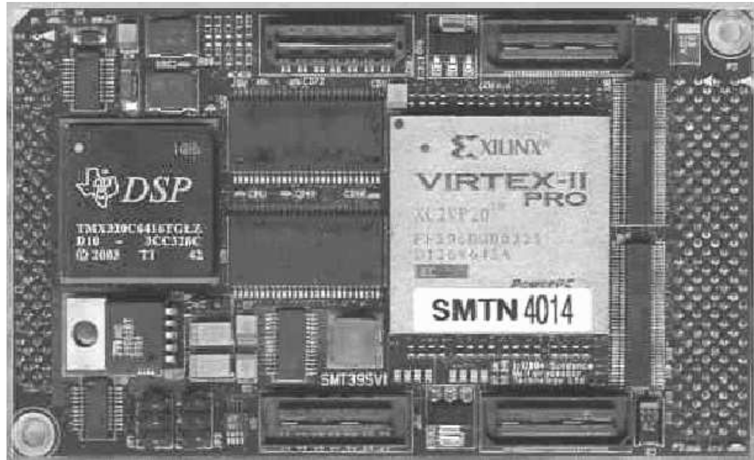


Figure 3.1: The DSP on the Sundance board

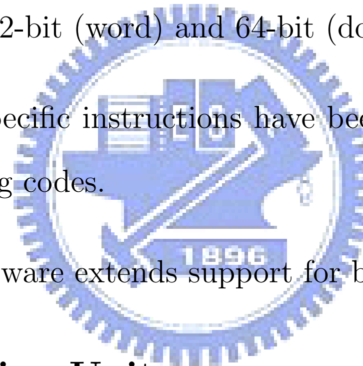
these 8 functional units contain 2 multipliers and 6 arithmetic units. C6000 features:

- Advanced VLIW executes up to eight instructions per cycle and allows designers to develop highly effective RISC-like code for fast development time.
- Instruction packing gives code size equivalence for eight instructions executed serially or in parallel and reduces code size, program fetches, and power consumption.
- Conditional execution of all instructions reduces costly branching. and increases parallelism for higher sustained performance.
- Efficient code execution on independent functional units include efficient C compiler on DSP benchmark suite. and assembly optimizer for fast development and improved parallelization.
- 8/16/32-bit data support, providing efficient memory support for a variety of applications.
- 40-bit arithmetic options add extra precision for applications requiring it.

- Saturation and normalization provide support for key arithmetic operations.
- Field manipulation and instruction extract, set, clear, and bit counting support common operation found in control and data manipulation applications.

The additional features of C64x include:

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



3.1.2 Central Processing Unit

The block diagram of the C6416 DSP is shown in the Fig. 3.2. The C64x CPU, shaded in figure, contains:

- Program fetch unit.
- Instruction dispatch unit.
- Instruction decode unit.
- Two data paths, each with four functional units.
- 64 32-bit registers.

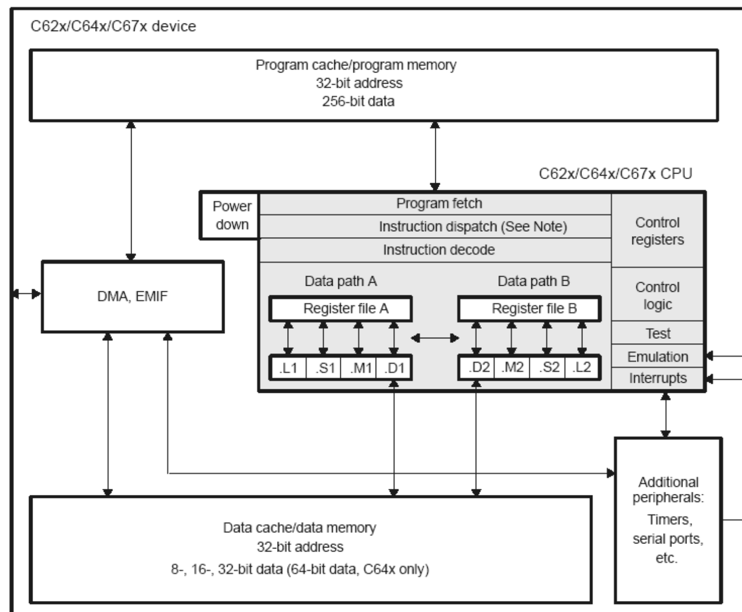
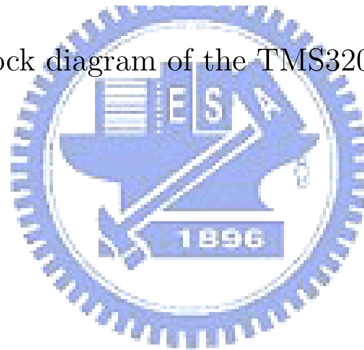


Figure 3.2: Block diagram of the TMS320C6416 DSP [7].



- Control registers.
- Control logic.
- Test, emulation, and interrupt logic.

The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, and .D) and 32 32-bit general-purpose registers for the C6416.

3.1.2.1 Pipeline Structure

The TMS320C64x DSP pipeline provides flexibility to simplify programming and improve performance. The pipeline can dispatch eight parallel instructions every cycle. The pipeline phases are divided into three stages as shown in Fig. 3.3.

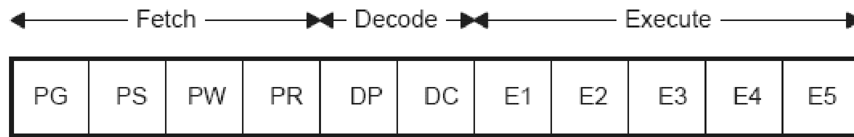


Figure 3.3: Pipeline phases of TMS320C6416 DSP [7].

- Fetch has 4 phases:
  - PG (program address generate): The address of the fetch packet is determined.
  - PS (program address send): The address of the fetch packet is sent to memory.
  - PW (program access ready wait): A program memory access is performed.
  - PR (program fetch packet receive): The fatch packet is at the CPU boundary.
- Decode has two phases:
  - DP (instruction dispatch): The next execute packet in the fetch packet is determined and sent to the appropriate functional units to be decoded.
  - DC (instruction decode): Instructions are decoded in functional units.
- Execute has five phases:
  - E1: Execute 1.
  - E2: Execute 2.
  - E3: Execute 3.
  - E4: Execute 4.
  - E5: Execute 5.

The pipeline operation of the C62x/C64x instructions can be categorized into seven instruction types. Six of these are shown in Table 3.1, which gives a mapping of operations

Table 3.1: Execution Stage Length Description for Each Instruction Type [7]

		Instruction Type					
		Single Cycle	16 X 16 Single Multiply/ C64x .M Unit Non-Multiply	Store	C64x Multiply Extensions	Load	Branch
Execution phases	E1	Compute result and write to register	Read operands and start computations	Compute address	Reads operands and start computations	Compute address	Target-code in PG‡
	E2		Compute result and write to register	Send address and data to memory		Send address to memory	
	E3			Access memory		Access memory	
	E4				Write results to register	Send data back to CPU	
	E5					Write data into register	
Delay slots		0	1	0†	3	4†	5‡

occurring in each execution phase for the different instruction types. The delay slots associated with each instruction type are listed in the bottom row.

The execution of instructions can be defined in terms of delay slots. A delay slot is a CPU cycle that occurs after the first execution phase (E1) of an instruction. Results from instructions with delay slots are not available until the end of the last delay slot. For example, a multiply instruction has one delay slot, which means that one CPU cycle elapses before the results of the multiply are available for use by a subsequent instruction. However, results are available from other instructions finishing execution during the same CPU cycle in which the multiply is in a delay slot.

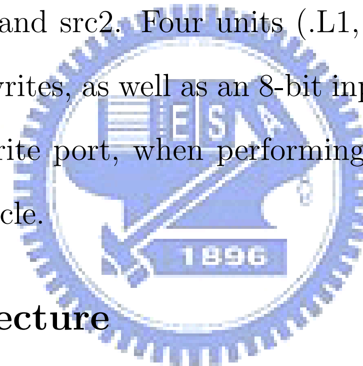
3.1.2.2 Functional Units

The eight functional units in the C6000 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 functional units are described in Table 3.2 and Table 3.3.

Besides being able to perform 32-bit operations, the C64x also contains many 8-bit to 16-bit extensions to the instruction set. For example, the MPYU4 instruction performs four 8×8 unsigned multiplies with a single instruction on an .M unit. The ADD4 instruction performs four 8-bit additions with a single instruction on an .L unit.

The data line in the CPU supports 32-bit operands, 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 (listed in Fig. 3.4). 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.



3.1.3 Memory Architecture

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

A variety of memory options are available for the C6000 platform. In our system, the memory types we can use are:

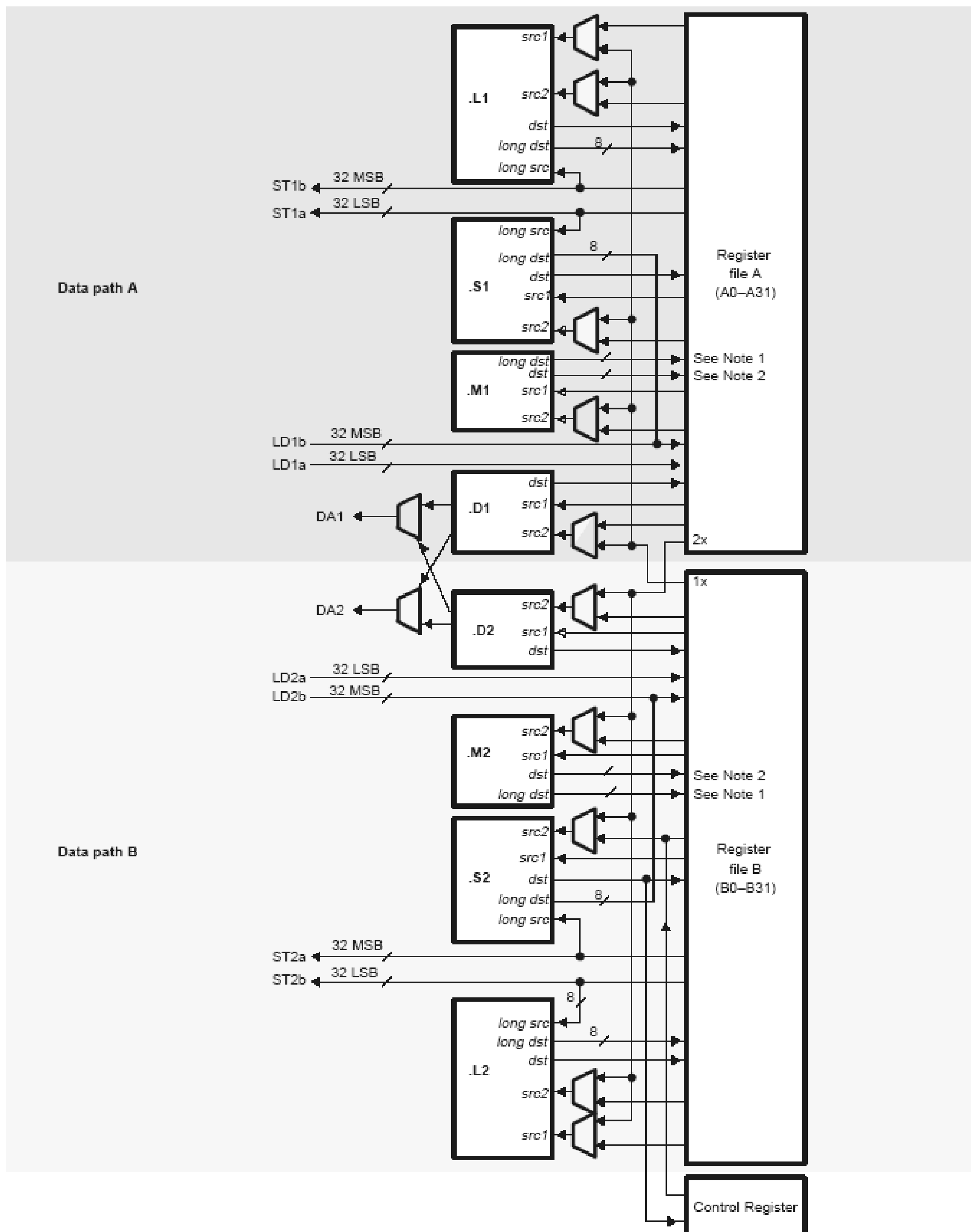
- On-chip RAM, up to 875 MB.
- Program cache.

Table 3.2: Functional Units and Operations Performed [7]

Functional Unit	Fixed-Point Operations	Floating-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	Arithmetic operations DP → SP, INT → DP, INT → SP conversion 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	Compare Reciprocal and reciprocal square-root operations Absolute value operations SP → DP conversion operations

Table 3.3: Functional Units and Operations Performed (Continued) [7]

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



Notes for .M unit:
1. *long dst* is 32 MSB
2. *dst* is 32 LSB

Figure 3.4: TMS320C64x CPU data paths [7].

- 32-bit external memory interface supports SDRAM, SBSRAM, SRAM, and other asynchronous memories.

- Two-level caches [8]. Level 1 cache is split into program (L1P) and data (L1D) cache. Each L1 cache is 16 KB. Level 2 memory is configurable and can be split into L2 SRAM (addressable on-chip memory) and L2 cache for caching external memory locations. The size of L2 is 1 MB. External memory can be several MB large. The access time depends on the memory technology used but is typically around 100 to 133 MHz. In our system, the external memory usable by the DSP is a 32 MB SDRAM.

3.2 The Code Composer Studio Development Tools [9], [10]

We now introduce the software environment used in our work. TI supports a useful GUI development tool set to DSP users for developing and debugging their projects: the Code Composer Studio (CCS). The CCS development tools are a key element of the DSP software and development tools from TI. The fully integrated development environment includes real-time analysis capabilities, easy to use debugger, C/C++ compiler, assembler, linker, editor, visual project manager, simulators, XDS560 and XDS510 emulation drivers and DSP/BIOS support.

Some of CCS's fully integrated host tools include:

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

- Integrated visual project manager with source control interface, multi-project support and the ability to handle thousands of project files.

- Source code debugger common interface for both simulator and emulator targets:

- C/C++/assembly language support.
- Simple breakpoints.
- Advanced watch window.
- Symbol browser.

- DSP/BIOS host tooling support (configure, real-time analysis and debug).
- Data transfer for real time data exchange between host and target.
- Profiler to analyze code performance.

CCS also delivers "foundation software" consisting of:

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

TI also supports some optimized DSP functions for the TMS320C64x devices: the TMS320C64x digital signal processor library (DSPLIB). This source code library includes C-callable functions (ANSI-C language compatible) for general signal processing mathematical and vector functions [11]. The routines included in the DSP library are organized as follows:

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

3.3 Code Optimization Methods [12]

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. This makes the compiler do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation, which simplifies the maintenance of the code, as everything resides in a C framework that is simple to maintain, support, and upgrade.

The recommended code development flow for the C6000 involves the phases described in Fig. 3.5. The tutorial section of the Programmer's Guide [12] focuses on phases 1 and phase 2, and the Guide also instructs the programmer about the tuning stage of phase 3. What is learned is the importance of giving the compiler enough information to fully maximize its potential. An added advantage is that this compiler provides direct feedback on the entire program's high MIPS areas (loops). Based on this feedback, there are some simple steps the programmer can take to pass complete and better information to the compiler to maximize the compiler performance.

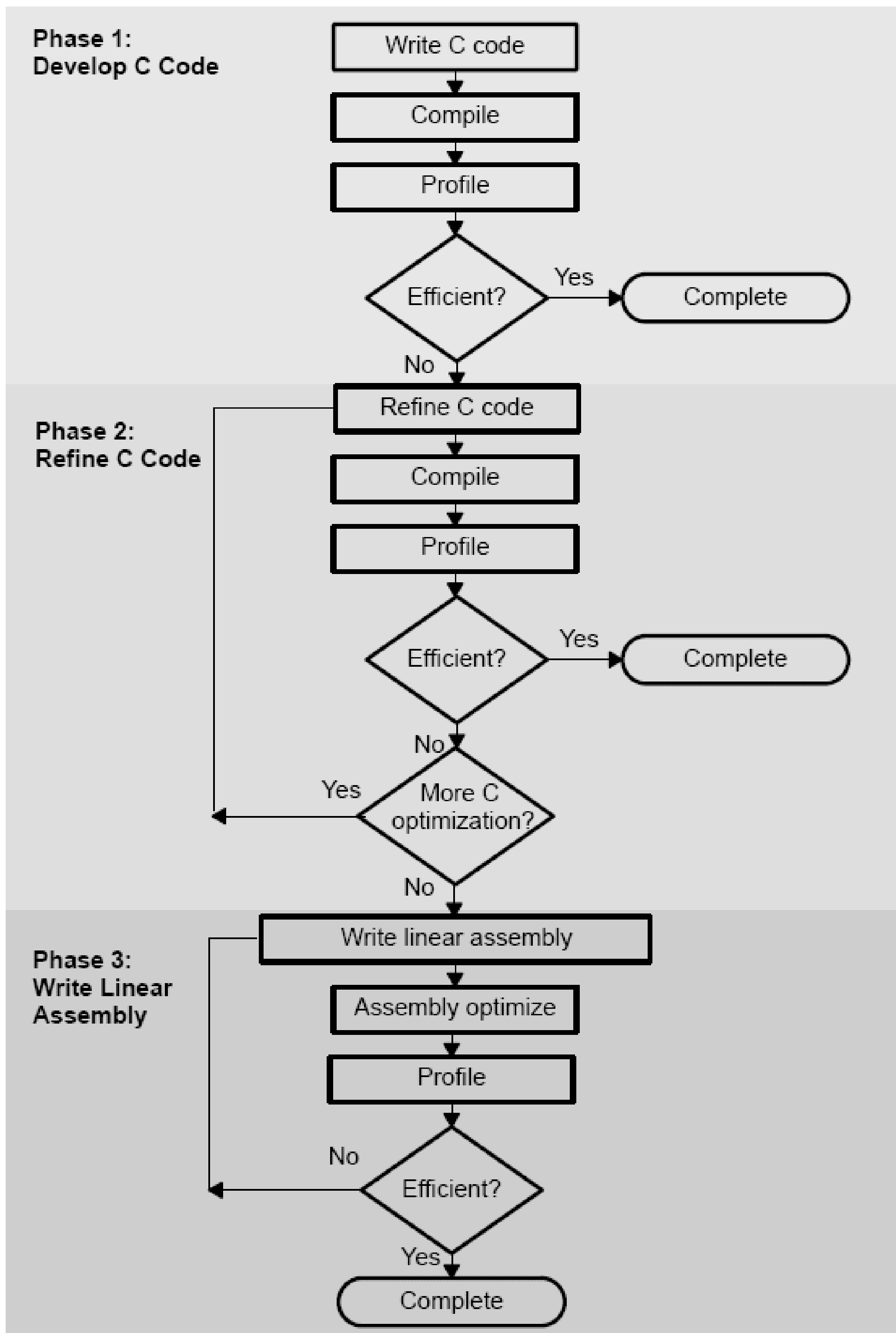


Figure 3.5: Code development flow for TI C6000 DSP [12].

The following items list the goal for each phase in the software development flow shown in Fig. 3.5.

- Developing C code (phase 1) without any knowledge of the C6000. Use the C6000 profiling tools to identify any inefficient areas that we might have in the C code. To improve the performance of the code, proceed to phase 2.
- Use techniques described in [12] to improve the C code. Use the C6000 profiling tools to check its performance. If the code is still not as efficient as we would like it to be, proceed to phase 3.
- Extract the time-critical areas from the C code and rewrite the code in linear assembly. We can use the assembly optimizer to optimize this code.

TI provides high performance C program optimization tools, and they do not suggest the programmer to code by hand in assembly. In this thesis, the development flow is stopped at phase 2. We do not optimize the code by writing linear assembly. Coding the program in high level language keeps the flexibility of porting to other platforms.

3.3.1 Compiler Optimization Options [9], [10]

The compiler supports several options to optimize the code. The compiler options can be used to optimize code size or execution performance. Our primary concern in this work is the execution performance. Hence we do not care very much about the code size. The easiest way to invoke optimization is to use the cl6x shell program, specifying the -on option on the cl6x command line, where n denotes the level of optimization (0, 1, 2, 3) which controls the type and degree of optimization:

- -o0.

- Performs control-flow-graph simplification.
- Allocates variables to registers.
- Performs loop rotation.
- Eliminates unused code.
- Simplifies expressions and statements.
- Expands calls to functions declared inline.

• -o1. Performs all -o0 optimization, and:

- Performs local copy/constant propagation.
- Removes unused assignments.
- Eliminates local common expressions.

• -o2. Performs all -o1 optimizations, and:

- Performs software pipelining.
- Performs loop optimizations.
- Eliminates global common subexpressions.
- Eliminates global unused assignments.
- Converts array references in loops to incremented pointer form.
- Performs loop unrolling.

• -o3. Performs all -o2 optimizations, and:

- Removes all functions that are never called.
- Simplifies functions with return values that are never used.
- Inlines calls to small functions.

- 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.

The -o2 is the default if -o is set without an optimization level.

The program-level optimization can be specified by using the -pm option with the -o3 option. With program-level optimization, all of the source files are compiled into one intermediate file called a module. The module moves through the optimization and code generation passes of the compiler. Because the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization:

- If a particular argument in a function always has the same value, the compiler replaces the argument with the value and passes the value instead of the argument.
- If a return value of a function is never used, the compiler deletes the return code in the function.
- If a function is not called directly or indirectly, the compiler removes the function.

When program-level optimization is selected in Code Composer Studio, options that have been selected to be file-specific are ignored. The program level optimization is the highest level optimization option. We use this option to optimize our code.

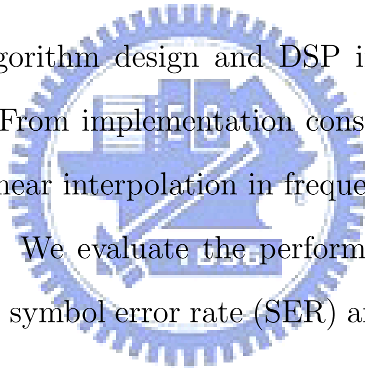
3.3.2 Using Intrinsics

The C6000 compiler provides intrinsics, which are special functions that map directly to C64x instructions, to optimize the C code performance. All instructions that are not easily

expressed in C code are supported as intrinsics. Intrinsics are specified with a leading underscore (_) and are accessed by calling them as we call a function. A table of TMS320C6000 C/C++ compiler intrinsics can be found in [12].

Chapter 4

Uplink Channel Estimation and DSP Implementation



The aim of our work is the algorithm design and DSP implementation of IEEE 802.16e OFDMA transmission system. From implementation consideration, we use simple channel estimation techniques such as linear interpolation in frequency domain and simple improvement methods in time domain. We evaluate the performance of each channel estimation method mainly by observing the symbol error rate (SER) and the mean square error (MSE).

4.1 Channel Estimation Techniques

Channel estimators in OFDMA system usually need pilot information as reference. A fading channel requires constant tracking, so pilot information has to be transmitted continuously. In general, the fading channel can be viewed as a two-dimensional (2-D) signal (time and frequency), whose values are sampled at pilot positions.

We consider three topics in this section, which are channel estimation at pilot subcarriers, interploation schemes and time-domain improvement methods. More specifically we use the least-squares (LS) technique to estimate the channel response at pilots, use linear interpolation to estimate the frequency response at nonpilot subcarriers in the frequency

domain, and consider two ways of time-domain improvement including simple average and exponential average. These are discussed seperately in the following subsections.

4.1.1 The Least-Squares (LS) Estimator

Based on the priori known data, we can estimate the channel information on pilot carriers roughly by the least-squares (LS) estimator. An LS estimator minimizes the squared error [13]

$$||\mathbf{Y} - \hat{\mathbf{H}}_{LS}\mathbf{X}||^2 \tag{4.1}$$

where $\mathbf{Y}$ is the received signal and $\mathbf{X}$ is a priori known pilots, both in the frequency domain and both being $N \times 1$ vectors where N is the FFT size. $\hat{\mathbf{H}}_{LS}$ is an $N \times N$ matrix whose values are 0 except at pilot locations m_i where $i = 0, \cdots, N_p - 1$:

$$\hat{\mathbf{H}}_{LS} = \begin{bmatrix} H_{m_0,m_0} & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & H_{m_1,m_1} & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & H_{m_2,m_2} & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & H_{m_{N_p-1},H_{m_{N_p-1}}} \end{bmatrix}. \tag{4.2}$$

Therefore, (4.1) can be rewritten as

$$[Y(m) - \hat{H}_{LS}(m)X(m)]^2, \text{ for all } m = m_i. \tag{4.3}$$

Then the estimate of pilot signals, based on only one observed OFDMA symbol, is given by

$$\hat{H}_{LS}(m) = \frac{Y(m)}{X(m)} = \frac{X(m)H(m) + N(m)}{X(m)} = H(m) + \frac{N(m)}{X(m)} \tag{4.4}$$

where $N(m)$ is the complex white Gaussian noise on subcarrier m. We collect $H_{LS}(m)$ into $\hat{\mathbf{H}}_{\mathbf{p},\mathbf{LS}}$, an $N_p \times 1$ vector where N_p is the total number of pilots, as

$$\begin{aligned} \hat{\mathbf{H}}_{p,LS} &= [H_{p,LS}(0)\ H_{p,LS}(1)\ \cdots H_{p,LS}(N_p-1)]^T \\ &= [\tfrac{Y_p(0)}{X_p(0)}, \tfrac{Y_p(1)}{X_p(1)}, \ldots, \tfrac{Y_p(N_p-1)}{X_p(N_p-1)}]^T. \end{aligned} \tag{4.5}$$

The LS estimator is a simplest channel estimator one can think of.

4.1.2 Linear Interpolation

After obtaining the channel response estimate at the pilot subcarriers, we use interpolation to obtain the response at the rest of the subcarriers. Linear interpolation is a commonly considered scheme due to its low complexity. It does the interpolation between two known data. That is, we use the channel information at two pilot subcarriers obtained by the LS estimator to estimate the channel frequency response information at the data subcarriers between them. We also use linear extrapolation to estimate the response as the data subcarriers beyond the outermost pilot subcarriers.

The channel estimatw at data subcarrier k, $mL < k < (m+1)L$, using linear interpolation is given by [14]

$$H_e(k) = H_e(m+l) = (H_p(m+1) - H_p(m))\frac{l}{L} + H_p(m) \tag{4.6}$$

where $H_p(k), k = 0, 1, \cdots, N_p$, are the channel frequency responses at pilot subcarriers, L is the pilot subcarriers spacing, and $0 \leq l < L$.

4.1.3 Time Averaging

We also consider processing the channel information along the time axis to get better estimation. Averaging several channel responses over a period of time should mitigate the influence of noise. Coherence time is a statistical measure of the time duration over which the channel impulse response is essentially invariant. It quantifies the similarity of the channel response at different times. The channel can be considered slowly varying if the coherence time is greater than the OFDMA symbol period. The channel may even be assumed to be static over one or several reciprocal Doppler spread intervals.

For example, assume the SS moves at a speed of 60 km/h. The maximum Dopper shift

with a center frequency 3.5 GHz can be calculated as

$$f_m = \frac{v}{\lambda} = 194.44 \text{ Hz}. \tag{4.7}$$

The corresponding coherence time is approximately [15]

$$T_c \approx \frac{9}{16\pi f_m} = 920.83\ \mu\text{s}. \tag{4.8}$$

Consider an OFDMA system of bandwidth 20 MHz, and using 2048-FFT and 256-point cyclic prefix. The symbol period is

$$\frac{(2048+256)}{\left(\lfloor \frac{\frac{28}{25}\cdot 20M}{8000} \rfloor \times 8000\right)} = 102.86\ \mu\text{s}. \tag{4.9}$$

Hence, the channel response over $\lfloor \frac{920.83}{102.86} \rfloor = 8$ symbols can be regarded static. Thus we may use simple averaging over 3 symbols to reduce noise effect as

$$H_{avg}(k) = \frac{H_0^{interp}(k) + H_{-1}^{interp}(k) + H_{-2}^{interp}(k)}{3} \tag{4.10}$$

where $H_n^{interp}(k)$ is the interpolated channel response at the previous nth symbol time.

If the channel remains static, over a longer time period, we may use more symbols in the averaging to reduce the noise effect more effectively. But then the storage requirement and the computational complexity both increase, a simple way to take more (or less) symbols into the average effectively and yet without the storage and complexity penalty is exponential averaging:

$$\tilde{h}_n^{exp}(f) = \begin{cases} w \cdot \tilde{h}_{n-1}^{exp}(f) + (1-w)\cdot \tilde{h}_n^{interp}(f), & n > 1, \\ \tilde{h}_n^{interp}(f), & n = 1, \end{cases} \tag{4.11}$$

where $\tilde{h}_n^{exp}(f)$ is the estimated channel after exponential averaging at nth symbol time, $\tilde{h}_n^{interp}(f)$ is the channel response by using only the interpolation discussed before at the nth symbol time, and w is the exponential factor.

Exponential averaging may yield better performance than simple moving average when the channel is very static, but its performance may degrade more significantly than that of

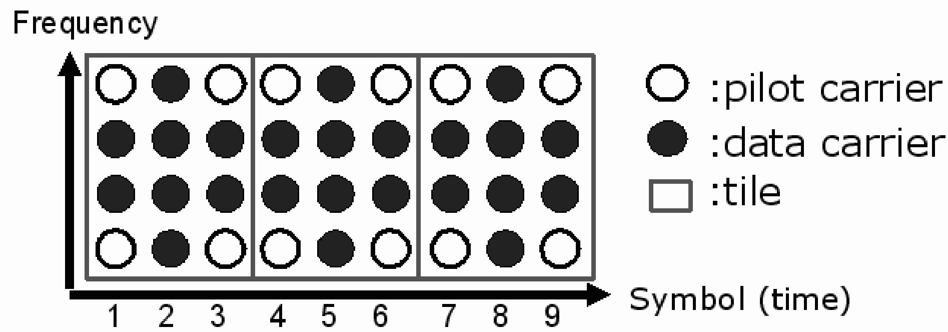


Figure 4.1: Tile structure.

moving average in fading channels. We will compare the performance at different values of w and in different conditions later.

4.1.4 Application to IEEE 802.16e OFDMA Uplink

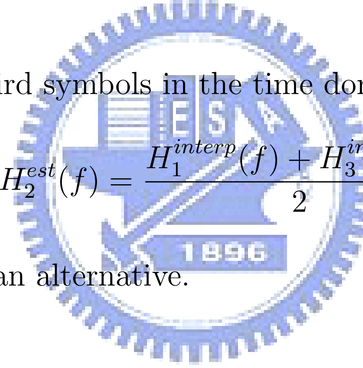
As described before in chapter 2, uplink transmission uses tile structure to transmit pilot and data information. Fig 4.1 shows an example of tile transmission. Within a tile, we first estimate the channel response at each pilot position. Second, we interpolate the frequency response at data subcarriers in symbol 1 and 3 by the estimated pilot. Last, we get the frequency response of symbol 2 by time averaging the channel response estimates of symbols 1 and 3.

We give the detail steps for channel estimation as follows:

- Estimate the channel response at each pilot location by using the LS technique.
- Use the linear interpolation scheme to get the data subcarrier response in symbols 1 and 3 from the estimated values at pilot locations.
- Estimate the channel response at middle symbol that contains no pilots in a tile by

Table 4.1: OFDMA Uplink Parameters

Parameters	Values
Bandwidth	20 MHz
Central frequency	3.5 GHz
N_{used}	1681
Sampling factor n	28/25
G	1/8
N_{FFT}	2048
Sampling frequency	22.4 MHz
Subcarrier spacing	10.94 kHz
Useful symbol time	91.43 μs
CP time	11.43 μs
OFDMA symbol time	102.86 μs
Sampling time	44.65 ns



averaging the first and third symbols in the time domain as

$$H_2^{est}(f) = \frac{H_1^{interp}(f) + H_3^{interp}(f)}{2}. \tag{4.12}$$

Exponential averaging is an alternative.

4.2 Simulation Parameters and Channel Model

This section gives the parameters and introduce the channel model used in our simulation work.

4.2.1 OFDMA Uplink System Parameters

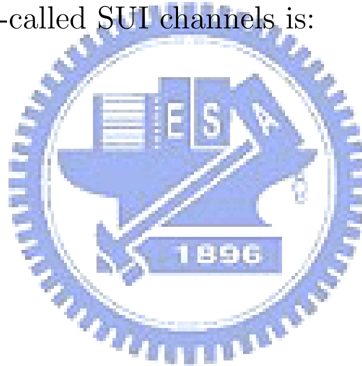
In chapter 2, we introduced the primitive and the derived parameters of the system. The system parameters used in our simulation are listed in Table 4.1.

4.2.2 Simulation Channel Model

Erceg et al [16] published a total of 6 different radio channel models for type G2 (i.e, LOS and NLOS) MMDS BWA systems in three terrain categories. The three types in suburban area are

- A: hilly terrain, heavy tree,
- C: flat terrain, light tree, and
- B: between A and C.

The correspondence with the so-called SUI channels is:



- C: SUI-1, SUI-2,
- B: SUI-3, SUI-4, and
- A: SUI-5, SUI-6.

In the above, SUI-1 and SUI-2 are Ricean multipath channels, whereas the other four are from Hari and are Rayleigh multipath channels. The Rayleigh channels are more hostile and exhibit a greater rms delay spread. And the SUI-2 represents a worst case link for terrain type C. We employ SUI-2 and SUI-3 model in our simulation, but we use Rayleigh fading to model all the paths in these channels. The channel charateristics are as shown in Table 4.2.

4.3 Simulation Results

4.3.1 Simulation Flow

Figure 4.2 illustrates our simulated system. We assume perfect synchronization and omit it in our simultion. After channel estimation, we calculate the MSE between the real channel

Table 4.2: Channel Profiles of SUI-2 and SUI-3 [16]

SUI – 2 Channel				
	Tap 1	**Tap 2**	**Tap 3**	**Units**
Delay	0	0.4	1.1	μs
Power (omni ant.)	0	-12	-15	dB
90% K-fact. (omni)	2	0	0	
75% K-fact. (omni)	11	0	0	
Power (30° ant.)	0	-18	-27	dB
90% K-fact. (30°)	8	0	0	
75% K-fact. (30°)	36	0	0	
Doppler	0.2	0.15	0.25	Hz
Antenna Correlation: $\rho_{ENV} = 0.5$ **Gain Reduction Factor:** GRF = 2 dB **Normalization Factor:** $F_{omni} = -0.3930$ dB, $F_{30°} = -0.0768$ dB		**Terrain Type:** C **Omni antenna:** $\tau_{RMS} = 0.202$ μs, overall K: K = 1.6 (90%); K = 5.1 (75%) **30° antenna:** $\tau_{RMS} = 0.069$ μs, overall K: K = 6.9 (90%); K = 21.8 (75%)		

SUI – 3 Channel				
	Tap 1	**Tap 2**	**Tap 3**	**Units**
Delay	0	0.4	0.9	μs
Power (omni ant.)	0	-5	-10	dB
90% K-fact. (omni)	1	0	0	
75% K-fact. (omni)	7	0	0	
Power (30° ant.)	0	-11	-22	dB
90% K-fact. (30°)	3	0	0	
75% K-fact. (30°)	19	0	0	
Doppler	0.4	0.3	0.5	Hz
Antenna Correlation: $\rho_{ENV} = 0.4$ **Gain Reduction Factor:** GRF = 3 dB **Normalization Factor:** $F_{omni} = -1.5113$ dB, $F_{30°} = -0.3573$ dB		**Terrain Type:** B **Omni antenna:** $\tau_{RMS} = 0.264$ μs, overall K: K = 0.5 (90%); K = 1.6 (75%) **30° antenna:** $\tau_{RMS} = 0.123$ μs, overall K: K = 2.2 (90%); K = 7.0 (75%)		

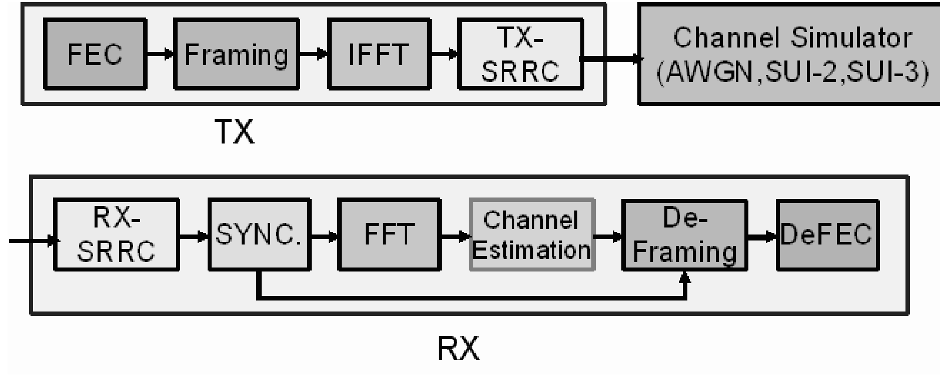


Figure 4.2: Block diagram of the simulated system.

and the estimated one, where the average is taken over the subcarriers. The symbol error rate (SER) can also be obtained after demapping.

4.3.2 Validation of Simulation Model

Before considering multipath channels, we do simulation with an AWGN channel to validate the simulation model. We validate the model by comparing theoretical SER curves and the SER curves resulting from simulations.

For an even number of bits per symbol, the SER of rectangular QAM is given by

$$P_s = 4\left(1 - \frac{1}{\sqrt{M}}\right) Q\left(\sqrt{\frac{3}{M-1}\frac{E_s}{N_0}}\right) \tag{4.13}$$

where

- M = number of symbols in modulation constellation; for example, M = 4 for QPSK, M = 16 for 16QAM and M = 64 for 64QAM,
- E_s = average symbol energy,
- N_0 = noise power spectral density (W/Hz), and
- $Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-t^2/2} dt,\ x \geq 0.$

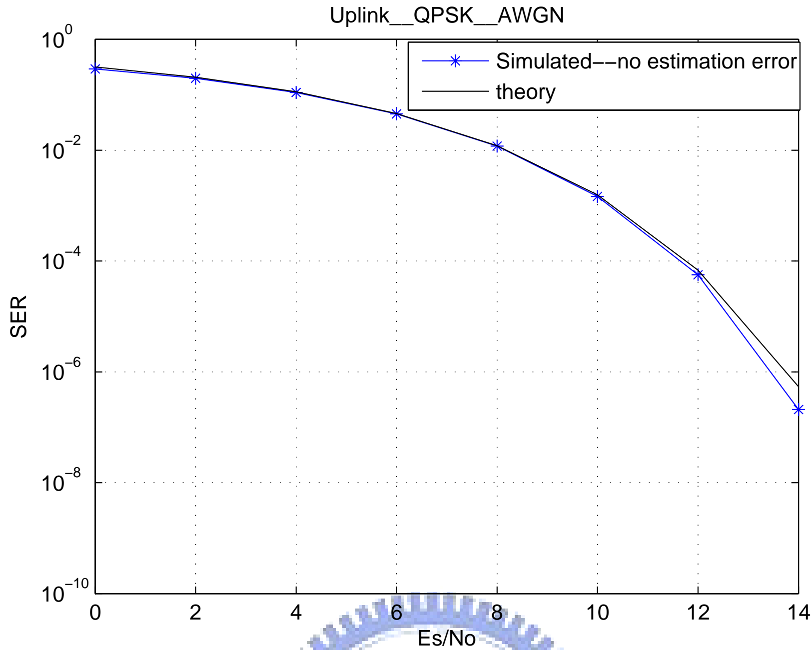


Figure 4.3: The SER curve for uncoded QPSK resulting from simulation matches the theoretical one.

In Figure 4.3, the theoretical symbol error rate (SER) curve versus E_s/N_0 for uncoded QPSK is plotted together with the SER curve resulting from the simulation. In this figure, we simulate for no channel estimation error. This validates the simulation (we use C/C++ programming language and TI's code composer studio).

4.3.3 Floating-point Simulation

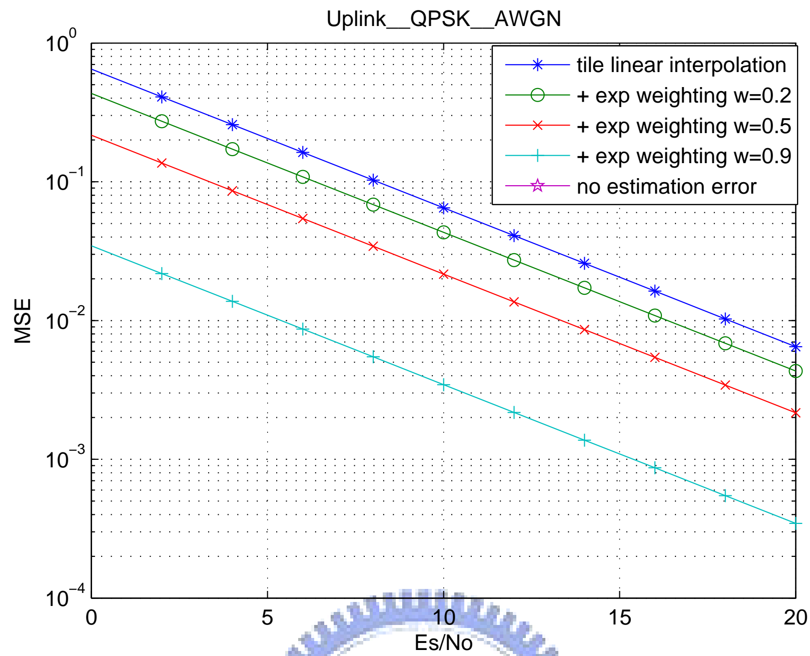
In our simulation, we assume using 10 subchannels to transmit. Figure 4.4 shows the performance of tile linear interpolation with different exponential weighting in AWGN. The method of with weighting $w = 0.9$ has the best SER and MSE. But in the condition of SUI-2, velocity being 60, this becomes the worst situation in both SER and MSE (see Fig. 4.5). It is because in multipath such as SUI-2, the variance of channel condition is much violent than in AWGN. We also get the validation from the analysis of in given velocity, calculating

the MSE by using the variance of Bessel function. Therefore, using exponential weighting of previous tile can not help estimate validly. Figure 4.6 shows tile linear interpolation with exponential weighting 0.9 in SUI-2 with different velocity of QPSK. We use no tile exponential averaging in following work.

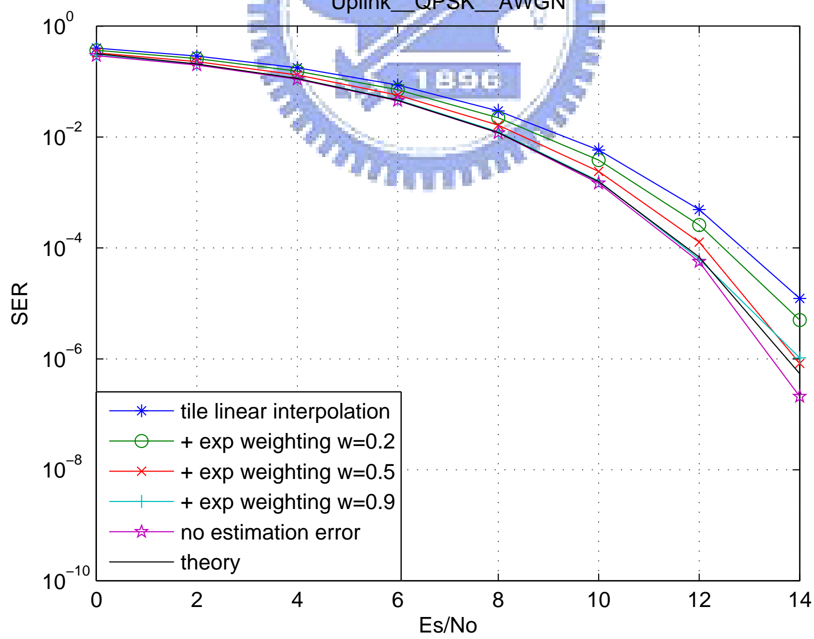
Figure 4.7 illustrates tile linear interpolation with different modulations (uncoded QPSK, 16QAM and 64QAM) in AWGN. We compare our simulation results with theory and no estimation error curves in SER (Fig. 4.7(b)). Figure 4.8 shows tile linear interpolation compared with another theory curve which takes data MSE into consideration in AWGN. Figure 4.7(a) shows the MSE curves of these three modulation types. The three lines match with each other as a straight line with slope $m = -1$. The results of MSE are unrelated to the modulation type because the pilots are BPSK modulated in each modulation case. And the channel response is interpolated only using the pilot information.[17]

The simulation of tile linear interpolation with different velocity and different modulation in SUI-2 is given in Fig. 4.9, and Fig. 4.10 gives only QPSK in SUI-3.

Figure 4.11 illustrates tile linear interpolation with different used number of subchannels in AWGN and SUI-2. We use 60 tiles to transmit in occupying 10 subchannels while using 120 tiles for 20 subchannels.

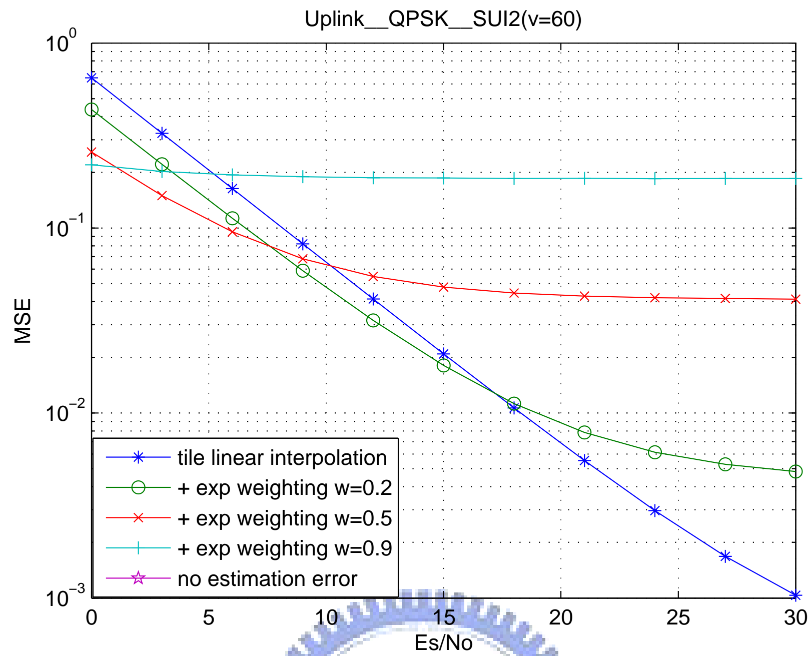


(a)

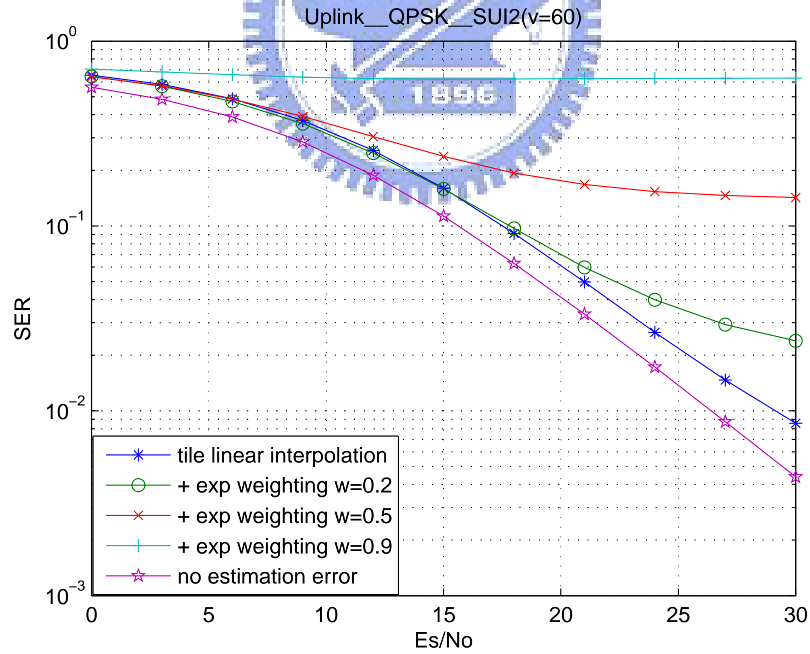


(b)

Figure 4.4: Tile linear interpolation with different exponential weighting in AWGN with QPSK. (a) MSE. (b) SER.

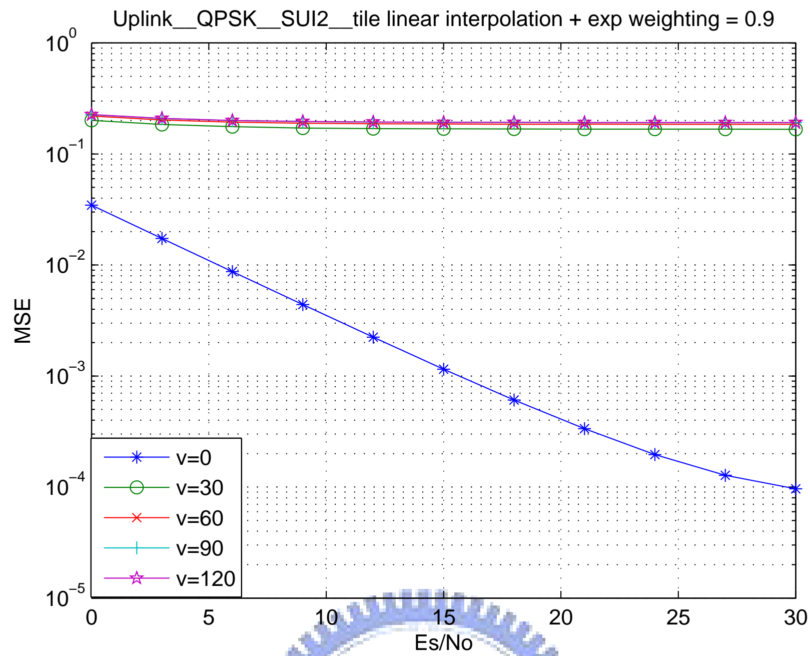


(a)

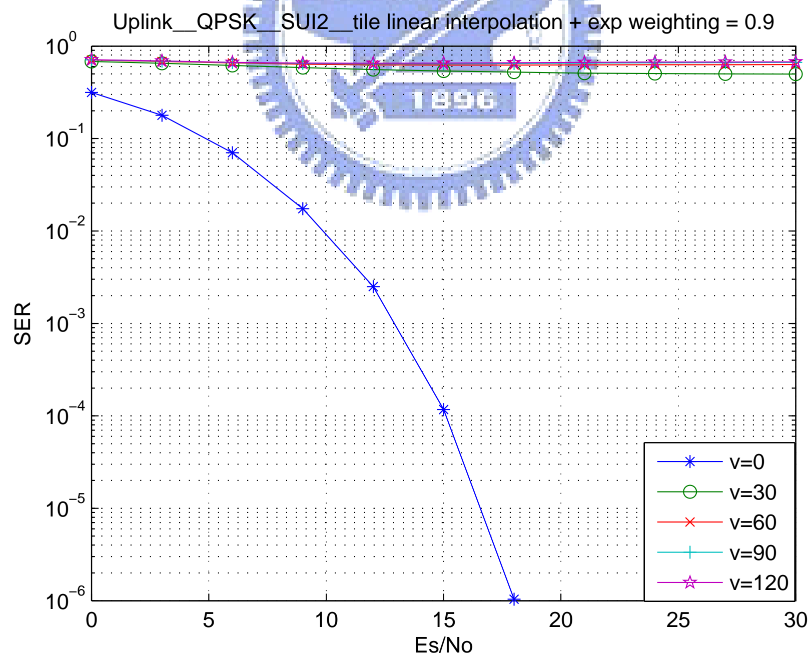


(b)

Figure 4.5: Tile linear interpolation with different exponential weighting in SUI-2 with velocity v=60 km/hr with QPSK. (a) MSE. (b) SER.

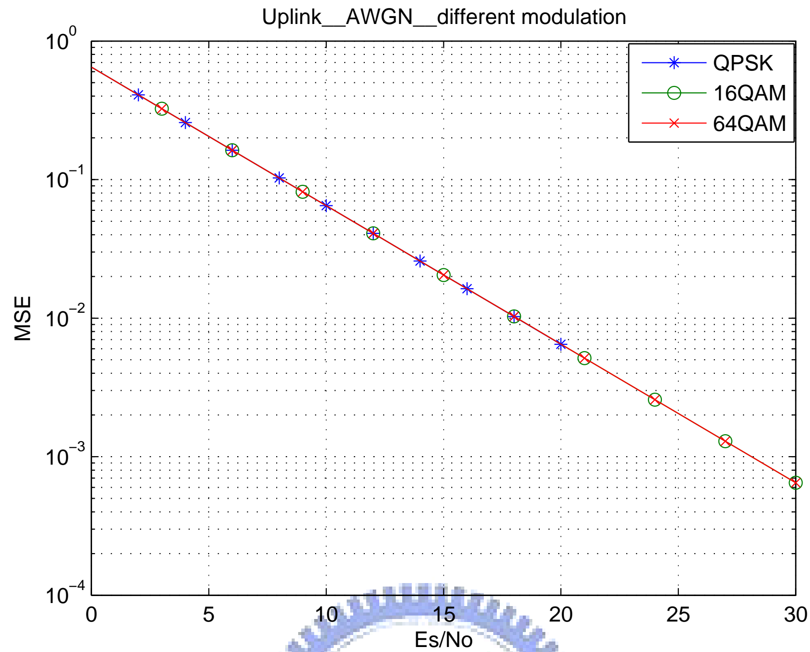


(a)

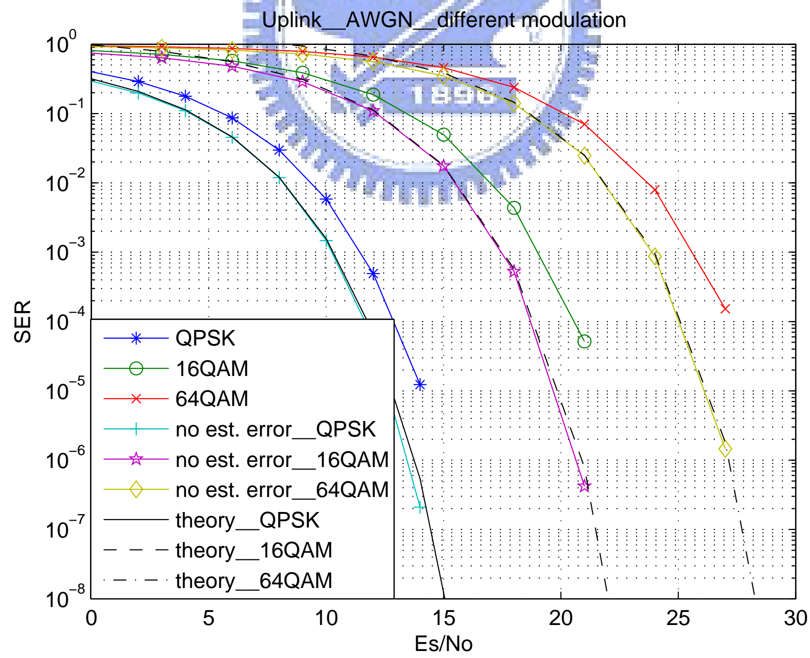


(b)

Figure 4.6: Tile linear interpolation of exponential weighting 0.9 with different velocities in SUI-2 with QPSK. (a) MSE. (b) SER.

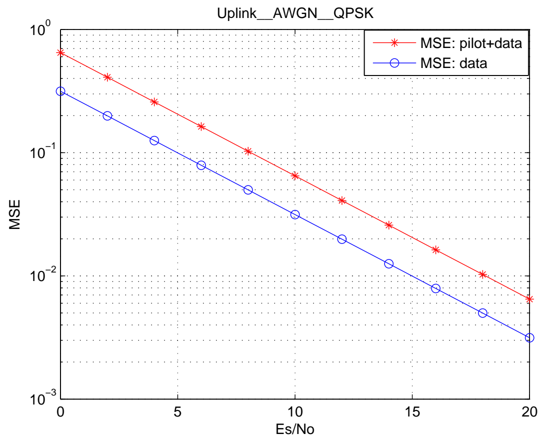


(a)

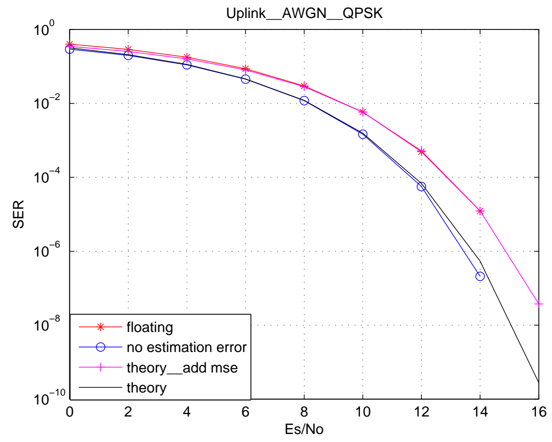


(b)

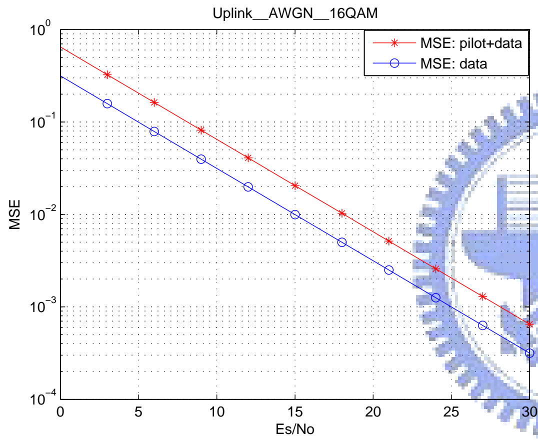
Figure 4.7: Tile linear interpolation with different modulations in AWGN. (a) MSE. (b) SER.



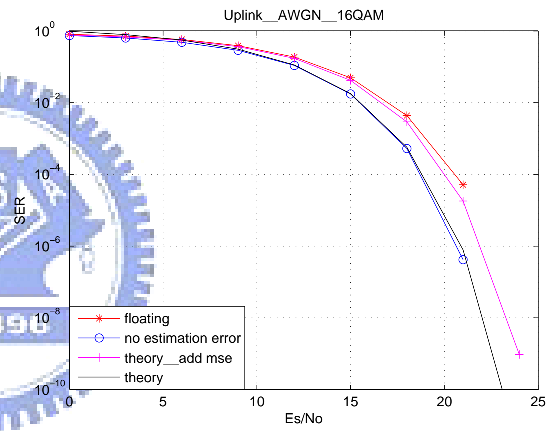
(a)



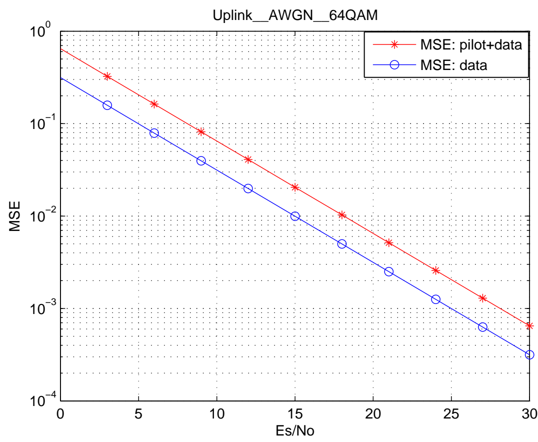
(b)



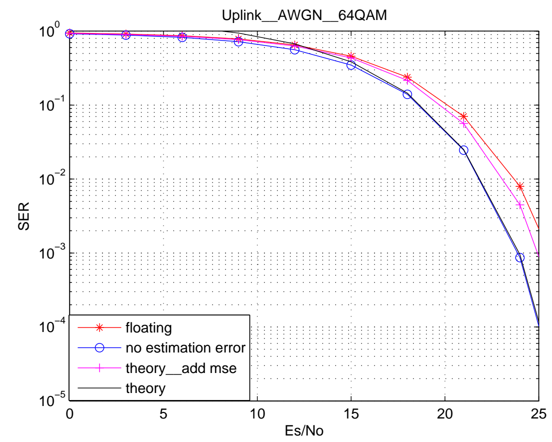
(c)



(d)

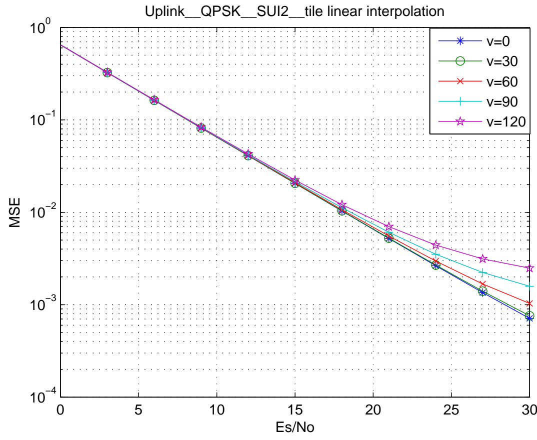


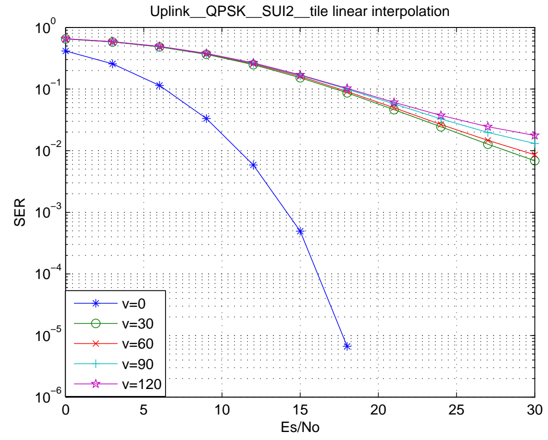
(e)



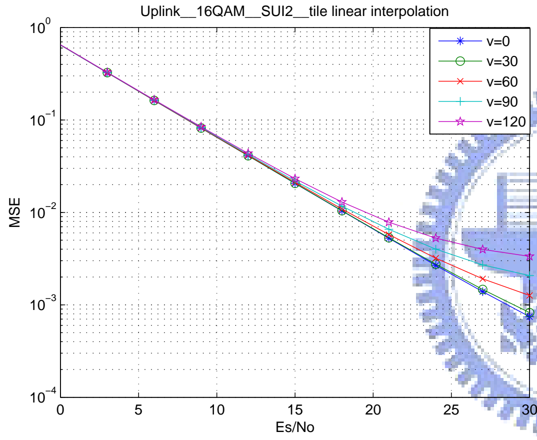
(f)

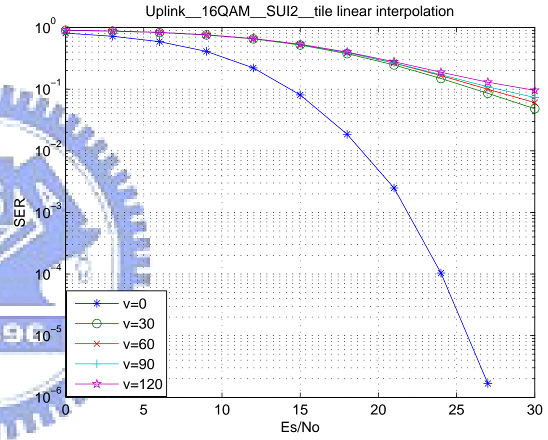
Figure 4.8: Tile linear interpolation compared with theory adding data MSE in AWGN: (a),(b) QPSK. (c),(d) 16QAM. (e),(f) 64QAM.



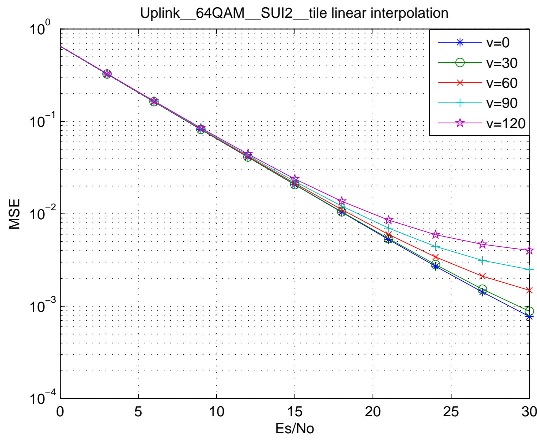


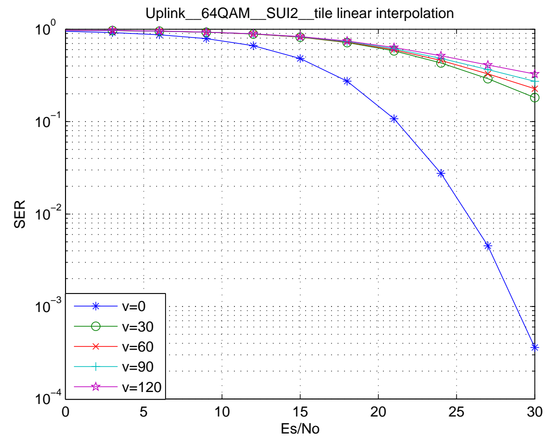
(a) (b)





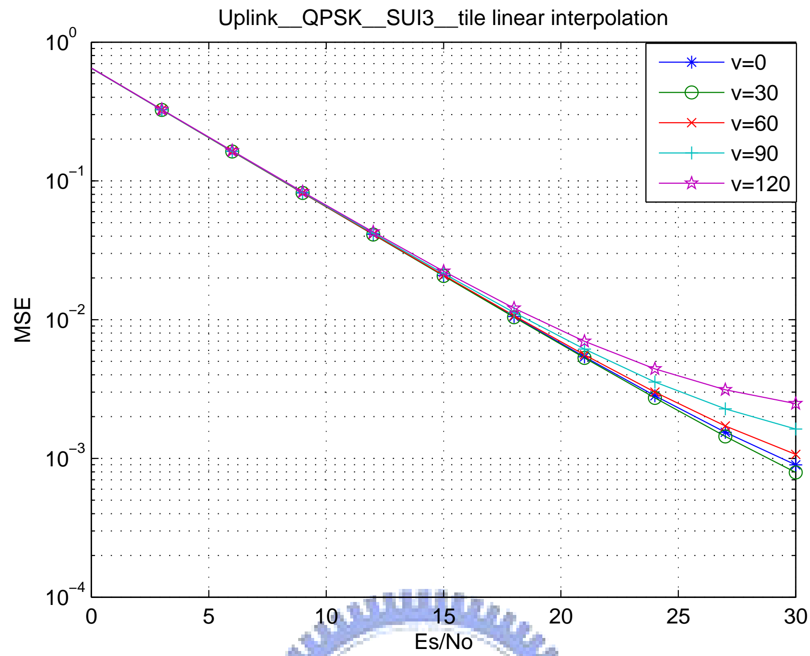
(c) (d)



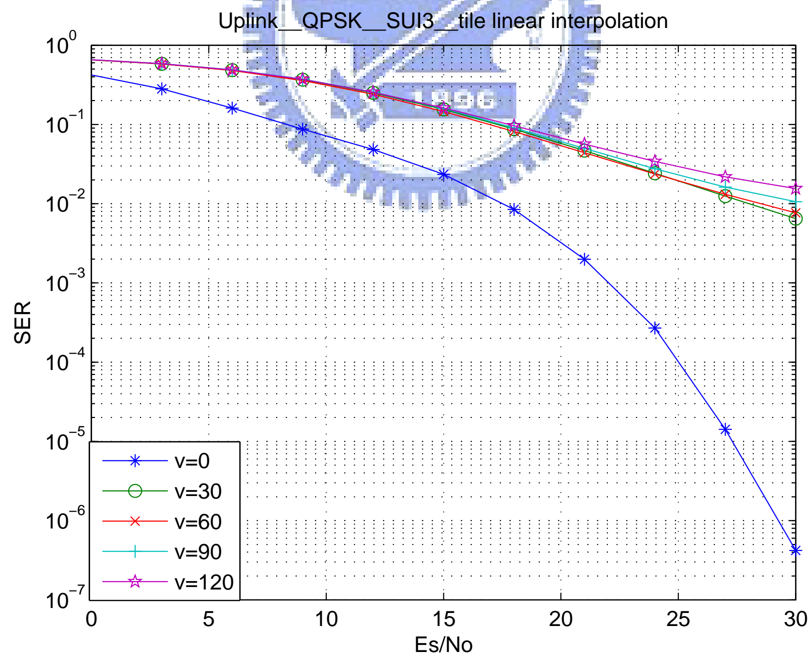


(e) (f)

Figure 4.9: Tile linear interpolation with different velocity and different modulations in SUI-2. (a),(b) QPSK. (c),(d) 16QAM. (e),(f) 64QAM.



(a)



(b)

Figure 4.10: Tile linear interpolation with different velocities in SUI-3 with QPSK. (a) MSE. (b)SER.

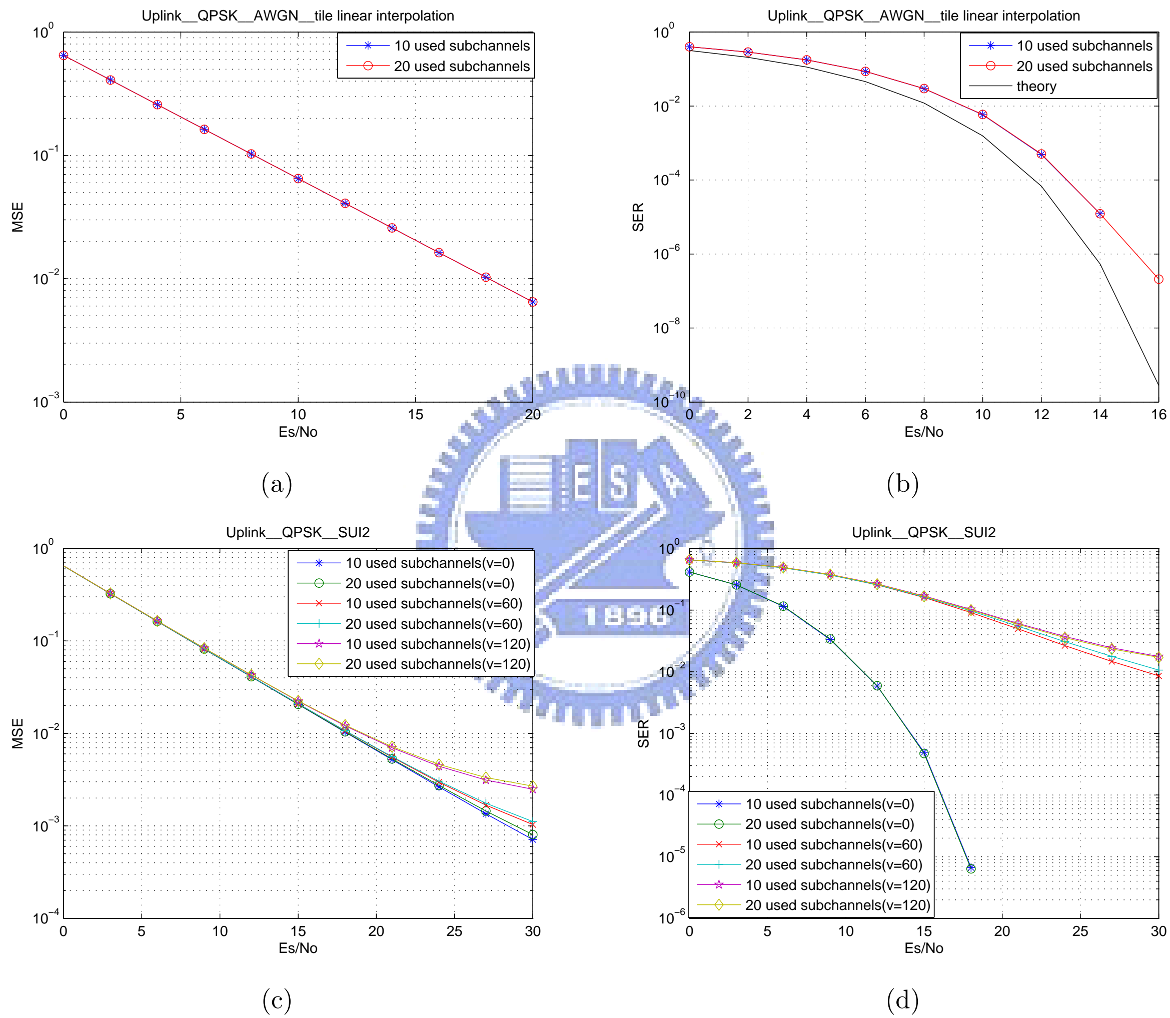


Figure 4.11: Tile linear interpolation with different used subchannels. (a),(b) AWGN. (c),(d) SUI-2.

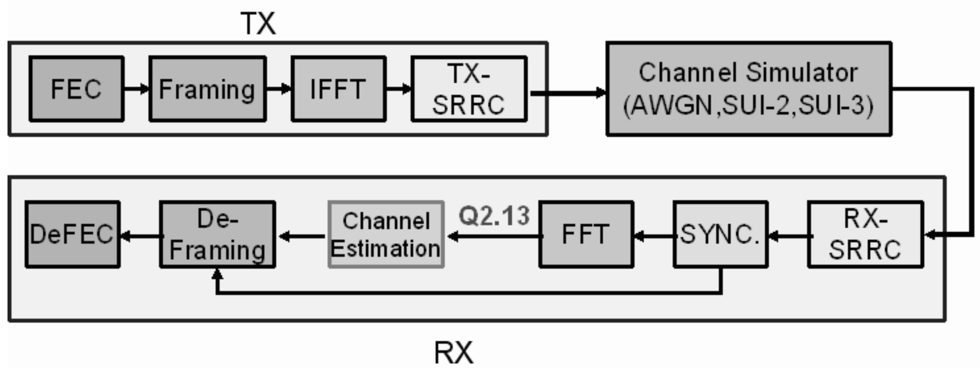


Figure 4.12: Fixed-point data format in our design.

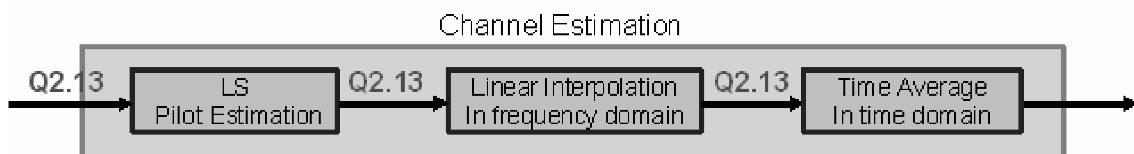


Figure 4.13: Fixed-point formats in channel estimation of our design.

4.4 DSP Implementation

4.4.1 Fixed-Point Data Formats

In algorithm development, it is often convenient to employ floating-point computation to acquire better accuracy. However, for the sake of power consumption, execution speed, and hardware costs, practical implementations usually adopt fixed-point computations. The DSP chip used in our work, TI's TMS320C6416 is also of the fixed-point category. It means that fixed-point computations are executed more efficiently than floating-point ones on this platform. Due to these facts, we do simulation in 16-bit fixed-point domain. Meanwhile, compared with 32-bit computation, it has better efficiency and negligible accuracy loss. Although fixed operation has less accuracy , it does have much shorter the executing time.

In our simulation, we try several kinds of data formats to simulate the fixed-point compu-

tation. We find out that Q2.13 is relatively close to our results of floating-point computation. Q2.13 means a 16-bit fixed point with one sign bit, 2 integer bits, and then 13 fractional bits at the right side of dot. Here we only focus on the "channel estimation" function. Therefore, we only translate the input to channel estimation into fixed for simplicity. The fixed-point data formats used in our design based on linear frequency-domain interpolation are as shown in Fig. 4.12. The detail data formats of channel estimation are also illustrated in Fig. 4.13.

4.4.2 Fixed-Point Simulation

Figure 4.14 illustrates the performance of fixed-operation compared with floating operation in AWGN and SUI-2. As we can see, there is almost no difference between the two.

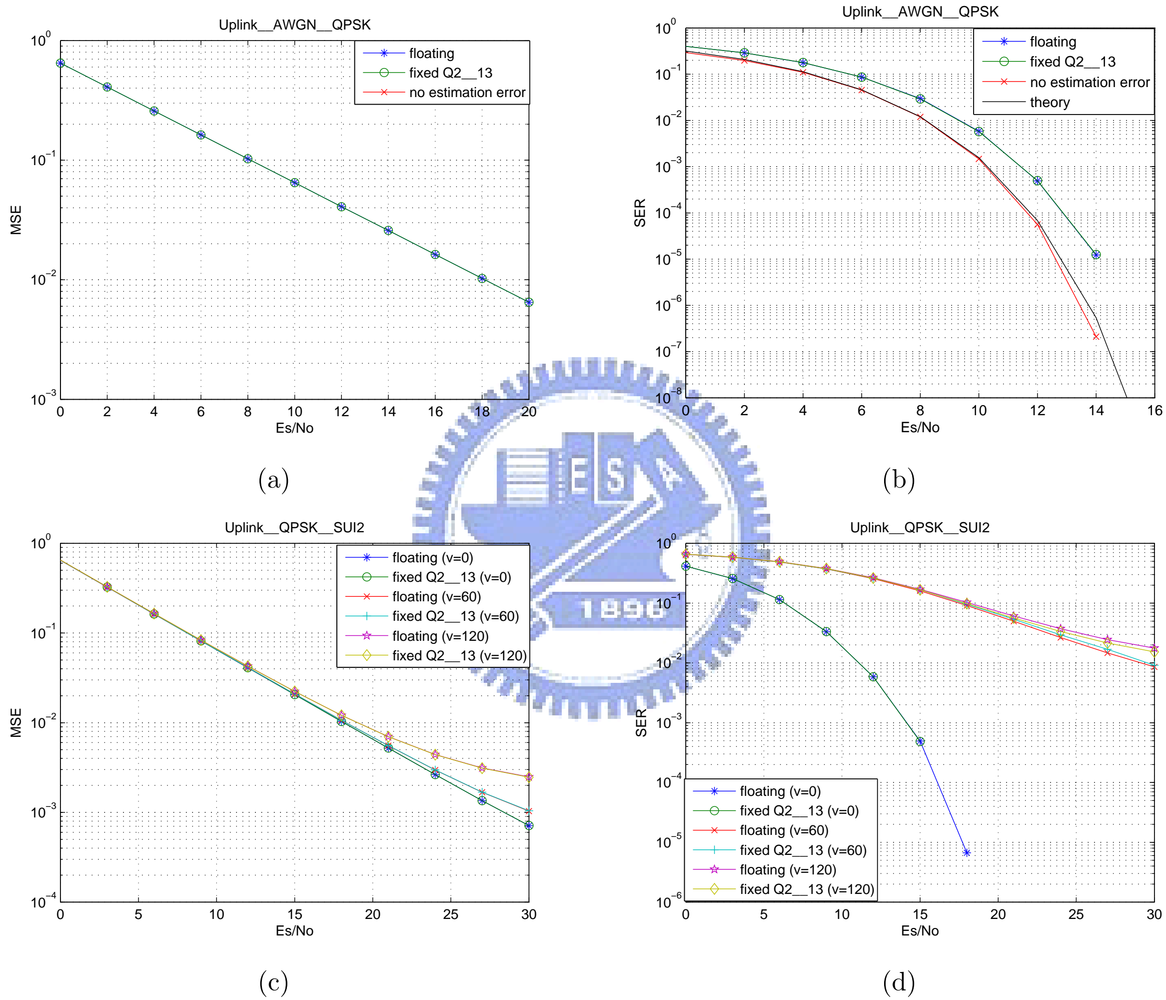


Figure 4.14: Performance of fixed-point computation of tile linear interpolation (10 used subchannels) compared to floating-point computation (a),(b) AWGN. (c),(d) SUI-2.

Table 4.3: OFDMA Uplink DSP Loading

Condition	Cycle count	DSP loading for channel estimation
1024-FFT, BW: 10 MHz 10 subchannels	4576	0.015
2048-FFT, BW: 20 MHz 10 subchannels	4800	0.016
2048-FFT, BW: 20 MHz 20 subchannels	9224	0.031

4.4.3 DSP Computational Load

The last part of our work is to do DSP implementation. We use CCS to simulate. In the condition of 2048-FFT using 10 subchannels, it takes 4800 cycles to complete the channel estimation job when executing on CCS. Since a tile contains 3 symbols, it equals 1600 cycles to be taken for per symbol. The DSP we use is C6416T and its processor clock rate is 1 GHz with 256 MB DRAM. As the BW is 20MHz and target symbol time is 102.86 μsec, it may take approximately 0.016 DSP computational load.

We also compare with two other conditions: 1024-FFT using 10 subchannels (BW: 10MHz, symbol time: 102.86 μs) and 2048-FFT using 20 subchannels. Table 4.3 illustrates the needed cycles, and we can find that the cycle count depends on the number of used subchannels. We transmit twice number of tiles when using 20 subchannels than only using 10 subchannels. It means the complexity is almost twice. Therefore, the needed DSP loading is almost twice.

```
#define Q1_14 short
#define Q2_13 short
// Though same type (short), using different types to declare is much more clear
// and avoid some mistakes

#define ftoQ1_14(A) (short) (A*16384)    // float to Q1.14 (2^14=16384)
#define ftoQ2_13(A) (short) (A*8192 )    // float to Q2.13 (2^13=8192)
#define ftoQ3_12(A) (short) (A*4096 )    // float to Q3.12 (2^12=4096)

#define Q1_14tof(A) (((float) A)/16384)  // Q1.14 to float
#define Q2_13tof(A) (((float) A)/8192)   // Q2.13 to float
#define Q3_12tof(A) (((float) A)/4096)   // Q3.12 to float
```

Figure 4.15: *FIXED.H.*

4.5 Appendix

Fig. 4.15 shows the header file *FIXED.H* which we use to transform into the formats of fixed-point. Function *channel_estimation_FIXED* is the main function of channel estimation. It contains two subfunctions of *pilot_extraction_FIXED* and *interpolation_FIXED*. Function *pilot_extraction_FIXED* gets the channel response at pilot subcarriers by using the LS technique, and function *interpolation_FIXED* does the interpolation part which plays an important role in the channel estimation scheme and also estimate the frequency response of the middle symbol within a tile. The original codes are shown in Fig. 4.16, Fig. 4.17, and Fig. 4.18. The corresponding assembly codes of function *channel_estimation_FIXED* and *interpolation_FIXED* are also listed in Figures 4.19 and 4.20. Software pipelining information of function *channel_estimation_FIXED* is illustrated in Fig. 4.21.

```
void channel_estimation_FIXED( short N_pilot, short *data0, short *data1,
                               short *pilot,  short *wk_PUSC,
                               Q2_13 *tile_ChannelOut0_Re,  Q2_13 *tile_ChannelOut0_Im,
                               Q2_13 *tile_ChannelOut1_Re,  Q2_13 *tile_ChannelOut1_Im,
                               Q2_13 *tile_ChannelOut2_Re,  Q2_13 *tile_ChannelOut2_Im,
                               Q2_13 *tile_ChannelResp0_Re,Q2_13 *tile_ChannelResp0_Im,
                               Q2_13 *tile_ChannelResp1_Re,Q2_13 *tile_ChannelResp1_Im,
                               Q2_13 *tile_ChannelResp2_Re,Q2_13 *tile_ChannelResp2_Im  )
{
 pilot_extraction_FIXED( N_pilot, pilot, wk_PUSC, tile_ChannelOut0_Re, tile_ChannelOut0_Im,
                         tile_ChannelOut2_Re,   tile_ChannelOut2_Im, tile_ChannelResp0_Re,
                         tile_ChannelResp0_Im,tile_ChannelResp2_Re,tile_ChannelResp2_Im    );

 interpolation_FIXED(N_pilot, data0, data1, pilot, tile_ChannelResp0_Re,tile_ChannelResp0_Im,
                     tile_ChannelResp1_Re,tile_ChannelResp1_Im,
                     tile_ChannelResp2_Re,tile_ChannelResp2_Im                            );
}
```

Figure 4.16: Function *channel_estimation_FIXED*.

```
void pilot_extraction_FIXED(short N_pilot, short *pilot, short *wk_PUSC,
                            Q2_13 *tile_ChannelOut0_Re,  Q2_13 *tile_ChannelOut0_Im,
                            Q2_13 *tile_ChannelOut2_Re,  Q2_13 *tile_ChannelOut2_Im,
                            Q2_13 *tile_ChannelResp0_Re, Q2_13 *tile_ChannelResp0_Im,
                            Q2_13 *tile_ChannelResp2_Re, Q2_13 *tile_ChannelResp2_Im  )
{
   short i,temp,location;

   for(i=0;i<N_pilot;i++)
   {
     location=pilot[i];
     //load pilot_i's location in carrier distrbution
     temp=1-2*wk_PUSC[location];
     //  =2*(0.5-wk_PUSC[pilot[i]])
     //  =1 or -1

     tile_ChannelResp0_Re[location]=tile_ChannelOut0_Re[location]*temp;
     tile_ChannelResp0_Im[location]=tile_ChannelOut0_Im[location]*temp;
     tile_ChannelResp2_Re[location]=tile_ChannelOut2_Re[location]*temp;
     tile_ChannelResp2_Im[location]=tile_ChannelOut2_Im[location]*temp;
   }
}
```

Figure 4.17: Function *pilot_extraction_FIXED*.

```
void interpolation_FIXED(short N_pilot, short *data0, short *data1, short *pilot,
                         Q2_13 *tile_ChannelResp0_Re, Q2_13 *tile_ChannelResp0_Im,
                         Q2_13 *tile_ChannelResp1_Re, Q2_13 *tile_ChannelResp1_Im,
                         Q2_13 *tile_ChannelResp2_Re, Q2_13 *tile_ChannelResp2_Im   )
{
    Q1_14 spacing;
    Q2_13 spacing_real, spacing_imag, delta_real, delta_imag, delta2_real, delta2_imag;
    short i,PilotLocOdd,PilotLocEven,DataLocOdd,DataLocEven,DataLoc1;
    spacing=ftoQ1_14(0.333333333);  // /3=*(1/3)

    for(i=0;i<(N_pilot/2);i++)
    {
        PilotLocOdd=pilot[2*i+1];
        PilotLocEven=pilot[2*i];
        DataLocOdd=data0[2*i+1];
        DataLocEven=data0[2*i];

        //======================data0======================//
        spacing_real=tile_ChannelResp0_Re[PilotLocOdd]-tile_ChannelResp0_Re[PilotLocEven];
        delta_real = ( spacing_real * spacing ) >>14;// /16384;
        //Q2_13*Q1_14-->Q2_13
        spacing_imag=tile_ChannelResp0_Im[PilotLocOdd]-tile_ChannelResp0_Im[PilotLocEven];
        delta_imag = ( spacing_imag * spacing ) >>14;
        //Q2_13*Q1_14-->Q2_13
        delta2_real=2*delta_real;
        delta2_imag=2*delta_imag;
        tile_ChannelResp0_Re[DataLocEven]=tile_ChannelResp0_Re[PilotLocEven] + delta_real;
        tile_ChannelResp0_Im[DataLocEven]=tile_ChannelResp0_Im[PilotLocEven] + delta_imag;
        tile_ChannelResp0_Re[DataLocOdd]= tile_ChannelResp0_Re[PilotLocEven] + delta2_real;
        tile_ChannelResp0_Im[DataLocOdd]= tile_ChannelResp0_Im[PilotLocEven] + delta2_imag;

        //========================data2=========================//
        spacing_real=tile_ChannelResp2_Re[PilotLocOdd]- tile_ChannelResp2_Re[PilotLocEven];
        delta_real= ( spacing_real * spacing ) >>14;
        //Q2_13*Q1_14-->Q2_13
        spacing_imag=tile_ChannelResp2_Im[PilotLocOdd]- tile_ChannelResp2_Im[PilotLocEven];
        delta_imag= (spacing_imag * spacing) >>14;
        //Q2_13*Q1_14-->Q2_13
        delta2_real=2*delta_real;
        delta2_imag=2*delta_imag;
        tile_ChannelResp2_Re[DataLocEven]=tile_ChannelResp2_Re[PilotLocEven] + delta_real;
        tile_ChannelResp2_Im[DataLocEven]=tile_ChannelResp2_Im[PilotLocEven] + delta_imag;
        tile_ChannelResp2_Re[DataLocOdd]= tile_ChannelResp2_Re[PilotLocEven] + delta2_real;
        tile_ChannelResp2_Im[DataLocOdd]= tile_ChannelResp2_Im[PilotLocEven] + delta2_imag;

        //========================data1=========================//
        DataLoc1=data1[4*i];

        tile_ChannelResp1_Re[DataLoc1]=
            (tile_ChannelResp0_Re[DataLoc1]+tile_ChannelResp2_Re[DataLoc1])>>1;
        tile_ChannelResp1_Im[DataLoc1]=
            (tile_ChannelResp0_Im[DataLoc1] + tile_ChannelResp2_Im[DataLoc1]) >>1;
        tile_ChannelResp1_Re[DataLoc1+1]=
            (tile_ChannelResp0_Re[DataLoc1+1] + tile_ChannelResp2_Re[DataLoc1+1]) >>1;
        tile_ChannelResp1_Im[DataLoc1+1]=
            (tile_ChannelResp0_Im[DataLoc1+1] + tile_ChannelResp2_Im[DataLoc1+1]) >>1;
        tile_ChannelResp1_Re[DataLoc1+2]=
            (tile_ChannelResp0_Re[DataLoc1+2] + tile_ChannelResp2_Re[DataLoc1+2]) >>1;
        tile_ChannelResp1_Im[DataLoc1+2]=
            (tile_ChannelResp0_Im[DataLoc1+2] + tile_ChannelResp2_Im[DataLoc1+2]) >>1;
        tile_ChannelResp1_Re[DataLoc1+3]=
            (tile_ChannelResp0_Re[DataLoc1+3] + tile_ChannelResp2_Re[DataLoc1+3]) >>1;
        tile_ChannelResp1_Im[DataLoc1+3]=
            (tile_ChannelResp0_Im[DataLoc1+3] + tile_ChannelResp2_Im[DataLoc1+3]) >>1;
    }
}
```

Figure 4.18: Function *interpolation_FIXED*.

```
;******************************************************************************
;* FUNCTION NAME: _channel_estimation_FIXED                                   *
;*                                                                            *
;*   Regs Modified     : A0,A1,A3,A4,A5,A6,A7,A8,A9,A10,A12,B0,B3,B4,B5,B6,B7,*
;*                           B8,B9,B10,SP,FP,A16,A17,A18,A19,A20,A21,A22,A23, *
;*                           A24,A25,A26,A27,A28,A29,A30,B16,B17,B18,B19,B20, *
;*                           B21,B22,B23,B24,B25,B26,B27,B28                  *
;*   Regs Used         : A0,A1,A3,A4,A5,A6,A7,A8,A9,A10,A12,B0,B3,B4,B5,B6,B7,*
;*                           B8,B9,B10,B12,DP,SP,FP,A16,A17,A18,A19,A20,A21,  *
;*                           A22,A23,A24,A25,A26,A27,A28,A29,A30,B16,B17,B18, *
;*                           B19,B20,B21,B22,B23,B24,B25,B26,B27,B28          *
;*   Local Frame Size  : 0 Args + 0 Auto + 20 Save = 20 byte                  *
;******************************************************************************
;* Using -g (debug) with optimization (-o3) may disable key optimizations!    *
;******************************************************************************
_channel_estimation_FIXED:
;** --------------------------------------------------------------------------*
        .dwcfa  0x0e, 0
        .dwcfa  0x09, 126, 19
           MV      .L1X    SP,FP             ; |22|
||         STW     .D2T1   FP,*SP--(24)      ; |22|
        .dwcfa  0x80, 32, 0
           STW     .D2T2   B10,*+SP(20)
        .dwcfa  0x80, 26, 1
           STW     .D2T2   B3,*+SP(16)
        .dwcfa  0x80, 19, 2
           STW     .D1T1   A12,*-FP(12)
        .dwcfa  0x80, 12, 3
           STW     .D1T1   A10,*-FP(16)
        .dwcfa  0x80, 10, 4

           LDW     .D1T2   *+FP(8),B17       ; |22|
           LDW     .D1T2   *+FP(12),B28      ; |22|
           LDW     .D1T2   *+FP(4),B16       ; |22|
           LDW     .D1T1   *+FP(24),A3       ; |22|
           LDW     .D1T2   *+FP(20),B18      ; |22|
           MV      .L2     B12,B9            ; |22|
||         LDW     .D1T1   *+FP(16),A10      ; |22|
||         MV      .S2X    A12,B8            ; |22|
||         MV      .L1X    B6,A9             ; |22|
||         MV      .D2     B8,B7             ; |22|

           MVC     .S2     CSR,B27
           AND     .L2     -2,B27,B5
           MVC     .S2     B5,CSR            ; interrupts off
||         MV      .L1     A6,A8
||         MV      .S1     A8,A16
```

Figure 4.19: Assembly code of function *channel_estimation_FIXED*.

```
;******************************************************************************
;* FUNCTION NAME: _interpolation_FIXED                                        *
;*                                                                            *
;*   Regs Modified     : A0,A3,A4,A5,A6,A7,A8,A9,B4,B5,B7,B8,B9,A16,A17,A18,  *
;*                           A19,A20,A21,A22,A23,A24,A25,A26,A27,A28,A29,A30, *
;*                           B16,B17,B18,B19,B20,B21,B22,B23,B24              *
;*   Regs Used         : A0,A3,A4,A5,A6,A7,A8,A9,A10,A12,B3,B4,B5,B6,B7,B8,B9,*
;*                           B10,DP,SP,A16,A17,A18,A19,A20,A21,A22,A23,A24,   *
;*                           A25,A26,A27,A28,A29,A30,B16,B17,B18,B19,B20,B21, *
;*                           B22,B23,B24                                      *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                    *
;******************************************************************************
;* Using -g (debug) with optimization (-o3) may disable key optimizations!    *
;******************************************************************************
_interpolation_FIXED:
;** -------------------------------------------------------------------------*
        .dwcfa  0x0e, 0
        .dwcfa  0x09, 126, 19

           MV      .L2X    A8,B5             ; |44|
           MV      .L1X    B4,A19            ; |44|
||         SUB     .L2X    A4,4,B21
           MV      .L1X    B10,A9            ; |44|
||         MV      .D1     A12,A8            ; |44|
||         MV      .L2X    A10,B4            ; |44|
||         MVK     .S1     0x1555,A22

           MV      .L1     A6,A21
           LDH     .D1T2   *+A21(2),B16      ; |54| (P) <0,2>
           LDH     .D1T2   *A21,B7           ; |54| (P) <0,3>
           LDH     .D1T1   *+A21(2),A7       ; |96| (P) <0,0>
||         MV      .L2X    A19,B8
||         LDH     .D2T2   *++B21(4),B20     ; |78| (P) <0,6>
||         MV      .L1X    B8,A20
           LDH     .D2T1   *B8,A29           ; |142| (P) <0,2>
||         MVC     .S2     CSR,B24
           LDH     .D2T2   *B8,B9            ; |124| (P) <0,5>
||         LDH     .D1T1   *A21,A3           ; |96| (P) <0,4>
||         AND     .L2     -2,B24,B17
           LDH     .D2T1   *B8,A6            ; |136| (P) <0,3>
||         MVC     .S2     B17,CSR           ; interrupts off
```

Figure 4.20: Assembly code of function *interpolation_FIXED*.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 180
;*      Loop opening brace source line   : 181
;*      Loop closing brace source line   : 192
;*      Known Minimum Trip Count         : 120
;*      Known Maximum Trip Count         : 120
;*      Known Max Trip Count Factor      : 120
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 7
;*      Partitioned Resource Bound(*)    : 7
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                    0        0
;*      .S units                    1        1
;*      .D units                    7*       7*
;*      .M units                    3        1
;*      .X cross paths              2        2
;*      .T address paths            6        7*
;*      Long read paths             0        0
;*      Long write paths            0        0
;*      Logical  ops (.LS)          0        0     (.L or .S unit)
;*      Addition ops (.LSD)         0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)            1        1
;*      Bound(.L .S .D .LS .LSD)    3        3
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 7  Schedule found with 4 iterations in parallel
;*      Done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 2
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      For further improvement on this loop, try option -mh6
;*
;*      Minimum safe trip count     : 3
;*----------------------------------------------------------------------------*
```

Figure 4.21: Software pipelining information of function *channel_estimation_FIXED*.

Table 4.4: OFDMA Uplink Efficiency Performance Comparison

Condition	Execution Cycles	Minimum Needed Cycles	Efficiency
1024-FFT, BW: 10 MHz 10 subchannels	4576	900	19.67%
2048-FFT, BW: 20 MHz 10 subchannels	4800	900	18.75%
2048-FFT, BW: 20 MHz 20 subchannels	9224	1800	19.51%

Our DSP can execute 2 multiplications and 6 additions in one cycle. We have 5 multiplications and 1 additions per sample in function *pilot_extraction_FIXED*, and 12 multiplications, 40 additions, 12 shift computation per tile in function *interpolation_FIXED*. Assuming using 10 subchannels, we need a minimum $\max\{5/2, 1/6\} \times 120 = 360$ cycles in *pilot_extraction_FIXED* and $\max\{12/2, (40+12)/6\} \times 60 = 540$ for *interpolation_FIXED*. The total needed minimum cycles are $360+540=900$. If we use 20 subchannels to transmit, the needed minimum cycles would be twice and equal to 1800.

We compare the actual execution cycles taken by the compiled code with the minimum cycles needed and calculate the efficiency, where the efficiency is defined as:

$$\text{Efficiency} = \frac{\text{Minimum Cycles Needed}}{\text{Practical Execution Cycles}}[18]. \tag{4.14}$$

Table 4.4 illustrates the efficiency comparisons in three different kinds of transmission conditions.

Chapter 5

Downlink Channel Estimation and DSP Implementation

In this chapter, we introduce three methods to do the channel estimation in downlink transmission. The simple techniques and channel models we use are the same as described in chapter 4. The channel estimation techniques include LS in pilot positions, linear interpolation in frequency domain and several improvement methods in time domain. We evaluate the performance of each channel estimation approach mainly via symbol error rate (SER) and mean square error (MSE). The final of our work is the DSP implementation.

5.1 System Parameters and Channel Model

Table 5.1 gives the primitive and derived parameters used in our simulation work. In our system, we let the preamble be followed by 24 data symbols. In addition to AWGN, we use SUI-2 and SUI-3 to do simulation. Their profiles are already introduced in Table 4.2.

5.2 Channel Estimation Methods

The first symbol of the downlink transmission is preamble, and 24 data symbols are followed in each subframe. Pilots in the preamble appear every 3 subcarriers. Therefore, we first use

Table 5.1: OFDMA Downlink Parameters

Parameters	Values
Bandwidth	20 MHz
Central frequency	3.5 GHz
N_{used}	1681
Sampling factor n	28/25
G	1/8
N_{FFT}	2048
Sampling frequency	22.4 MHz
Subcarrier spacing	10.94 kHz
Useful symbol time	91.43 μs
CP time	11.43 μs
OFDMA symbol time	102.86 μs
Sampling time	44.65 ns

LS technique to estimate the channel response at each pilot location. Next, we do linear interpolation in frequency domain to get channel estimates at non-pilot subcarriers. We take this as the initial channel estimate.

In chapter 2, we mentioned downlink transmission uses cluster structure to transmit pilot and data information. Fig 5.1 shows the cluster transmission. The pilot positions are different in even and odd symbols. It plays an important role to the methods we propose.

5.2.1 Two-Point Cluster Linear Interpolation

Since there are two pilots in each cluster, we directly use these two pilots as reference to do linear interpolation in the frequency domain within the cluster. We also use exponential averaging to enhance the performance in time domain. Here are the detailed steps of this method:

- Estimate the channel response at each pilot location by using the LS technique.
- Use the linear interpolation scheme to get the data channel response from the two

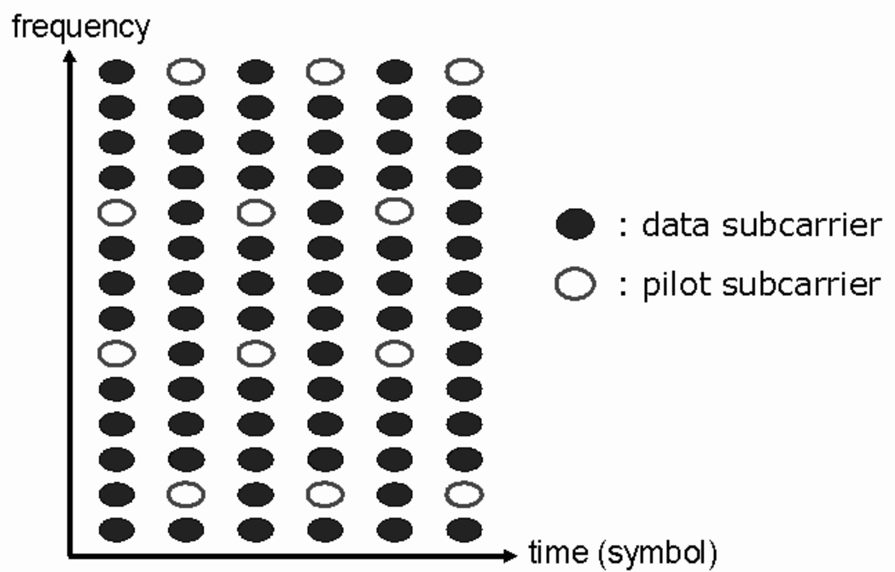
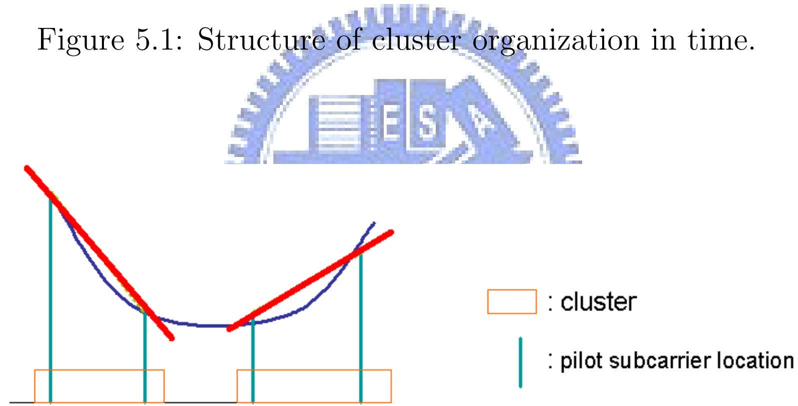
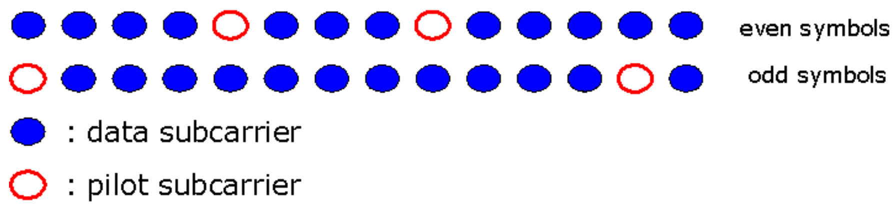


Figure 5.1: Structure of cluster organization in time.



(a)



(b)

Figure 5.2: (a) 2-point cluster linear interpolation illustration, bold line is our estimation of linear interpolation (b) pilot positions are different in even and odd symbols

estimated pilot values (see Fig. 5.2(a)).

- Preamble information is utilized by exponential averaging weighting $w = 0.9$.
- Exponential averaging of data symbols is used in time domain.

5.2.2 Four-Point Cluster Linear Interpolation

As mentioned, time-domain averaging over several OFDMA symbols can enhance the channel estimation performance, if the channel does not vary significantly over this time period. The channel response stays relatively constant over a few OFDMA symbols, or it can be approximately modeled as slowly linearly varying over a larger number of OFDMA symbols.

Since the pilot positions are different in even and odd symbols, we take the pilots in previous symbol as reference. There would be four pilots in a cluster to estimate other data channel response instead of its original two pilots. Next, we do linear interpolation in the frequency domain within the cluster.

Here are the detailed steps in this method:

- Estimate the channel response at each pilot location by using the LS technique.
- Take the pilots in previous symbol as reference (see Fig. 5.3(a)). It becomes four pilots in a cluster (see Fig. 5.3(b)).
- Use the linear interpolation scheme to get the data channel response from the four estimated pilot values (see Fig. 5.3(c)).
- Preamble information is utilized by exponential averaging weighting $w = 0.9$.

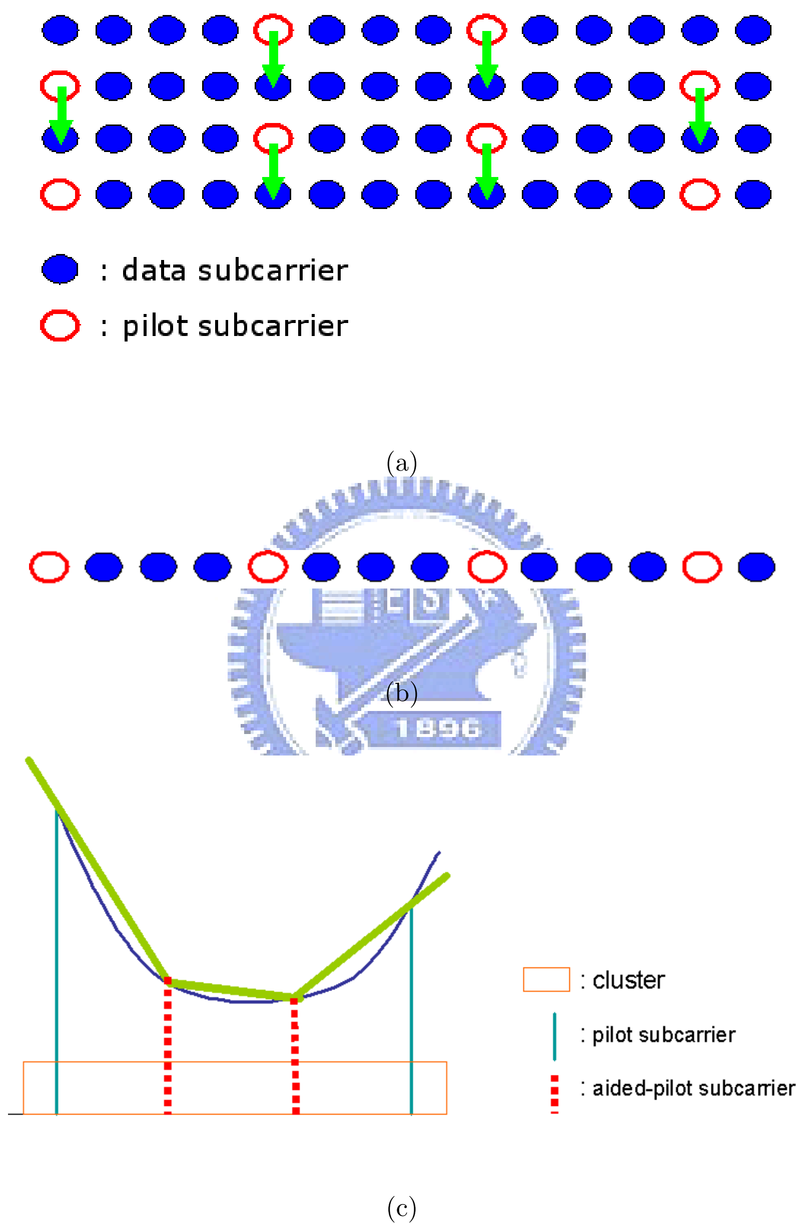


Figure 5.3: (a) Pilots in previous symbol taken as reference. (b) Four pilot points in cluster. (c) Four-point cluster linear interpolation illustration. Bold line is our estimation by linear interpolation.

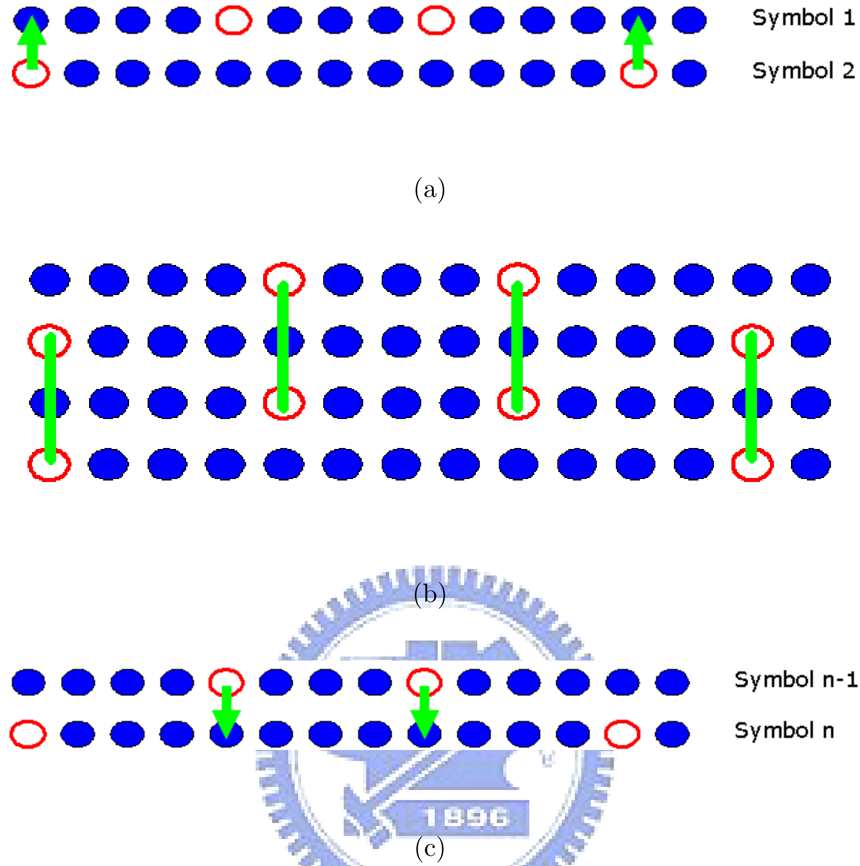


Figure 5.4: Advanced four-point cluster linear interpolation. (a) First data symbol. (b) Second to $(n-1)$th data symbols. (c) Last (nth) data symbol.

5.2.3 Advanced Four-Point Cluster Linear Interpolation

The channel may be modeled as linearly varying in a short time period can be used to yield a predicted channel response at future OFDMA symbol instants, for example,

$$H_k(t+1) = H_k(t) + [H_k(t) - H_k(t-1)].$$

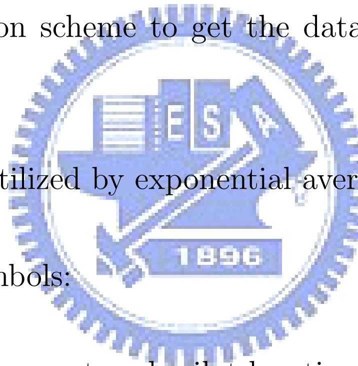
If the receiver latency is not a concern, time-domain interpolation can be performed. A simplest way of time-domain interpolation is, of course, linear interpolation, such as:

$$H_k(t) = \frac{1}{2}[H_k(t-1) + H_k(t+1)]$$ [19].

We take the pilots in previous and next symbols as reference. Therefore, there would also be four pilots in a cluster to estimate other data channel response instead of its original two pilots (see Fig. 5.3(b)). We do linear interpolation in the frequency domain within the cluster afterward (see Fig. 5.3(c)). The detailed steps each symbol are as follows:

1) First data symbol:

- Estimate the channel response at each pilot location by using the LS technique.
- Take the pilots only in next symbol as reference(see Fig. 5.4(a)). It becomes four pilots in a cluster.
- Use the linear interpolation scheme to get the data channel response from the four estimated pilot values.
- Preamble information is utilized by exponential averaging weighting $w = 0.9$.



2) Second to $(n-1)$th data symbols:

- Estimate the channel response at each pilot location by using the LS technique.
- Take the pilots in previous and next symbol as reference(see Fig 5.4(b)). It becomes 4 pilots in a cluster.
- Use the linear interpolation scheme to get the data channel response from the 4 estimated pilot values.

3) Last data symbol:

- Estimate the channel response at each pilot location by using the LS technique.
- Take the pilots only in previous symbol as reference(see Fig 5.4(c)). It becomes 4 pilots in a cluster.

- Use the linear interpolation scheme to get the data channel response from the 4 estimated pilot values.

Last to mention, this method gives our system a symbol time latency. For example, if we want to get the channel response of the first symbol, we must wait until we receive the second data symbol. In the last symbol, we get not only the previous symbol information but also the last one's.

5.3 Simulation Results

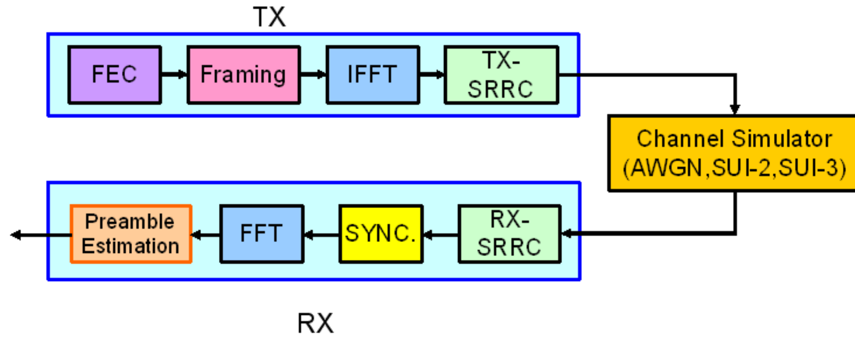
5.3.1 Simulation Flow

Figure 5.5 illustrates the block diagrams of our simulated system. We also assume perfect synchronization and omit it in our simultion. Because of all pilots in the preamble, there is no need to do DeFraming and DeFEC. After channel estimation, as we do in uplink transmission, we calculate the channel MSE between the real channel and the estimated one, where the average is taken over the subcarriers. The symbol error rate (SER) can also be obtained after demapping.

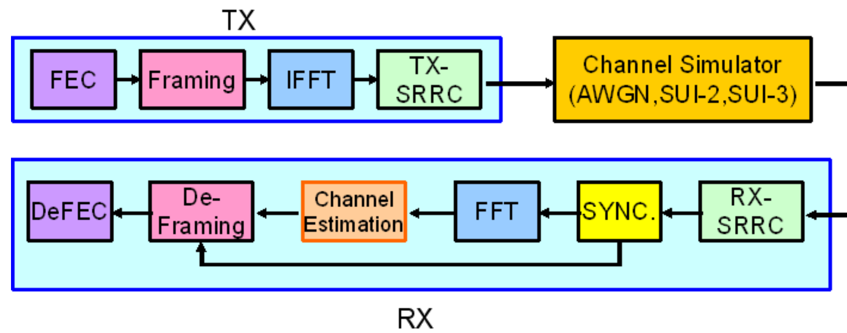
5.3.2 Validation with AWGN Channel

Before considering multipath channels, we do simulation with an AWGN channel to validate the simulation model. We validate this model by comparing theoretical SER curves, and the SER curves resulting from simulation.

In Figure 5.6, the theoretical symbol error rate (SER) curve versus E_s/N_0 for uncoded QPSK is plotted together with the SER curve resulting from the simulation. In this figure, we simulate for no channel estimation error. This validates the simulation (we use C/C++ programming language and TI's code composer studio).



(a)



(b)

Figure 5.5: Downlink transmission simulation flow. (a) Preamble. (b) Data symbols.

5.3.3 Floating-Point Simulation

Figure 5.7 shows the performance of 2-point cluster linear interpolation with different exponential weighting in AWGN and SUI-2 with velocity $v = 60km/hr$. The method of with weighting $w = 0.9$ in AWGN has the best SER and MSE. But in the condition of SUI-2, velocity being 60 km/hr, this becomes the worst situation in both SER and MSE. It is because the variance of channel condition is much violent in multipath such as SUI-2 than in AWGN. We also get the validation from the analysis of in given velocity, calculating the

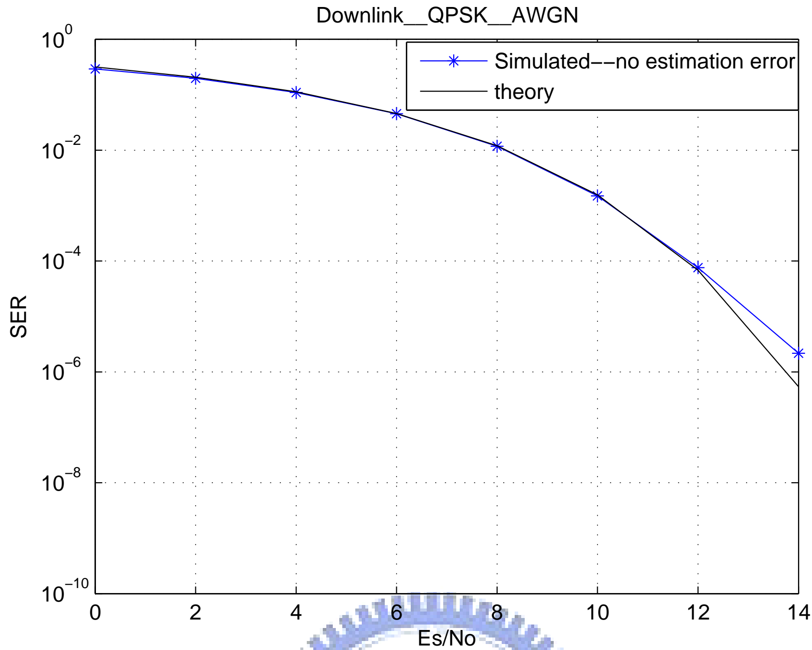


Figure 5.6: The SER curve for uncoded QPSK resulting from simulation matches the theoretical one.

MSE by using the variance of Bessel function. Therefore, the performance of using exponential weighting in multipath channel is very poor. Figure 5.8 shows 3 methods in SUI-2 with different velocity of QPSK, including: two-point with exponential weighting 0.9, two-point and four-point cluster cluster linear interpolation.

Figures 5.9 and 5.10 illustrate the comparison between all methods we use, including two-point, two-point with exponential weighting w=0.9, four-point and advanced four-point cluster linear interpolation in different channel condition. It is shown that the advanced four-point has the best performance in multipath channel. Therefore, we use this method to simulate other conditions in the following.

Figure 5.11 illustrates advanced four-point cluster linear interpolation with different modulation (uncoded QPSK, 16QAM and 64QAM) in AWGN. We also compare our simulation

results with theoretical and no estimation error curves in SER. Figure 5.12 shows comparisons with another theory curve which takes data MSE into consideration in AWGN. The 3 lines in MSE of different modulation match with each other as a straight line with slope $m = -1$ in Fig. 5.11(a). The results of MSE are unrelated to the modulation type because the pilots are boosted-BPSK modulated in each modulation case. And the channel response is interpolated only using the pilot information.[17]

The simulation with different velocity and different modulation in SUI-2 is given in Fig. 5.13, and Fig. 5.14 gives only QPSK in SUI-3.

Figure 5.15 shows the comparison of taking preamble information into consideration. When higher the velocity in SUI-2, the performance of no preamble effect in high E_s/N_0 is better.

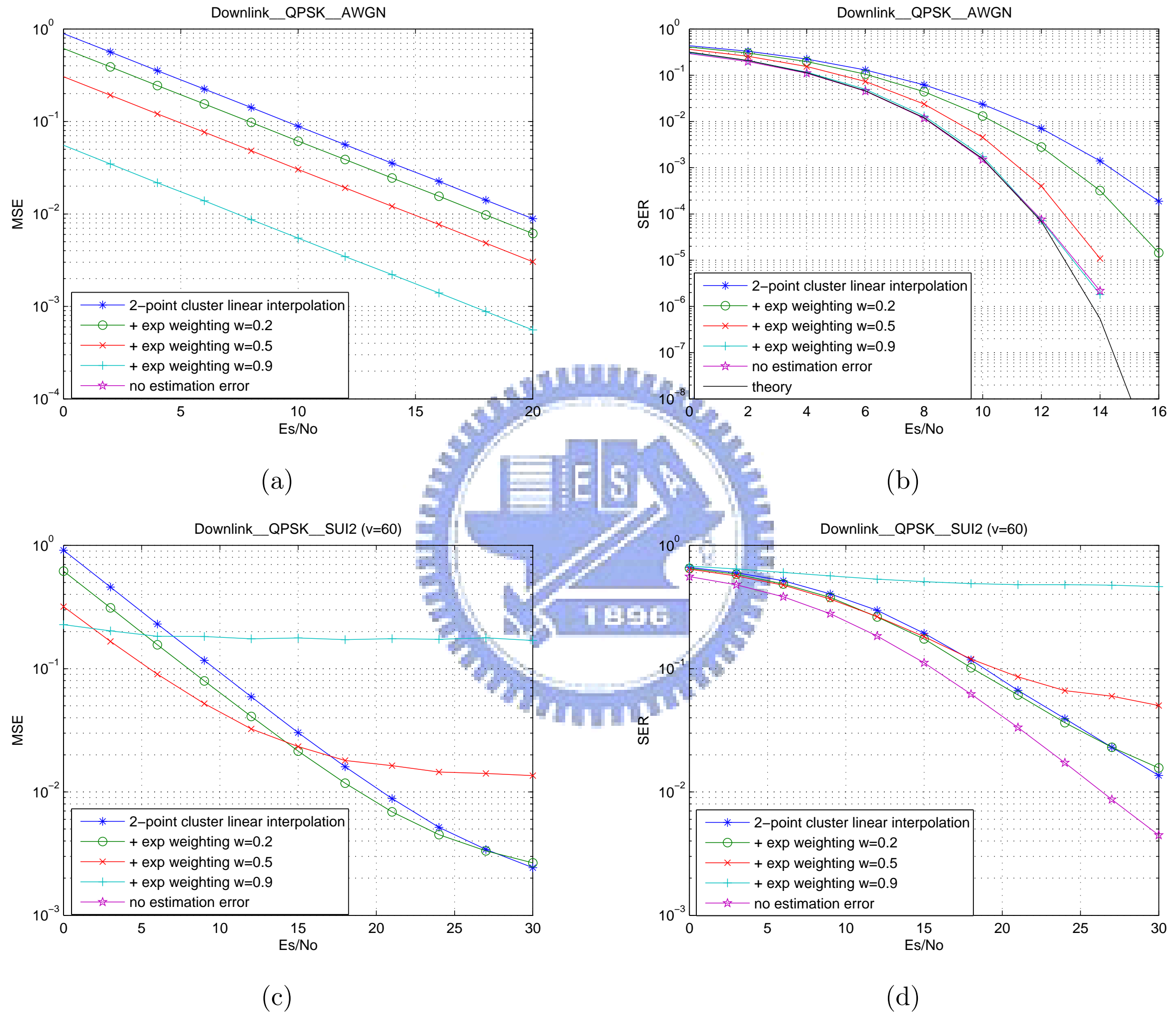
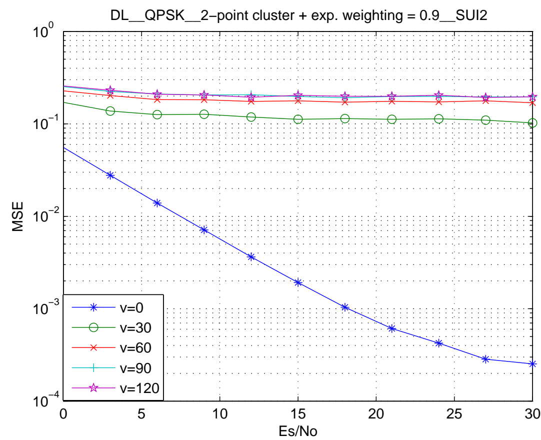
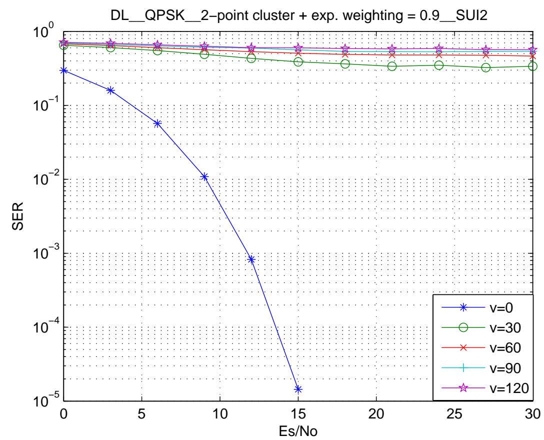
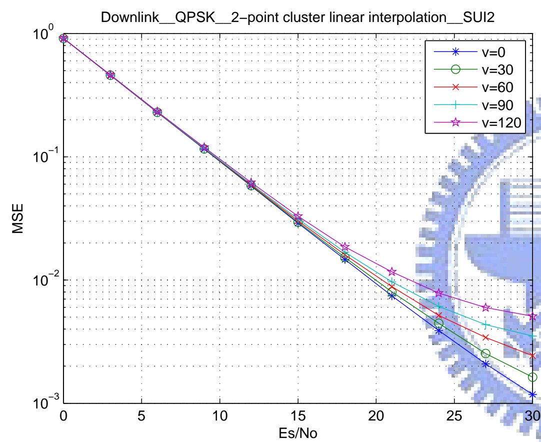


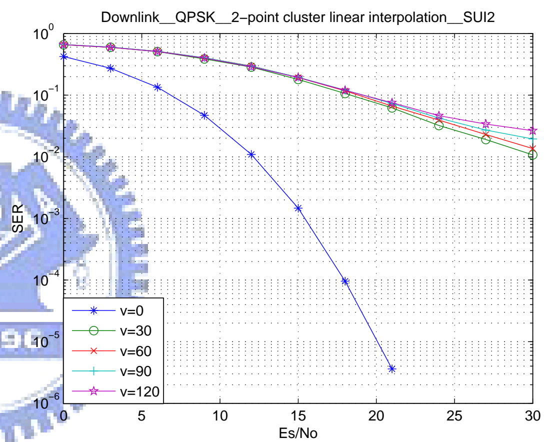
Figure 5.7: Two-point cluster linear interpolation with different exponential weighting with QPSK. (a),(b) In AWGN. (c),(d) In SUI-2 with velocity v=60 km/hr.



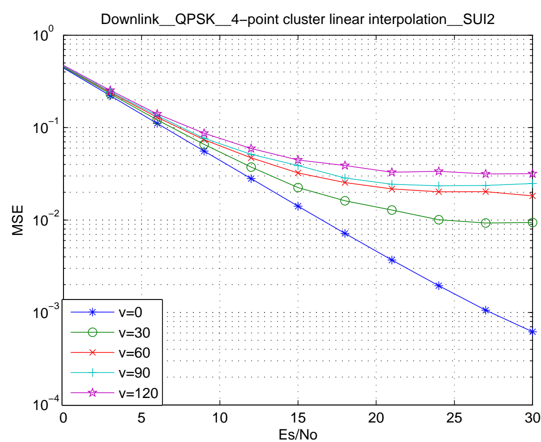


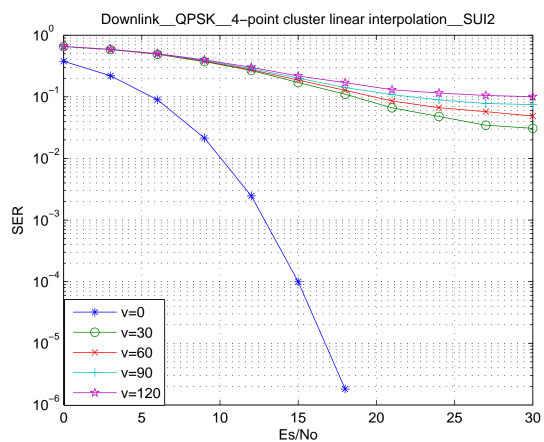
(a) (b)





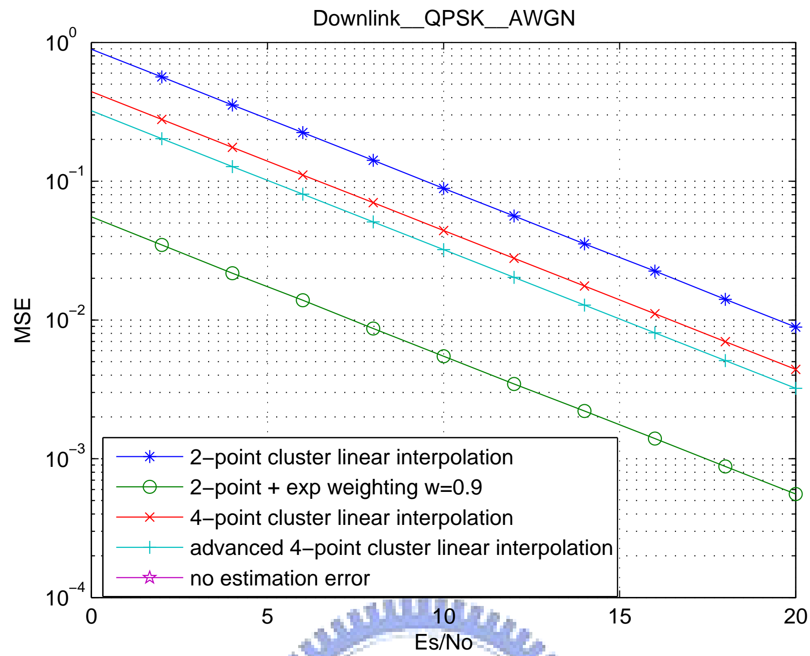
(c) (d)



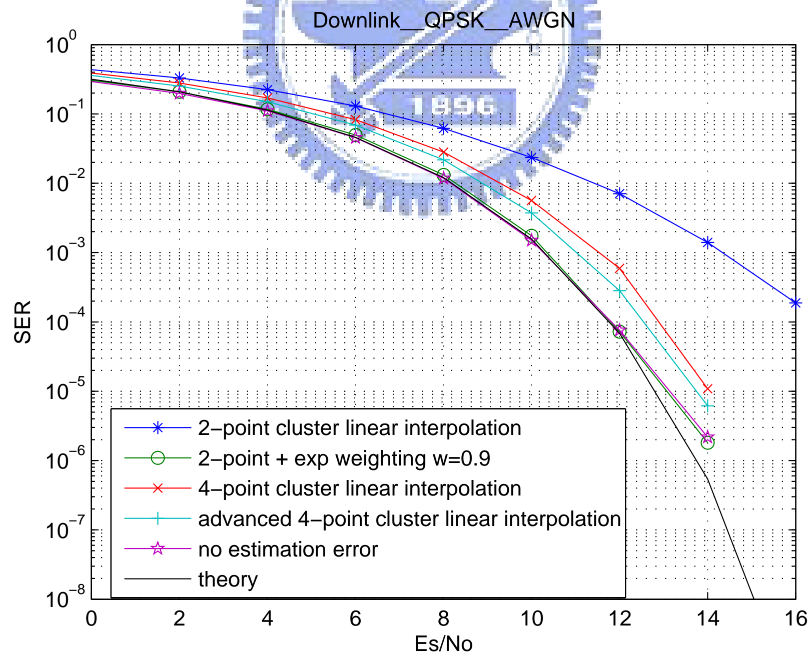


(e) (f)

Figure 5.8: Three methods of cluster cluster linear interpolation with different velocity in SUI-2 of QPSK. (a),(b) Two-point with exponential weighting w=0.9. (c),(d) Two-point. (e),(f) Four-point.

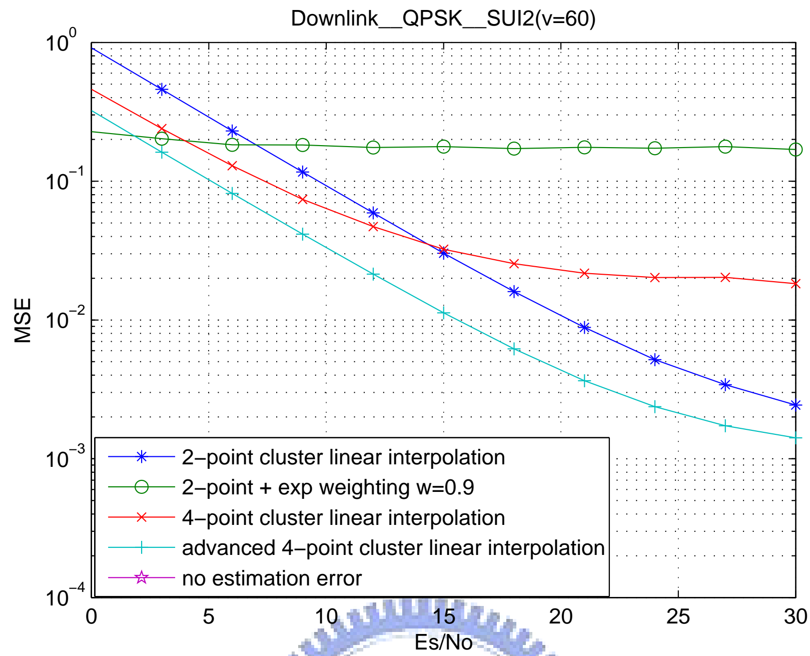


(a)

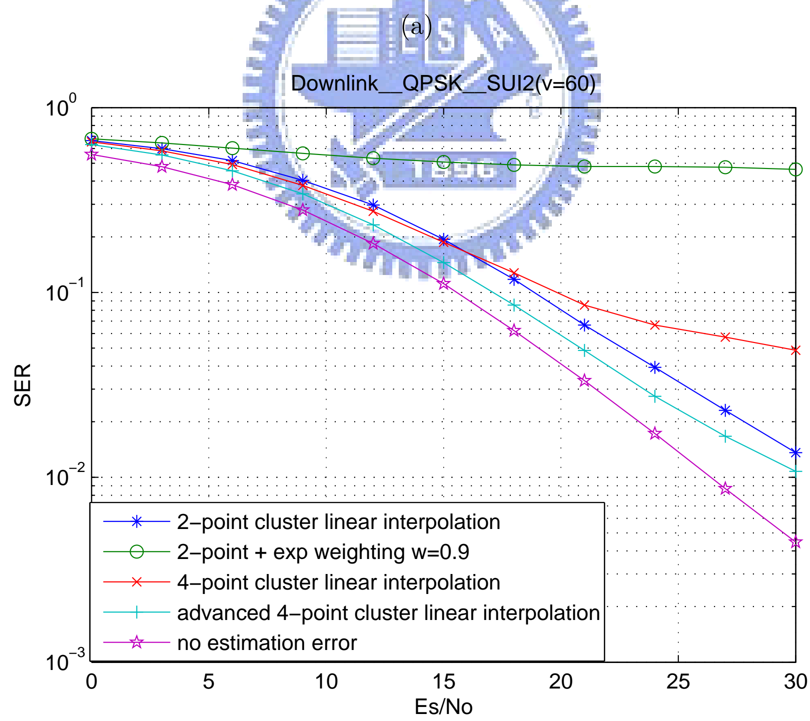


(b)

Figure 5.9: Comparison of all methods we use, including two-point, two-point with exponential weighting w=0.9, four-point and advanced four-point cluster linear interpolation in AWGN. (a) MSE. (b) SER.

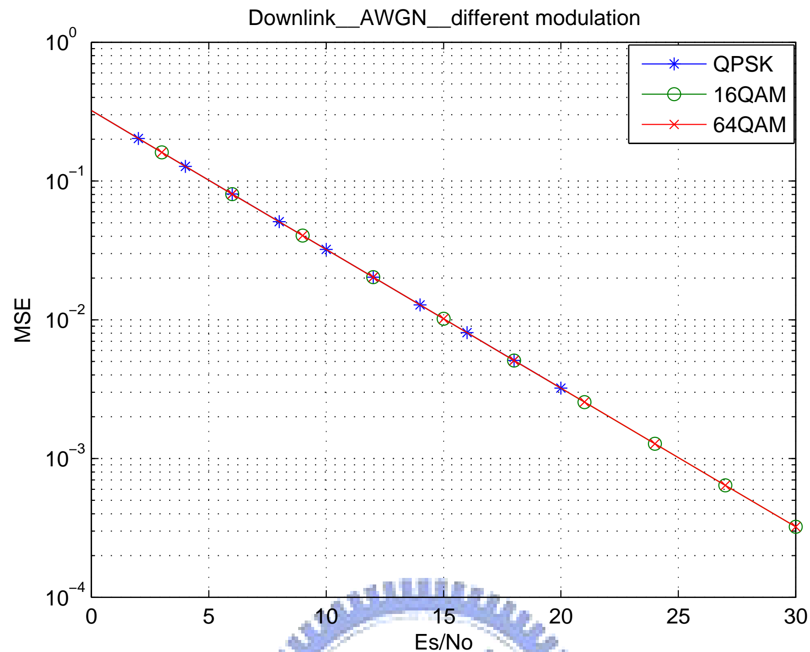


(a)

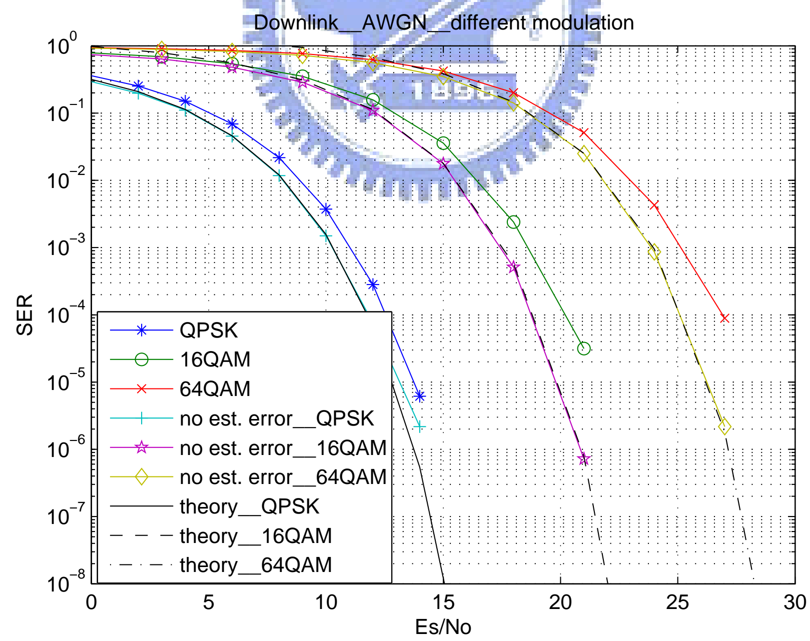


(b)

Figure 5.10: Comparison of all methods we use, including two-point , two-point with exponential weighting w=0.9, four-point and advanced four-point cluster linear interpolation in SUI-2 with velocity v=60 km/hr. (a) MSE. (b) SER.

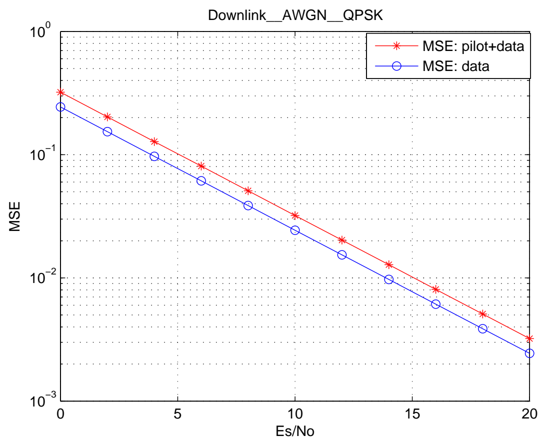


(a)

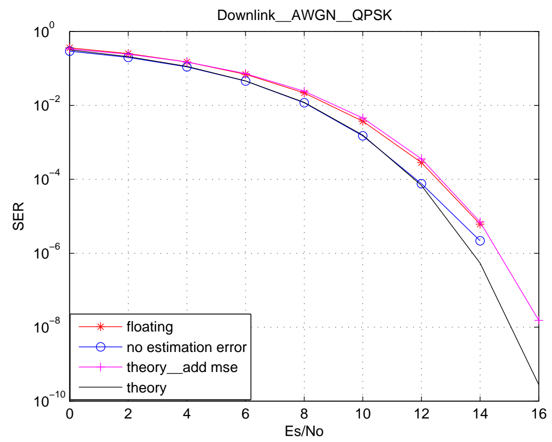


(b)

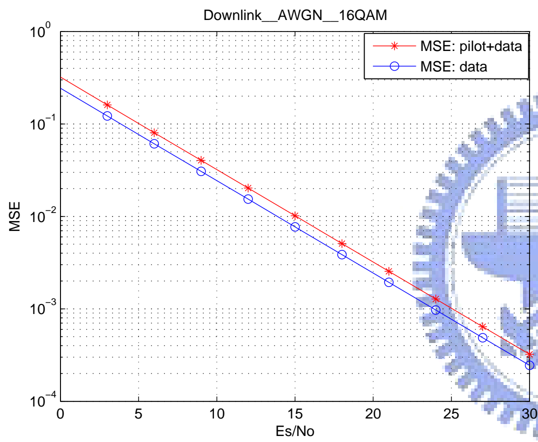
Figure 5.11: Advanced four-point linear interpolation with different modulation in AWGN. (a) MSE. (b) SER.



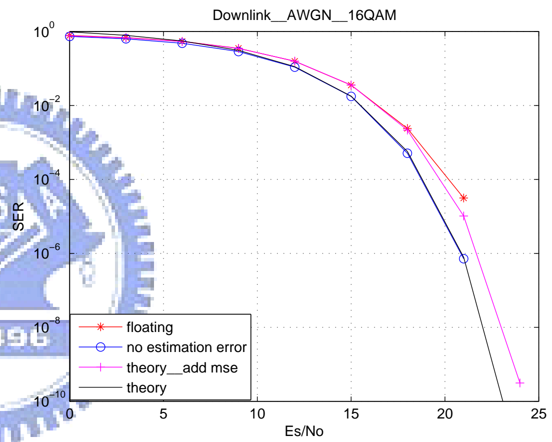
(a)



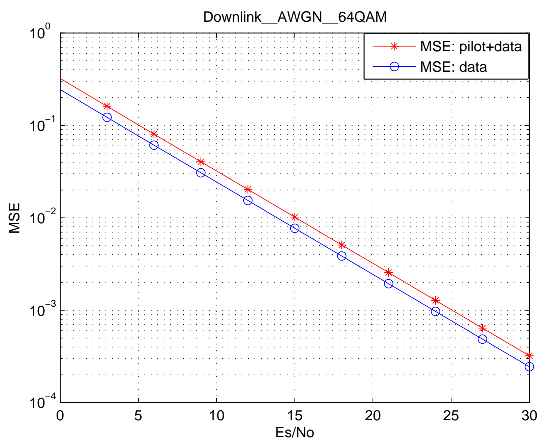
(b)



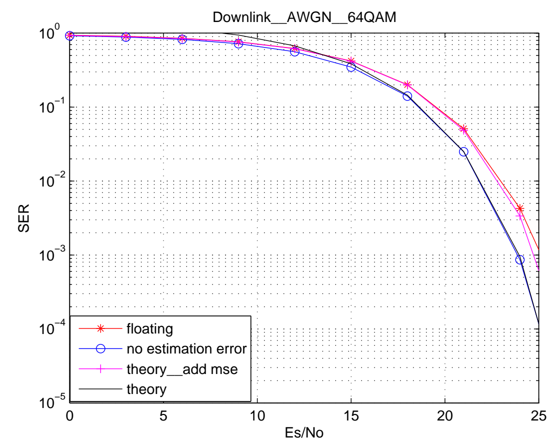
(c)



(d)

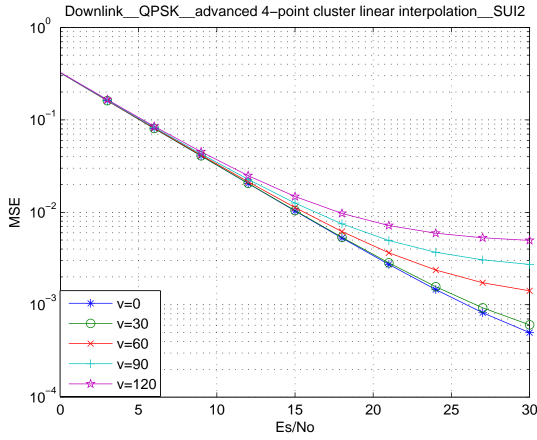


(e)

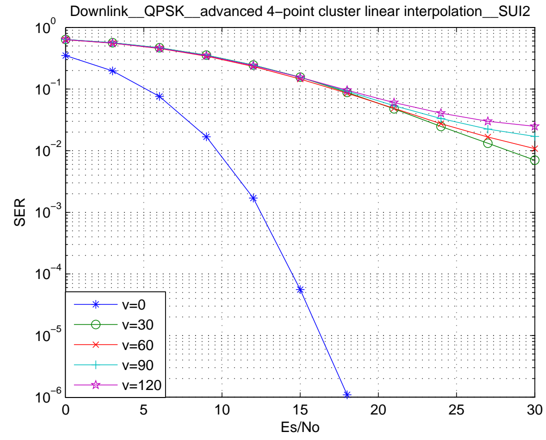


(f)

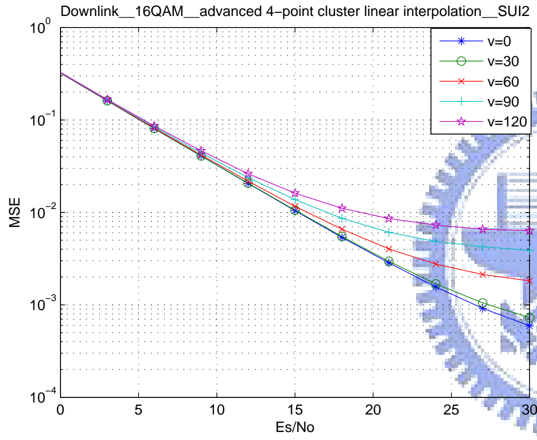
Figure 5.12: Advanced four-point cluster linear interpolation compared with theory adding data MSE in AWGN. (a),(b) QPSK. (c),(d) 16QAM. (e),(f) 64QAM.



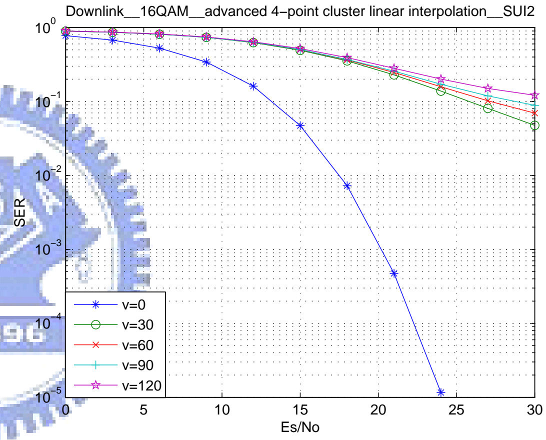
(a)



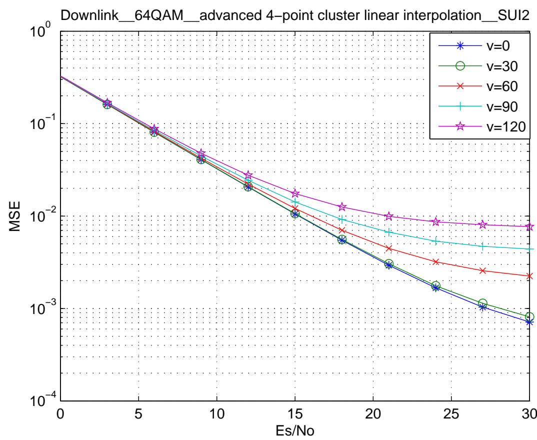
(b)



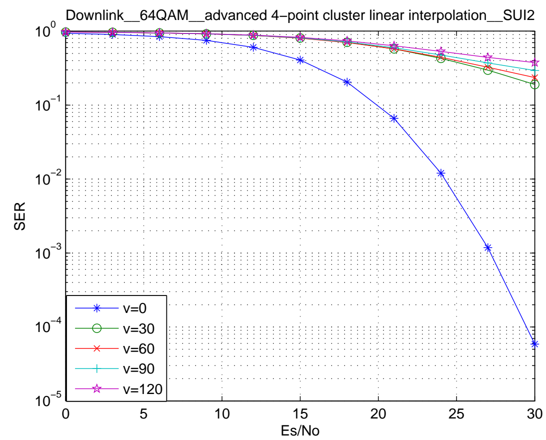
(c)



(d)

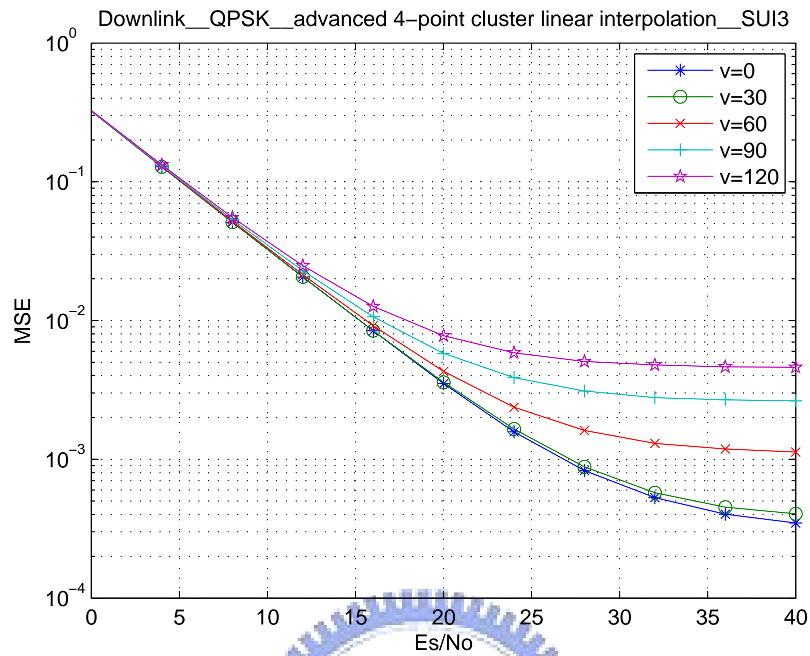


(e)

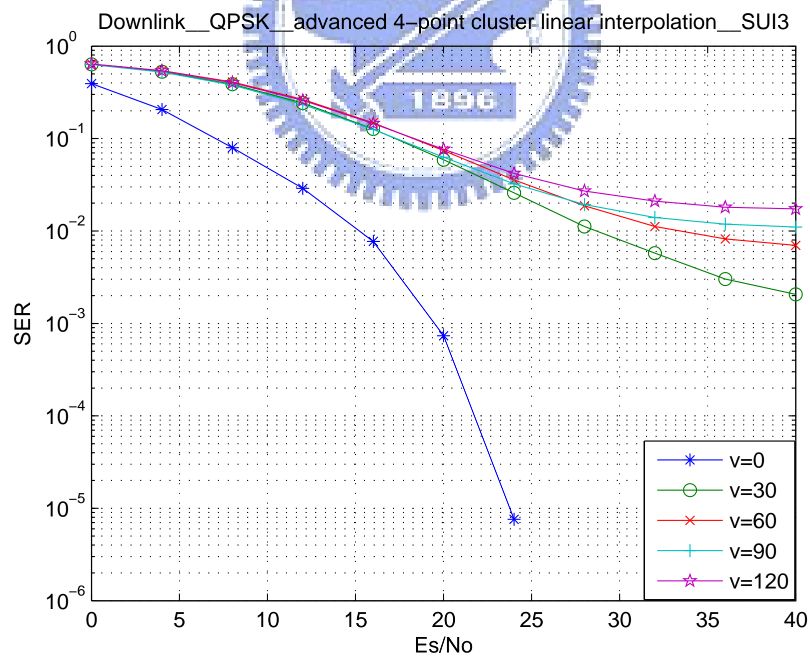


(f)

Figure 5.13: Advanced four-point cluster linear interpolation with different velocities and different modulations in SUI-2. (a),(b) QPSK. (c),(d) 16QAM. (e),(f) 64QAM.



(a)



(b)

Figure 5.14: Advanced four-point cluster linear interpolation with different velocities in SUI-3 with QPSK. (a) MSE. (b) SER.

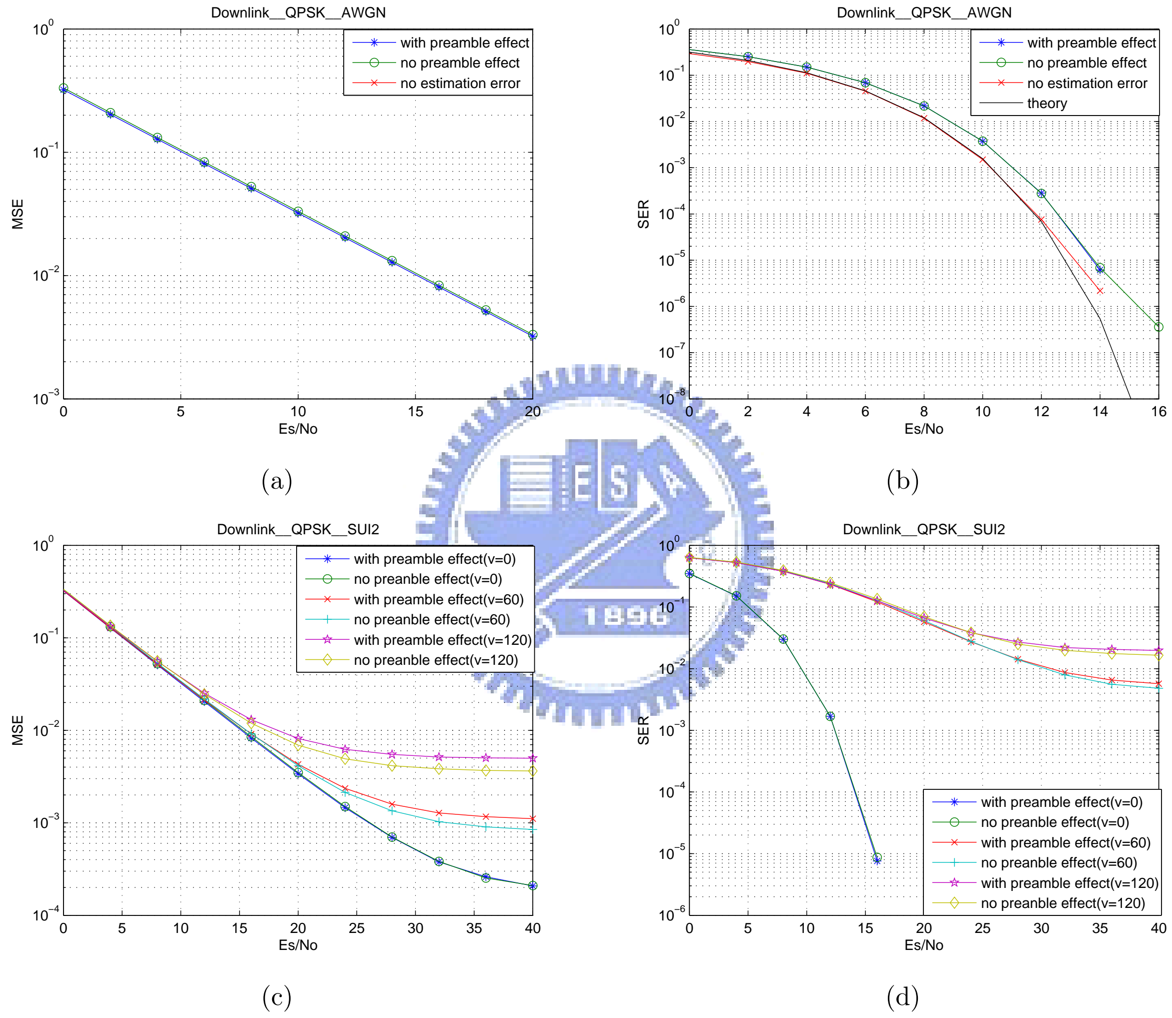
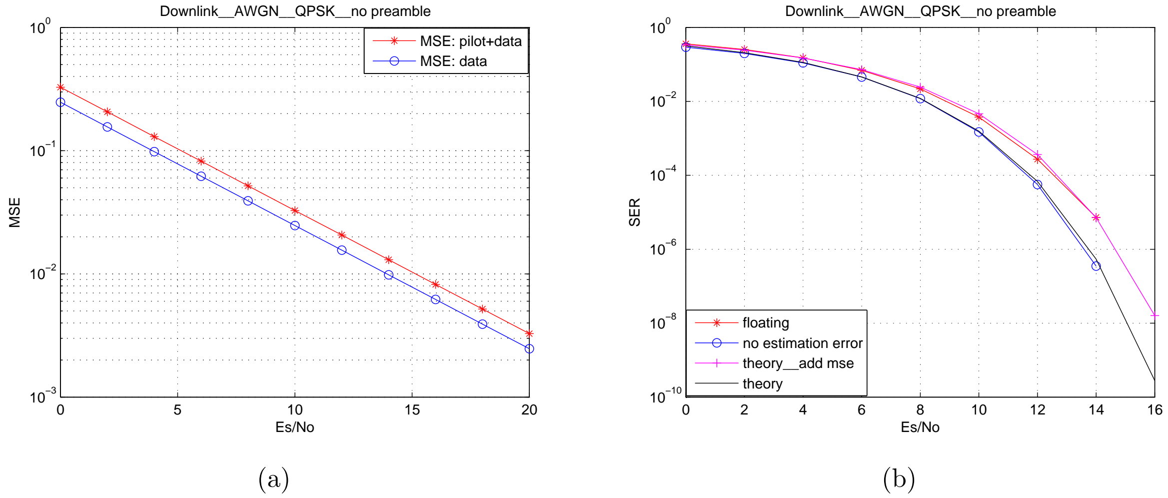


Figure 5.15: Advanced four-point cluster linear interpolation considering preamble effect. (a),(b) AWGN. (c),(d) SUI-2.



(a) (b)

Figure 5.16: Advanced four-point cluster linear interpolation with no preamble in AWGN. (a) MSE. (b) SER.

5.3.4 Cluster Analysis

In this section, we give detail performance analysis of advanced 4-point cluster linear interpolation. Here we do not transmit any preambles but all data symbols for simulation.

Figure 5.16 shows the SER and MSE in AWGN without transmitting any preambles. Here we especially illustrate the MSE curve resulting from only data subcarriers. In Fig. 5.16(b), we give another theoretical curve considering the data MSE in our simulation. We can find out that our simulation curve is near to it.

We can find the SER and MSE spread over all subcarriers in Fig. 5.17. If we collect MSE of all used clusters and average, we get the curve showed in Fig. 5.18(a). We calculate the theory MSE value of non-pilot positions. It appears that our results are match with them. We put these MSE values into consideration when showing the theoretical SER curve. As shown in Fig. 5.18(b), our resulted average cluster SER curve is lower than the theory. It is because the MSE values contains not only pilot but data subcarriers.

(a)

(b)

Figure 5.17: (a)MSE and (b)SER over used subcarriers in AWGN at 10 dB SNR.

Figure 5.19(a) shows the MSE values resulting only from data subcarriers. We calculate the theory MSE value of non-pilot positions too. In Fig. 5.19(b), we get the exact theory curve which is lower than our simulation. Figure 5.20 illustrates the SER and MSE performance in even and odd symbols. The pilot positions of both even and odd symbols have high MSE but SER is 0.

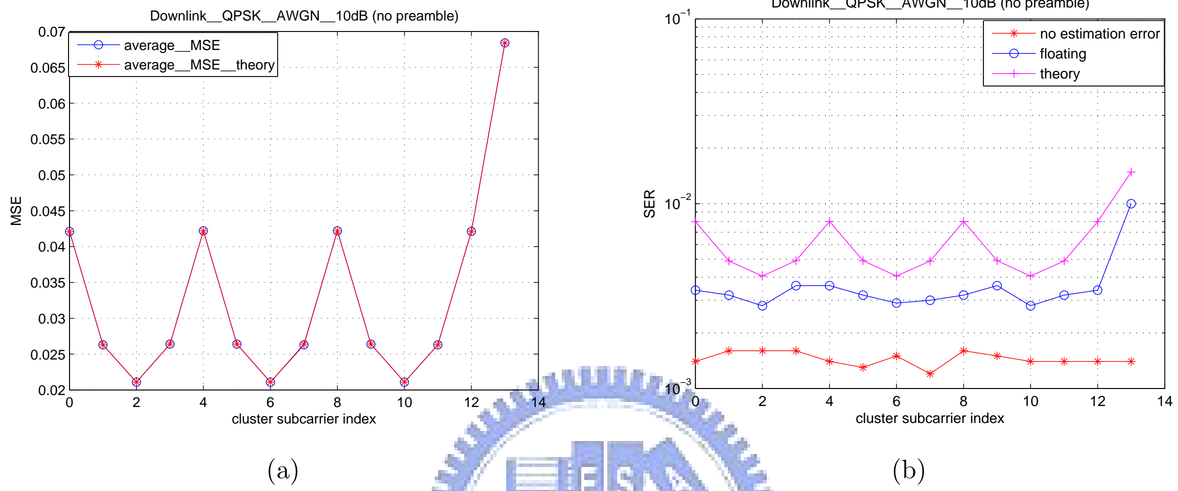


Figure 5.18: Average cluster performance in AWGN at 10 dB SNR. (a) MSE. (b) SER.

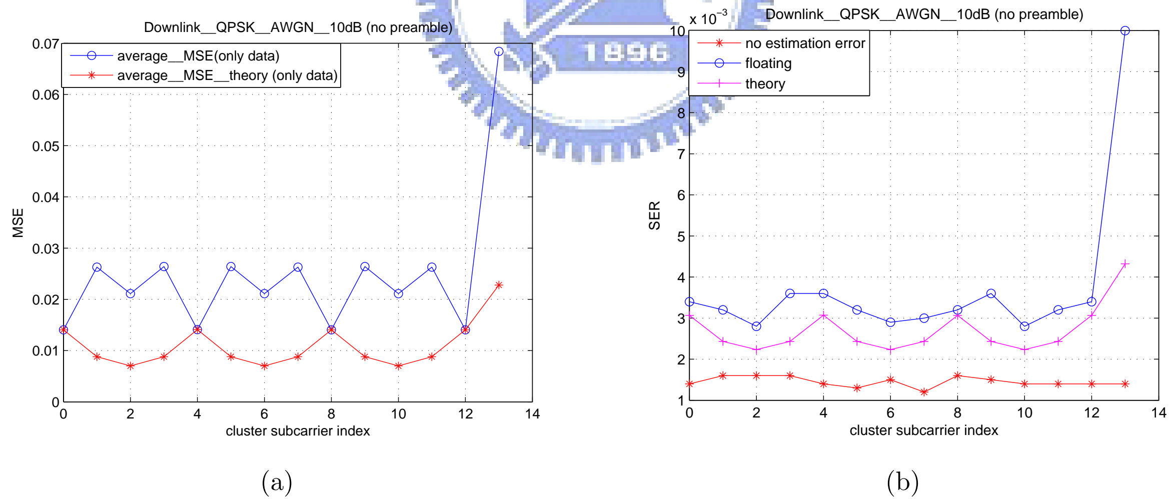
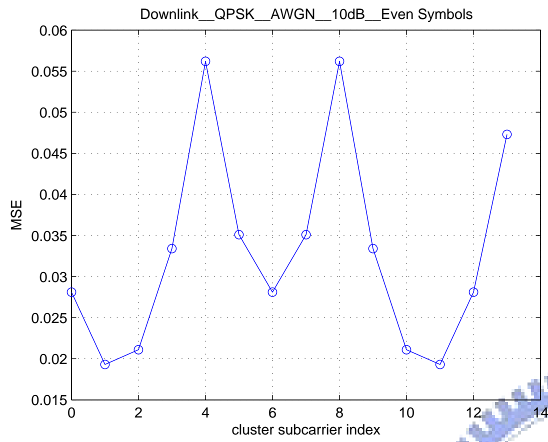
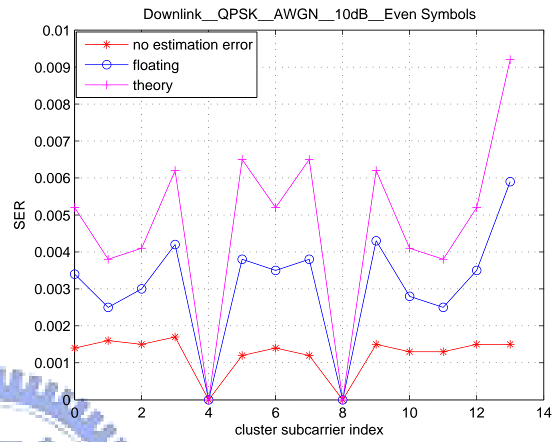


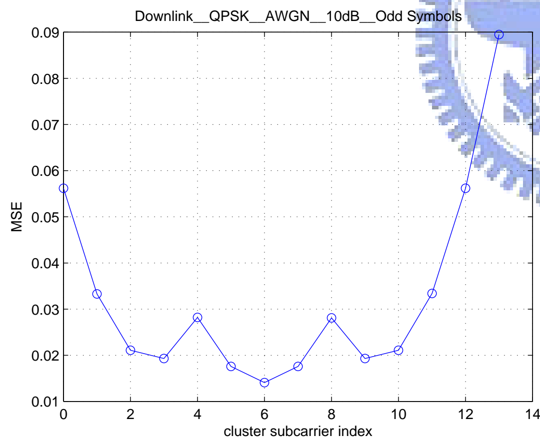
Figure 5.19: Average cluster performance in AWGN at 10 dB SNR. (a) MSE. (b) SER.



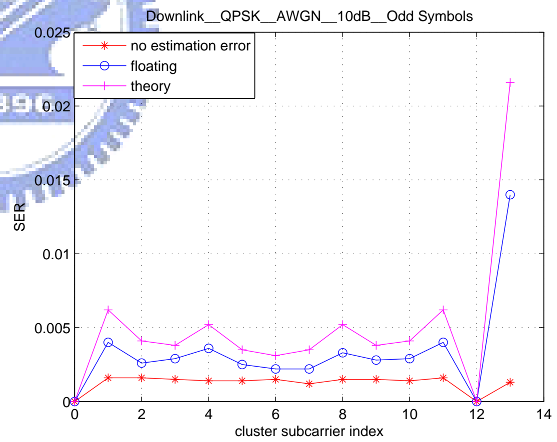
(a)



(b)



(c)



(d)

Figure 5.20: Average cluster performance in AWGN at 10 dB SNR. (a),(b) Even symbols. (c),(d) Odd Symbols.

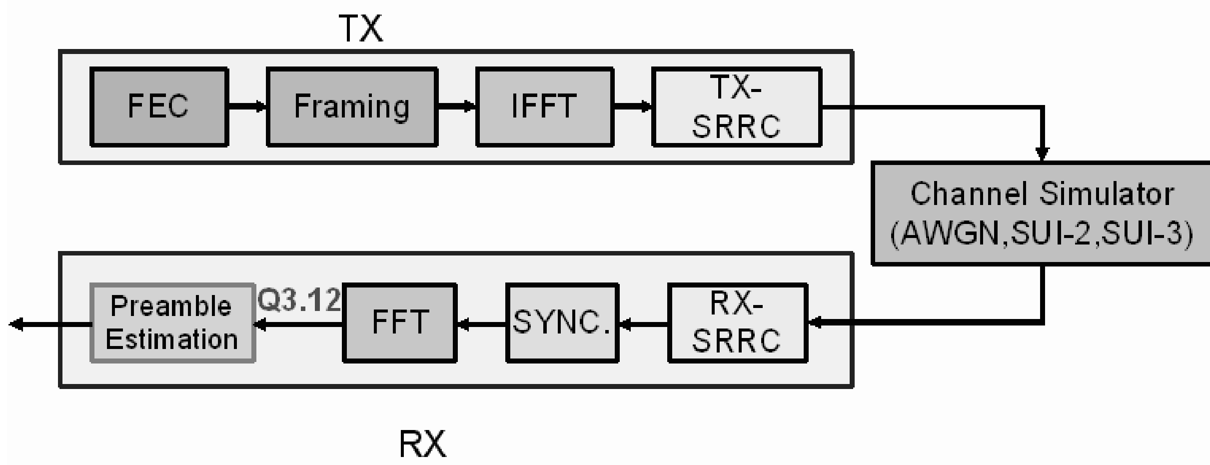


Figure 5.21: Fixed-point preamble transmission formats in our design.

5.4 DSP Implementation

5.4.1 Fixed-Point Data Formats

Here we only focus on the "channel estimation" function. Therefore, we translate the input to channel estimation into fixed for simplicity. The fixed-point preamble and data transmission formats used in our design based of advanced four-point cluster linear interpolation method are shown in Figures 5.21 and 5.22. We use Q3.12 instead of Q2.13 for preambles because of the range of pilot values in the preamble, which is $[-2\sqrt{2}, 2\sqrt{2}]$. The detail data formats of preamble estimation and channel estimation are also illustrated in Figures 5.23 and 5.24.

5.4.2 Fixed-Point Simulation

Figure 5.25 illustrates the performance of fixed-point computation compared with floating-point computation in AWGN. As we can see, there are almost no difference between these two kinds of fixed-point data formats in the function of channel estimation (preamble data formats are still Q3.12). We use Q2.13 as final because of more accuracy. Figure 5.26 compares fixed-point computation using Q2.13 and floating-point computation in SUI-2.

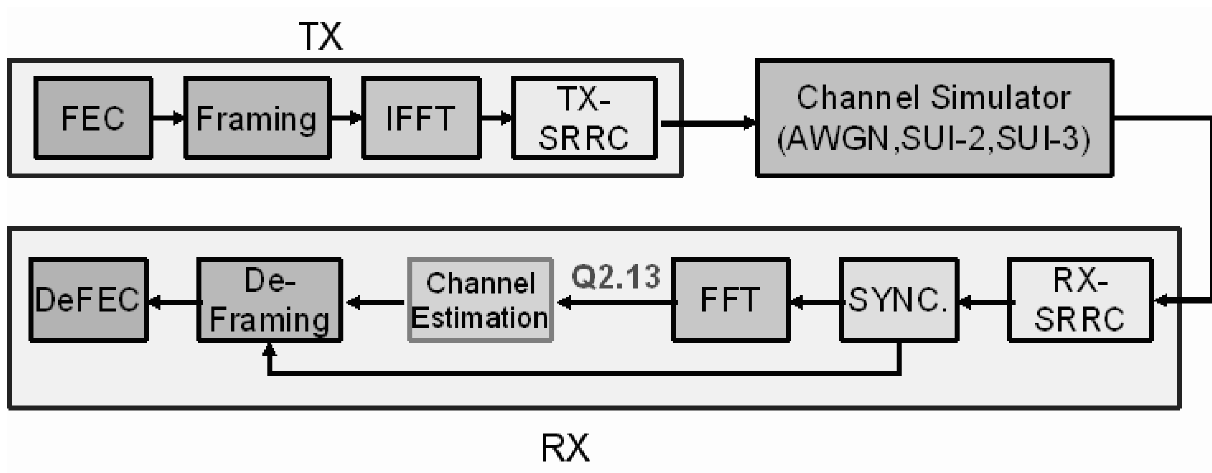


Figure 5.22: Fixed-point data transmission formats in our design.

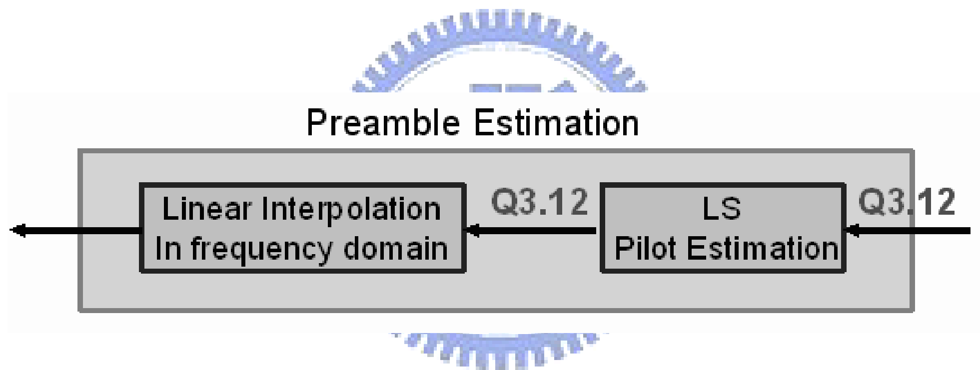


Figure 5.23: Fixed-point data formats in preamble estimation of our design.

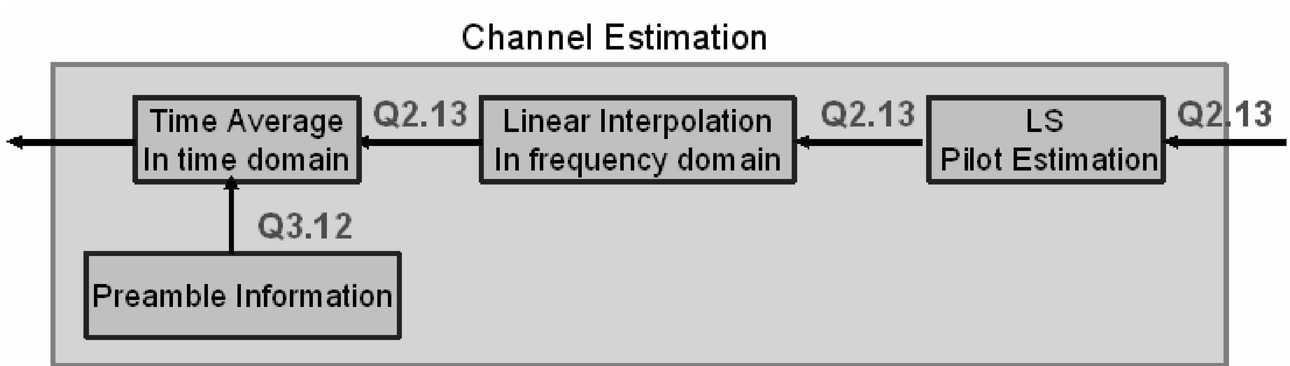
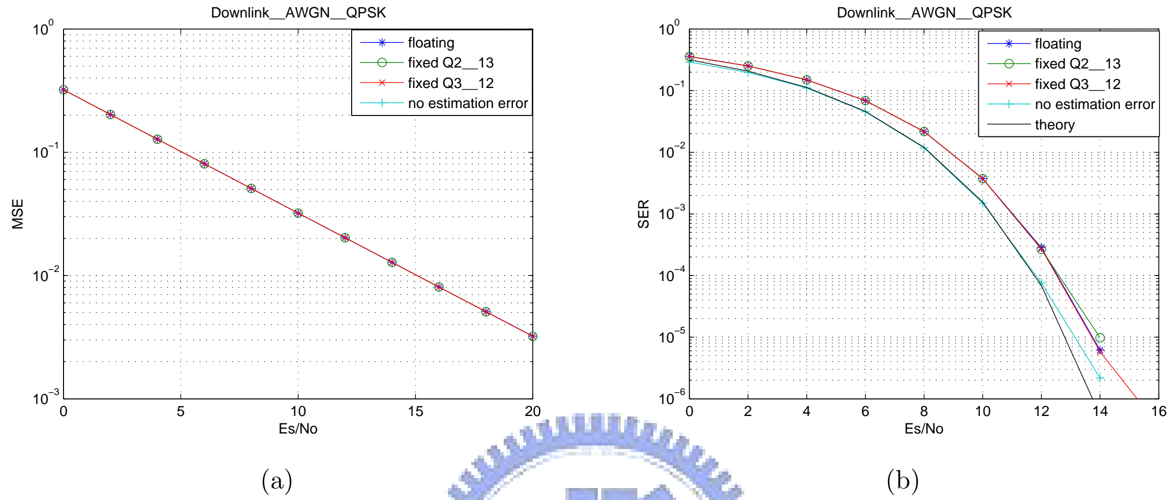
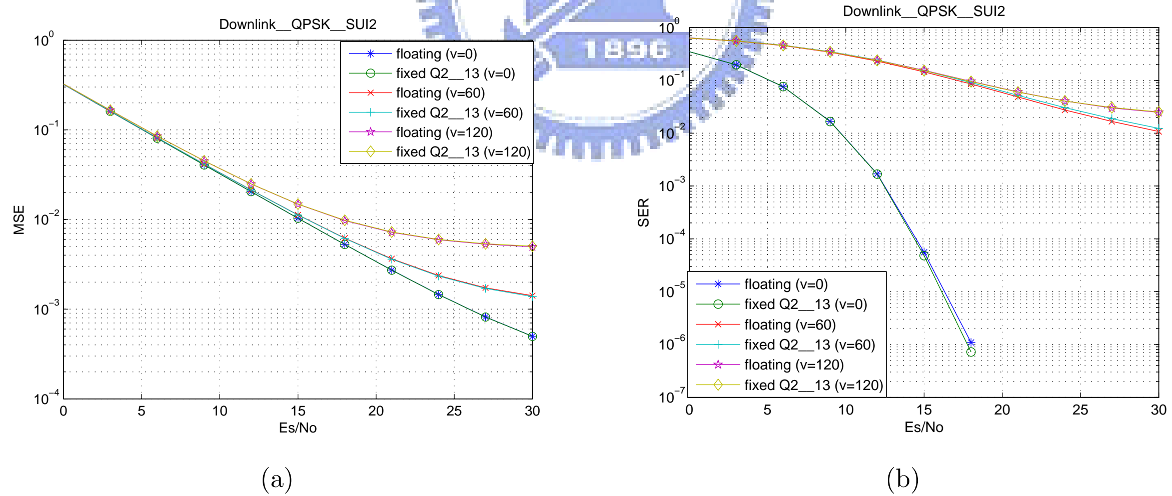


Figure 5.24: Fixed-point data formats in channel estimation of our design.



(a) (b)

Figure 5.25: Fixed-point computation of advanced four-point cluster linear interpolation in AWGN. (a) MSE. (b) SER.



(a) (b)

Figure 5.26: Fixed-point computation of advanced four-point cluster linear interpolation in SUI-2. (a) MSE. (b) SER.

Table 5.2: OFDMA DL DSP Loading for Channel Estimation in 2048-FFT, BW: 20 MHz

Symbol Number	Cycles	DSP Loading
preamble	16425	0.164
1st data symbol	560	0.006
2nd data symbol	8365	0.084
3rd–23rd data symbols (average)	4029	0.040
24th data symbol	3832+3094 =6926	0.069
average	7261	0.073

5.4.3 DSP Simulation Loading

The last part of our work is to do DSP implementation. We use CCS to simulate and get the cycle count we need. Table 5.2 shows the needed cycles and DSP loading for each symbol in a DL subframe. The cycles for estimating preamble is the highest because of calculating every subcarrier. In the first symbol data, we only calculate the channel response of pilots and store them for the next symbol. After receiving the second data symbol, we can get the channel response of the first data symbol. Here the cycles of the second data symbol is higher than other average data symbol due to taking preamble information into estimation. When getting the information of the last (24th) data symbol, we not only calculate the previous but also the last symbol's frequency response. The average cycle cost within a subframe is 7261, and it equals to 0.073 DSP loading as symbol time is 102.86 μs, and BW is 20 MHz in 2048-FFT.

```
#define Q1_14 short
#define Q2_13 short
// Though same type (short), using different types to declare is much more clear
// and avoid some mistakes

#define ftoQ1_14(A) (short) (A*16384)    // float to Q1.14 (2^14=16384)
#define ftoQ2_13(A) (short) (A*8192 )    // float to Q2.13 (2^13=8192)
#define ftoQ3_12(A) (short) (A*4096 )    // float to Q3.12 (2^12=4096)

#define Q1_14tof(A) (((float) A)/16384)  // Q1.14 to float
#define Q2_13tof(A) (((float) A)/8192)   // Q2.13 to float
#define Q3_12tof(A) (((float) A)/4096)   // Q3.12 to float
```

Figure 5.27: *FIXED.H*.

5.5 Appendix

Fig. 5.27 shows the header file *FIXED.H* which we use to transform into the formats of fixed-point. Since the first transmitted symbol in a DL subframe is preamble, we do the preamble estimation first. Fig. 5.28 illustrates the Function *preamble_estimation_FIXED* in which we gain the channel response of preamble.

Function *channel_estimation_FIXED* is the main function of channel estimation. It contains two subfunctions of *pilot_extraction_FIXED* and *interpolation_FIXED*. Function *pilot_extraction_FIXED* gets the channel response at pilot subcarriers by using the LS technique, and function *interpolation_FIXED* shows using the advanced 4-point cluster linear interpolation to get the frequency response on data subcarriers. Furthermore, we use the former preamble estimation results by multiplying an weighting of 0.9.

The original C codes are shown in Fig. 5.29, Fig. 5.30, Fig. 5.31, Fig. 5.32, and Fig. 5.33. The corresponding assembly codes to each function are also listed in Fig. 5.34, Fig. 5.35, Fig. 5.36 and 5.37. Software pipelining information of function*preamble_estimation_FIXED* and *channel_estimation_FIXED* are illustrated in Fig. 5.38 and Fig. 5.39.

```
void preamble_estimation_FIXED (int SEGMENT, short *Bin_preamble, Q3_12 *preamble_out_real,
Q3_12 *preamble_out_imag, Q2_13 *preamble_response_real,  Q2_13 *preamble_response_imag)
{

  short i,preamblepilot_PUSC[561];
  Q1_14 kk,aa;
  Q2_13 preamblespacing_real[560],preamblespacing_imag[560],preambledelta_real[560],preambledelta_imag[560];

  kk=ftoQ1_14(0.35355339);  //kk=1/(2*sqrt(2))=0.35355339
  aa=ftoQ1_14(0.33333333);  //aa=1/3=0.333333333333

  if(SEGMENT==0)
  {
    for(i=0;i<561;i++)
    {
       preamblepilot_PUSC[i]=SEGMENT+3*i;
       preamble_response_real[preamblepilot_PUSC[i]]
              = preamble_out_real[preamblepilot_PUSC[i]]*kk*(1-2*Bin_preamble[(i+4)])>>13;
       preamble_response_imag[preamblepilot_PUSC[i]]
              = preamble_out_imag[preamblepilot_PUSC[i]]*kk*(1-2*Bin_preamble[(i+4)])>>13;
       //Q3_12*Q1_14-->Q2_13
    }

    for(i=0;i<560;i++)
    {
       preamblespacing_real[i] = (Q2_13)preamble_response_real[preamblepilot_PUSC[i+1]]
                                -preamble_response_real[preamblepilot_PUSC[i]];
       preambledelta_real[i]=(preamblespacing_real[i]*aa)>>14;
       preamblespacing_imag[i] = (Q2_13)preamble_response_imag[preamblepilot_PUSC[i+1]]
                                -preamble_response_imag[preamblepilot_PUSC[i]];
       preambledelta_imag[i]=(preamblespacing_imag[i]*aa)>>14;
       //Q2_13*Q1_14-->Q2_13

       preamble_response_real[(preamblepilot_PUSC[i]+1)]
               = (Q2_13)preamble_response_real[preamblepilot_PUSC[i]]+preambledelta_real[i];
       preamble_response_imag[(preamblepilot_PUSC[i]+1)]
               = (Q2_13)preamble_response_imag[preamblepilot_PUSC[i]]+preambledelta_imag[i];
       preamble_response_real[(preamblepilot_PUSC[i]+2)]
               = (Q2_13)preamble_response_real[preamblepilot_PUSC[i]]+2*preambledelta_real[i];
       preamble_response_imag[(preamblepilot_PUSC[i]+2)]
               = (Q2_13)preamble_response_imag[preamblepilot_PUSC[i]]+2*preambledelta_imag[i];
    }
  }
  else
  {
    for(i=0;i<560;i++)
    {
       preamblepilot_PUSC[i]=SEGMENT+3*i;
       preamble_response_real[preamblepilot_PUSC[i]]
              = preamble_out_real[preamblepilot_PUSC[i]]*kk*(1-2*Bin_preamble[(i+2)])>>13;
       preamble_response_imag[preamblepilot_PUSC[i]]
              = preamble_out_imag[preamblepilot_PUSC[i]]*kk*(1-2*Bin_preamble[(i+2)])>>13;
       //Q3_12*Q1_14-->Q2_13
    }
    for(i=0;i<559;i++)
    {
       preamblespacing_real[i] = (Q2_13)preamble_response_real[preamblepilot_PUSC[i+1]]
                                -preamble_response_real[preamblepilot_PUSC[i]];
       preambledelta_real[i]=(preamblespacing_real[i]*aa)>>14;
       preamblespacing_imag[i] = (Q2_13)preamble_response_imag[preamblepilot_PUSC[i+1]]
                                -preamble_response_imag[preamblepilot_PUSC[i]];
       preambledelta_imag[i]=(preamblespacing_imag[i]*aa)>>14;
       //Q2_13*Q1_14-->Q2_13
       preamble_response_real[(preamblepilot_PUSC[i]+1)]
               = (Q2_13)preamble_response_real[preamblepilot_PUSC[i]]+preambledelta_real[i];
       preamble_response_imag[(preamblepilot_PUSC[i]+1)]
               = (Q2_13)preamble_response_imag[preamblepilot_PUSC[i]]+preambledelta_imag[i];
       preamble_response_real[(preamblepilot_PUSC[i]+2)]
               = (Q2_13)preamble_response_real[preamblepilot_PUSC[i]]+2*preambledelta_real[i]
       preamble_response_imag[(preamblepilot_PUSC[i]+2)]
               = (Q2_13)preamble_response_imag[preamblepilot_PUSC[i]]+2*preambledelta_imag[i]
    }
    if(SEGMENT==1)
    {
           preamble_response_real[0]=preamble_response_real[1]-preambledelta_real[0];
           preamble_response_imag[0]=preamble_response_imag[1]-preambledelta_imag[0];
           preamble_response_real[1679]=preamble_response_real[1678]+preambledelta_real[559];
           preamble_response_imag[1679]=preamble_response_imag[1678]+preambledelta_imag[559];
           preamble_response_real[1680]=preamble_response_real[1678]+2*preambledelta_real[559];
           preamble_response_imag[1680]=preamble_response_imag[1678]+2*preambledelta_imag[559];
          }
          else
          {
           preamble_response_real[0]=preamble_response_real[2]-2*preambledelta_real[0];
           preamble_response_imag[0]=preamble_response_imag[2]-2*preambledelta_imag[0];
           preamble_response_real[1]=preamble_response_real[2]-preambledelta_real[0];
           preamble_response_imag[1]=preamble_response_imag[2]-preambledelta_imag[0];
           preamble_response_real[1680]=preamble_response_real[1679]+preambledelta_real[559];
           preamble_response_imag[1680]=preamble_response_imag[1679]+preambledelta_imag[559];
          }

    }

}
```

Figure 5.28: Function *preamble_estimation_FIXED*.

```
#define symbollength 24
#define Nfft 2048
#define Nused 1681
#define Npilots_PUSC_group 48
#define NGsubLeft 184
#define NGsubRight 183
#define Ndata_group 288
short channel_real_pre_FIXED[2][1681],channel_imag_pre_FIXED[2][1681];
void pilot_extraction_FIXED(int ,int *,int *,Q2_13 *,Q2_13 *,Q2_13 *,Q2_13 *);
void interpolation_FIXED(int,int,int *,int *,Q2_13 *,Q2_13 *,Q2_13 *,Q2_13 *);

void channel_estimation_FIXED(int last,int PUSC_SymbolNumber,int *data_PUSC,int *pilot_PUSC,int *wk_PUSC,
Q2_13 *channel_out_real,Q2_13 *channel_out_imag,Q2_13 *channel_response_real,Q2_13 *channel_response_imag,
Q2_13 *preamble_response_real,Q2_13 *preamble_response_imag)
 {
    int i;

    if(last==1)
    {interpolation_FIXED(last,PUSC_SymbolNumber,data_PUSC,pilot_PUSC,channel_response_real,
                        channel_response_imag,preamble_response_real,preamble_response_imag);}
    else
    {     if(PUSC_SymbolNumber%symbollength==0)
          {
             pilot_extraction_FIXED(PUSC_SymbolNumber,pilot_PUSC,wk_PUSC,
                                   channel_out_real,channel_out_imag,channel_response_real,channel_response_imag);
             for(i=0;i<Npilots_PUSC_group;i++)
             {
              channel_real_pre_FIXED[0][i]=channel_response_real[pilot_PUSC[i]];
              channel_imag_pre_FIXED[0][i]=channel_response_imag[pilot_PUSC[i]];
             }
          }
          else
          {
             pilot_extraction_FIXED(PUSC_SymbolNumber,pilot_PUSC,wk_PUSC,channel_out_real,channel_out_imag,
                                   channel_response_real,channel_response_imag);
             interpolation_FIXED(last,PUSC_SymbolNumber,data_PUSC,pilot_PUSC,channel_response_real,
                                channel_response_imag,preamble_response_real,preamble_response_imag);
          }
    }
 }
```

Figure 5.29: Function *channel_estimation_FIXED*.

```
void pilot_extraction_FIXED(int PUSC_SymbolNumber,int *pilot_PUSC,int *wk_PUSC,
                            Q2_13 *channel_out_real,     Q2_13 *channel_out_imag,
                            Q2_13 *channel_response_real,Q2_13 *channel_response_imag)
 {
    short bb;//bb=3/4;
    int i;
    bb=ftoQ1_14(0.75);
    for(i=0;i<Npilots_PUSC_group;i++)
    {
       channel_response_real[pilot_PUSC[i]]=channel_out_real[pilot_PUSC[i]]*bb*(1-2*wk_PUSC[pilot_PUSC[i]])>>14 ;
       channel_response_imag[pilot_PUSC[i]]=channel_out_imag[pilot_PUSC[i]]*bb*(1-2*wk_PUSC[pilot_PUSC[i]])>>14 ;
       //Q2_13*Q1_14-->Q2_13
    }
}
```

Figure 5.30: Function *pilot_extraction_FIXED*.

```
void interpolation_FIXED(int last,int PUSC_SymbolNumber,int *data_PUSC,int *pilot_PUSC,
                         Q2_13 *channel_response_real,Q2_13 *channel_response_imag,
                         Q2_13 *preamble_response_real,Q2_13 *preamble_response_imag)
{
    int Npilots,Nclusters,i,j;
    Q1_14 exp_weightf,exp_weightf_inv;//exp_weight=0.9
    Q2_13 delta_real[24][3],delta_imag[24][3],channel_real_now[48],channel_imag_now[48];

    exp_weightf=ftoQ1_14(0.9);
    exp_weightf_inv=ftoQ1_14(0.1);
    Nclusters=Npilots_PUSC_group/2;
    Npilots=Npilots_PUSC_group;

    if(last==0)
    {
    if(PUSC_SymbolNumber%2==1)
          {
               for(i=0;i<Npilots_PUSC_group;i++)
               {
                channel_real_now[i]=channel_response_real[pilot_PUSC[i]];
                channel_imag_now[i]=channel_response_imag[pilot_PUSC[i]];
               }
               if(PUSC_SymbolNumber==1)
               {
                 for(i=0;i<Npilots_PUSC_group;i++)
                 {
                   channel_response_real[pilot_PUSC0[i]]=channel_real_pre_FIXED[0][i];
                   channel_response_imag[pilot_PUSC0[i]]=channel_imag_pre_FIXED[0][i];
                 }
               }
               else
               {
                  for(i=0;i<Npilots_PUSC_group;i++)
                  {
                  channel_response_real[pilot_PUSC0[i]]=channel_real_pre_FIXED[0][i];
                  channel_response_imag[pilot_PUSC0[i]]=channel_imag_pre_FIXED[0][i];
                  channel_response_real[pilot_PUSC1[i]]=(channel_real_pre_FIXED[1][i]
                                                  +channel_response_real[pilot_PUSC[i]])>>1;
                  channel_response_imag[pilot_PUSC1[i]]=(channel_imag_pre_FIXED[1][i]
                                                  +channel_response_imag[pilot_PUSC[i]])>>1;
                  // >>1 = /2
                  }
               }
               for(i=0;i<Npilots_PUSC_group;i++)
               {
                channel_real_pre_FIXED[1][i]=channel_real_now[i];
                channel_imag_pre_FIXED[1][i]=channel_imag_now[i];
               }
               for(i=0;i<Nclusters;i++)
               {
                 delta_real[i][0]=(Q2_13)(channel_response_real[pilot_PUSC0[2*i]]
                                  -channel_response_real[pilot_PUSC1[2*i]])>>2;
                 delta_imag[i][0]=(Q2_13)(channel_response_imag[pilot_PUSC0[2*i]]
                                  -channel_response_imag[pilot_PUSC1[2*i]])>>2;
                  // >>2 = /4
                 for(j=1;j<4;j++)
                 {
                   channel_response_real[pilot_PUSC1[2*i]+j]=channel_response_real[pilot_PUSC1[2*i]]
                                                       +j*delta_real[i][0];
                   channel_response_imag[pilot_PUSC1[2*i]+j]=channel_response_imag[pilot_PUSC1[2*i]]
                                                       +j*delta_imag[i][0];
                 }
                 delta_real[i][1]=(Q2_13)(channel_response_real[pilot_PUSC0[2*i+1]]
                                  -channel_response_real[pilot_PUSC0[2*i]])>>2;
                 delta_imag[i][1]=(Q2_13)(channel_response_imag[pilot_PUSC0[2*i+1]]
                                  -channel_response_imag[pilot_PUSC0[2*i]])>>2;
                  // >>2 = /4
                 for(j=1;j<4;j++)
                 {
                   channel_response_real[pilot_PUSC0[2*i]+j]=channel_response_real[pilot_PUSC0[2*i]]
                                                       +j*delta_real[i][1];
                   channel_response_imag[pilot_PUSC0[2*i]+j]=channel_response_imag[pilot_PUSC0[2*i]]
                                                       +j*delta_imag[i][1];
                 }
                 delta_real[i][2]=(Q2_13)(channel_response_real[pilot_PUSC1[2*i+1]]
                                  -channel_response_real[pilot_PUSC0[2*i+1]])>>2;
                 delta_imag[i][2]=(Q2_13)(channel_response_imag[pilot_PUSC1[2*i+1]]
                                  -channel_response_imag[pilot_PUSC0[2*i+1]])>>2;
                  // >>2 = /4
```

Figure 5.31: Function *interpolation_FIXED*.

```
                for(j=1;j<6;j++)
                {
                  channel_response_real[pilot_PUSC0[2*i+1]+j]=channel_response_real[pilot_PUSC0[2*i+1]]
                                                             +j*delta_real[i][2];
                  channel_response_imag[pilot_PUSC0[2*i+1]+j]=channel_response_imag[pilot_PUSC0[2*i+1]]
                                                             +j*delta_imag[i][2];
                }
              }
          }
     else
          {
              for(i=0;i<Npilots_PUSC_group;i++)
              {
               channel_real_now[i]=channel_response_real[pilot_PUSC[i]];
               channel_imag_now[i]=channel_response_imag[pilot_PUSC[i]];
              }
              for(i=0;i<Npilots_PUSC_group;i++)
              {
                 channel_response_real[pilot_PUSC0[i]]=(channel_real_pre_FIXED[0][i]
                                                      +channel_response_real[pilot_PUSC[i]])>>1;
                 channel_response_imag[pilot_PUSC0[i]]=(channel_imag_pre_FIXED[0][i]
                                                      +channel_response_imag[pilot_PUSC[i]])>>1;
                 // >>1 = /2
                 channel_response_real[pilot_PUSC1[i]]=channel_real_pre_FIXED[1][i];
                 channel_response_imag[pilot_PUSC1[i]]=channel_imag_pre_FIXED[1][i];
              }
              for(i=0;i<Npilots_PUSC_group;i++)
              {
               channel_real_pre_FIXED[0][i]=channel_real_now[i];
               channel_imag_pre_FIXED[0][i]=channel_imag_now[i];
              }
              for(i=0;i<Nclusters;i++)
              {
                delta_real[i][0]=(Q2_13)(channel_response_real[pilot_PUSC0[2*i]]
                                 -channel_response_real[pilot_PUSC1[2*i]])>>2;
                delta_imag[i][0]=(Q2_13)(channel_response_imag[pilot_PUSC0[2*i]]
                                 -channel_response_imag[pilot_PUSC1[2*i]])>>2;
                // >>2 = /4
                for(j=1;j<4;j++)
                {
                  channel_response_real[pilot_PUSC1[2*i]+j]=channel_response_real[pilot_PUSC1[2*i]]
                                                           +j*delta_real[i][0];
                  channel_response_imag[pilot_PUSC1[2*i]+j]=channel_response_imag[pilot_PUSC1[2*i]]
                                                           +j*delta_imag[i][0];
                }
                delta_real[i][1]=(Q2_13)(channel_response_real[pilot_PUSC0[2*i+1]]
                                 -channel_response_real[pilot_PUSC0[2*i]])>>2;
                delta_imag[i][1]=(Q2_13)(channel_response_imag[pilot_PUSC0[2*i+1]]
                                 -channel_response_imag[pilot_PUSC0[2*i]])>>2;
                // >>2 = /4
                for(j=1;j<4;j++)
                {
                  channel_response_real[pilot_PUSC0[2*i]+j]=channel_response_real[pilot_PUSC0[2*i]]
                                                           +j*delta_real[i][1];
                  channel_response_imag[pilot_PUSC0[2*i]+j]=channel_response_imag[pilot_PUSC0[2*i]]
                                                           +j*delta_imag[i][1];
                }

                delta_real[i][2]=(Q2_13)(channel_response_real[pilot_PUSC1[2*i+1]]
                                 -channel_response_real[pilot_PUSC0[2*i+1]])>>2;
                delta_imag[i][2]=(Q2_13)(channel_response_imag[pilot_PUSC1[2*i+1]]
                                 -channel_response_imag[pilot_PUSC0[2*i+1]])>>2;
                // >>2 = /4
                for(j=1;j<6;j++)
                {
                  channel_response_real[pilot_PUSC0[2*i+1]+j]=channel_response_real[pilot_PUSC0[2*i+1]]
                                                             +j*delta_real[i][2];
                  channel_response_imag[pilot_PUSC0[2*i+1]+j]=channel_response_imag[pilot_PUSC0[2*i+1]]
                                                             +j*delta_imag[i][2];
                }
              }
          }
}
```

Figure 5.32: Function *interpolation_FIXED* (cont.).

```
else //(last==1)
        {
          for(i=0;i<Npilots_PUSC_group;i++)
           {
                channel_response_real[pilot_PUSC0[i]]=channel_real_pre_FIXED[0][i];
                channel_response_imag[pilot_PUSC0[i]]=channel_imag_pre_FIXED[0][i];
                channel_response_real[pilot_PUSC1[i]]=channel_real_pre_FIXED[1][i];
                channel_response_imag[pilot_PUSC1[i]]=channel_imag_pre_FIXED[1][i];
           }
          for(i=0;i<Nclusters;i++)
            {
                delta_real[i][0]=(Q2_13)(channel_response_real[pilot_PUSC0[2*i]]
                                 -channel_response_real[pilot_PUSC1[2*i]])>>2;
                delta_imag[i][0]=(Q2_13)(channel_response_imag[pilot_PUSC0[2*i]]
                                -channel_response_imag[pilot_PUSC1[2*i]])>>2;
                // >>2 = /4
                for(j=1;j<4;j++)
                {
                  channel_response_real[pilot_PUSC1[2*i]+j]=channel_response_real[pilot_PUSC1[2*i]]
                                                            +j*delta_real[i][0];
                  channel_response_imag[pilot_PUSC1[2*i]+j]=channel_response_imag[pilot_PUSC1[2*i]]
                                                            +j*delta_imag[i][0];
                }

                delta_real[i][1]=(Q2_13)(channel_response_real[pilot_PUSC0[2*i+1]]
                                 -channel_response_real[pilot_PUSC0[2*i]])>>2;
                delta_imag[i][1]=(Q2_13)(channel_response_imag[pilot_PUSC0[2*i+1]]
                                 -channel_response_imag[pilot_PUSC0[2*i]])>>2;
                // >>2 = /4
                for(j=1;j<4;j++)
                {
                  channel_response_real[pilot_PUSC0[2*i]+j]=channel_response_real[pilot_PUSC0[2*i]]
                                                            +j*delta_real[i][1];
                  channel_response_imag[pilot_PUSC0[2*i]+j]=channel_response_imag[pilot_PUSC0[2*i]]
                                                            +j*delta_imag[i][1];
                }
               delta_real[i][2]=(Q2_13)(channel_response_real[pilot_PUSC1[2*i+1]]
                                -channel_response_real[pilot_PUSC0[2*i+1]])>>2;
               delta_imag[i][2]=(Q2_13)(channel_response_imag[pilot_PUSC1[2*i+1]]
                                -channel_response_imag[pilot_PUSC0[2*i+1]])>>2;
               // >>2 = /4
               for(j=1;j<6;j++)
               {
                 channel_response_real[pilot_PUSC0[2*i+1]+j]=channel_response_real[pilot_PUSC0[2*i+1]]
                                                            +j*delta_real[i][2];
                 channel_response_imag[pilot_PUSC0[2*i+1]+j]=channel_response_imag[pilot_PUSC0[2*i+1]]
                                                            +j*delta_imag[i][2];
               }
          }
    }
//exponential averaging for preamble---------------------------------------------------
    if(PUSC_SymbolNumber%symbollength==1)
    {
        for(i=0;i<Ndata_group;i++)
        {
         channel_response_real[data_PUSC[i]]=(exp_weightf_inv*channel_response_real[data_PUSC[i]]
                                            +exp_weightf*preamble_response_real[data_PUSC[i]])>>14;
         channel_response_imag[data_PUSC[i]]=(exp_weightf_inv*channel_response_imag[data_PUSC[i]]
                                            +exp_weightf*preamble_response_imag[data_PUSC[i]])>>14;
          //Q2_13*Q1_14-->Q2_13
        }
        for(i=0;i<Npilots_PUSC_group;i++)
        {
         channel_response_real[pilot_PUSC[i]]=(exp_weightf_inv*channel_response_real[pilot_PUSC[i]]
                                             +exp_weightf*preamble_response_real[pilot_PUSC[i]])>>14;
         channel_response_imag[pilot_PUSC[i]]=(exp_weightf_inv*channel_response_imag[pilot_PUSC[i]]
                                             +exp_weightf*preamble_response_imag[pilot_PUSC[i]])>>14;
         //Q2_13*Q1_14-->Q2_13
        }
    }

}
```

Figure 5.33: Function *interpolation_FIXED* (cont.).

```
;******************************************************************************
;* FUNCTION NAME: _preamble_estimation_FIXED                                  *
;*                                                                            *
;*   Regs Modified     : A0,A3,A4,A5,A6,A7,A8,A9,B0,B1,B2,B3,B4,B5,B6,B7,B8,  *
;*                           B9,B10,SP,A16,A17,A18,A19,A20,A21,A22,A23,A24,   *
;*                           A25,A26,A27,A28,A29,A30,A31,B16,B17,B18,B19,B20, *
;*                           B21,B22,B23,B24,B25,B26,B27,B28,B29,B30,B31      *
;*   Regs Used         : A0,A3,A4,A5,A6,A7,A8,A9,B0,B1,B2,B3,B4,B5,B6,B7,B8,  *
;*                           B9,B10,DP,SP,A16,A17,A18,A19,A20,A21,A22,A23,A24,*
;*                           A25,A26,A27,A28,A29,A30,A31,B16,B17,B18,B19,B20, *
;*                           B21,B22,B23,B24,B25,B26,B27,B28,B29,B30,B31      *
;*   Local Frame Size  : 0 Args + 5612 Auto + 8 Save = 5620 byte              *
;******************************************************************************
;* Using -g (debug) with optimization (-o3) may disable key optimizations!    *
;******************************************************************************
_preamble_estimation_FIXED:
;** --------------------------------------------------------------------------*
        .dwcfa  0x0e, 0
        .dwcfa  0x09, 126, 19
           ADDK    .S2     -5624,SP          ; |11|
        .dwcfa  0x0e, 5624
           STW     .D2T2   B10,*+SP(5624)    ; |11|
        .dwcfa  0x80, 26, 0
           STW     .D2T2   B3,*+SP(5620)     ; |11|
        .dwcfa  0x80, 19, 1

           ADD     .L2X    10,A4,B24
           ADD     .L2X    8,A4,B25

           ADD     .L2     10,SP,B27
||         ADD     .S2     12,SP,B21
||         MV      .D2X    A4,B3             ; |11|
||         ADD     .L1     14,A4,A26
||         MV      .S1     A6,A19            ; |11|
||         MV      .D1     A8,A21            ; |11|

           MV      .L2     B6,B16            ; |11|
||         ADD     .S2     8,SP,B26
||         ADD     .D2     14,SP,B22
||         MVK     .S1     0xb5,A17
||         ADD     .L1     12,A4,A25
||         MV      .D1X    B4,A18            ; |11|

           ZERO    .L2     B20
           ADD     .L2     1,B20,B23         ; |26| (P) <0,0>

           MPY     .M2     B23,3,B7          ; |26| (P) <0,1>
||         ADD     .L2     2,B20,B31         ; |26| (P) <0,0>
||         ZERO    .L1     A24

           MPY     .M2     B31,3,B2          ; |26| (P) <0,2>
||         ADD     .L1     3,A24,A7          ; |26| (P) <0,0>

           MPY     .M1     A7,3,A31          ; |26| (P) <0,1>
||         STH     .D2T2   B7,*+B27[B20]     ; |26| (P) <0,4>

           MPY     .M1     A24,3,A29         ; |26| (P) <0,0>
||         STH     .D2T2   B2,*+B21[B20]     ; |26| (P) <0,5>

           STH     .D2T1   A31,*+B22[B20]    ; |26| (P) <0,8>
||         MPY     .M1     A24,3,A3          ; |27| (P) <0,2>
||         MVC     .S2     CSR,B10

           STH     .D2T1   A29,*+B26[B20]    ; |26| (P) <0,10>
||         AND     .L2     -2,B10,B4
||         MV      .L1X    B22,A28

           LDH     .D2T2   *+B27[B20],B30    ; |28| (P) <0,13>
||         LDH     .D1T1   *+A28[A24],A9     ; |28| (P) <0,14>
||         EXT     .S1     A3,16,16,A4       ; |27| (P) <0,14>
||         MVC     .S2     B4,CSR            ; interrupts off
||         MPY     .M2     B31,3,B5          ; |27| (P) <0,14>
```

Figure 5.34: Assembly code of function *preamble_estimation_FIXED*.

```
;******************************************************************************
;* FUNCTION NAME: _channel_estimation_FIXED                                   *
;*                                                                            *
;*   Regs Modified     : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,B0,B1,B2,B3,B4,B5, *
;*                           B6,B7,B8,B9,B10,SP,A16,A17,A18,A19,A20,A21,A22,  *
;*                           A23,A24,A25,A26,A27,A28,A29,A30,A31,B16,B17,B18, *
;*                           B19,B20,B21,B22,B23,B24,B25,B26,B27,B28,B29,B30, *
;*                           B31                                              *
;*   Regs Used         : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A12,B0,B1,B2,B3,B4,*
;*                           B5,B6,B7,B8,B9,B10,B12,DP,SP,A16,A17,A18,A19,A20,*
;*                           A21,A22,A23,A24,A25,A26,A27,A28,A29,A30,A31,B16, *
;*                           B17,B18,B19,B20,B21,B22,B23,B24,B25,B26,B27,B28, *
;*                           B29,B30,B31                                      *
;*   Local Frame Size  : 0 Args + 0 Auto + 12 Save = 12 byte                  *
;******************************************************************************
;* Using -g (debug) with optimization (-o3) may disable key optimizations!    *
;******************************************************************************
_channel_estimation_FIXED:
;** --------------------------------------------------------------------------*
        .dwcfa  0x0e, 0
        .dwcfa  0x09, 126, 19
           STW     .D2T2   B10,*SP--(16)     ; |21|
        .dwcfa  0x0e, 16
        .dwcfa  0x80, 26, 0
           STW     .D2T2   B3,*+SP(12)       ; |21|
        .dwcfa  0x80, 19, 1
           STW     .D2T1   A10,*+SP(8)       ; |21|
        .dwcfa  0x80, 10, 2

           MV      .L1X    B4,A25            ; |21|
           MV      .L1X    B10,A24           ; |21|

           MV      .L2X    A8,B6             ; |21|
||         MV      .L1X    B6,A6             ; |21|
||         MV      .S1     A6,A22            ; |21|

           MV      .L2X    A4,B1             ; |21|
||         MV      .L1X    B8,A8             ; |21|
||         MV      .S2     B12,B10           ; |21|
||         MV      .S1     A10,A23           ; |21|

   [ B1]   BNOP    .S1     L52,5             ; |24|
           ; BRANCHCC OCCURS {L52}           ; |24|
```

Figure 5.35: Assembly code of function *channel_estimation_FIXED*.

```
;****************************************************************************
;* FUNCTION NAME: _pilot_extraction_FIXED                                    *
;*                                                                           *
;*   Regs Modified     : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,B0,B4,B5,B6,B7,B8,B9,  *
;*                           A16,A17,A18,A19,A20,A21,B16,B17,B18,B19,B20,B21, *
;*                           B22,B23,B24,B25,B26,B27,B28,B29,B30             *
;*   Regs Used         : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,B0,B3,B4,B5,B6,B7, *
;*                           B8,B9,DP,SP,A16,A17,A18,A19,A20,A21,B16,B17,B18, *
;*                           B19,B20,B21,B22,B23,B24,B25,B26,B27,B28,B29,B30  *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                   *
;****************************************************************************
;* Using -g (debug) with optimization (-o3) may disable key optimizations!    *
;****************************************************************************
_pilot_extraction_FIXED:
;** --------------------------------------------------------------------------*
        .dwcfa  0x0e, 0
        .dwcfa  0x09, 126, 19

           MV      .L1X    B8,A6             ; |251|
||         MV      .S1     A6,A7             ; |251|

           MV      .L1X    B6,A5             ; |251|
||         SUB     .L2     B4,8,B20
||         MV      .S2X    A10,B28           ; |251|

           MV      .L2X    A8,B27
||         MVK     .D2     1,B4              ; |257|
||         MVC     .S2     CSR,B30

           MV      .L1X    B20,A4
||         MV      .L2X    A6,B24
||         AND     .S2     -2,B30,B5
||         MVK     .D2     3,B26             ; |257|
||         MVK     .S1     0x2,A2            ; init prolog collapse predicate

           MVC     .S2     B5,CSR            ; interrupts off
||         MV      .L1X    B4,A3
||         MV      .L2X    A5,B23
||         MVK     .S1     23,A0             ; |255|
||         MVK     .D1     0x3,A1            ; init prolog collapse predicate
||         MVK     .D2     0x1,B0            ; init prolog collapse predicate
```

Figure 5.36: Assembly code of function *pilot_extraction_FIXED*.

```
;******************************************************************************
;* FUNCTION NAME: _interpolation_FIXED                                        *
;*                                                                            *
;*   Regs Modified     : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,   *
;*                           A15,B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,SP,A16,*
;*                           A17,A18,A19,A20,A21,A22,A23,A24,A25,A26,A27,A28, *
;*                           A29,A30,A31,B16,B17,B18,B19,B20,B21,B22,B23,B24, *
;*                           B25,B26,B27,B28,B29,B30,B31                      *
;*   Regs Used         : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,   *
;*                           A15,B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,DP,SP, *
;*                           A16,A17,A18,A19,A20,A21,A22,A23,A24,A25,A26,A27, *
;*                           A28,A29,A30,A31,B16,B17,B18,B19,B20,B21,B22,B23, *
;*                           B24,B25,B26,B27,B28,B29,B30,B31                  *
;*   Local Frame Size  : 0 Args + 484 Auto + 36 Save = 520 byte               *
;******************************************************************************
;* Using -g (debug) with optimization (-o3) may disable key optimizations!    *
;******************************************************************************
_interpolation_FIXED:
;** --------------------------------------------------------------------------*
        .dwcfa  0x0e, 0
        .dwcfa  0x09, 126, 19

           MV      .L1X    SP,A31            ; |49|
||         ADDK    .S2     -520,SP           ; |49|

           STW     .D2T1   A15,*+SP(520)
        .dwcfa  0x80, 15, 0
           STW     .D2T2   B10,*+SP(512)
           STW     .D2T2   B11,*+SP(516)
        .dwcfa  0x80, 26, 1
        .dwcfa  0x80, 27, 2
           STW     .D2T2   B3,*+SP(508)
        .dwcfa  0x80, 19, 3
           STW     .D1T1   A14,*-A31(16)
        .dwcfa  0x80, 14, 4
           STDW    .D1T1   A13:A12,*-A31(24)
        .dwcfa  0x80, 12, 5
        .dwcfa  0x80, 13, 6
           STW     .D2T1   A10,*+SP(488)
           STW     .D2T1   A11,*+SP(492)
        .dwcfa  0x80, 10, 7
        .dwcfa  0x80, 11, 8

           MV      .L1     A6,A15            ; |49|
||         MV      .L2X    A10,B3            ; |49|
||         MV      .S1     A4,A0             ; |49|
||         MV      .S2     B4,B2             ; |49|
||         MV      .D2     B6,B1             ; |49|
||         MV      .D1X    B8,A23            ; |49|

           MV      .L2X    A8,B18            ; |49|
        .dwpsn  "D:\CCS_UL_ChEst 1010\Channel_estimation_FIXED.c",64,5

   [ A0]   B       .S2     L36               ; |64|
|| [ A0]   MVKL    .S1     _channel_imag_pre_FIXED,A3

   [ A0]   MVKH    .S1     _channel_imag_pre_FIXED,A3
|| [ A0]   MVK     .S2     3354,B4

   [ A0]   MVK     .S2     0x691,B7

   [ A0]   MVK     .S2     0x693,B9
|| [ A0]   ADD     .L2X    B4,A3,B20

   [ A0]   MVK     .S2     0x694,B16
   [ A0]   MVK     .S2     0x692,B8
           ; BRANCHCC OCCURS {L36}           ; |64|
;** --------------------------------------------------------------------------*
        .dwpsn  "D:\CCS_UL_ChEst 1010\Channel_estimation_FIXED.c",66,5
           SHRU    .S2     B2,31,B4          ; |66|
           ADD     .L2     B2,B4,B4          ; |66|
           AND     .L2     -2,B4,B4          ; |66|
           SUB     .L2     B2,B4,B4          ; |66|
           CMPEQ   .L2     B4,1,B0           ; |66|

   [ B0]   BNOP    .S1     L18,4             ; |66|
|| [ B0]   SUB     .D2     B1,16,B6

   [!B0]   SUB     .D2     B1,16,B6
           ; BRANCHCC OCCURS {L18}           ; |66|
;** --------------------------------------------------------------------------*
        .dwpsn  "D:\CCS_UL_ChEst 1010\Channel_estimation_FIXED.c",133,24
           MVK     .S1     384,A3
||         MVC     .S2     CSR,B20
||         MV      .L1     A23,A6
||         MV      .L2     B18,B5
||         MVK     .D1     0x1,A1            ; init prolog collapse predicate
||         ADD     .D2     4,B6,B8

           MVK     .S2     288,B4
||         AND     .L2     -2,B20,B31
||         ADD     .L1X    A3,SP,A5
||         MVK     .S1     11,A0             ; |133|

           ADD     .L2     B4,SP,B9
||         ADD     .D2X    A3,SP,B4
||         MVC     .S2     B31,CSR           ; interrupts off
||         MV      .L1X    B6,A3

           ADD     .L2     2,B4,B7
```

Figure 5.37: Assembly code of function *interpolation_FIXED*.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 24
;*      Loop opening brace source line   : 25
;*      Loop closing brace source line   : 30
;*      Loop Unroll Multiple             : 4x
;*      Known Minimum Trip Count         : 140
;*      Known Maximum Trip Count         : 140
;*      Known Max Trip Count Factor      : 140
;*      Loop Carried Dependency Bound(^) : 19
;*      Unpartitioned Resource Bound     : 16
;*      Partitioned Resource Bound(*)    : 16
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     6        7
;*      .D units                    16*      16*
;*      .M units                    12       12
;*      .X cross paths               2        1
;*      .T address paths            14       14
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          5        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             3        4
;*      Bound(.L .S .D .LS .LSD)     9        9
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 19 Did not find schedule
;*         ii = 20 Did not find schedule
;*         ii = 21 Did not find schedule
;*         ii = 22 Did not find schedule
;*         ii = 23 Did not find schedule
;*         ii = 24 Schedule found with 2 iterations in parallel
;*      Done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      For further improvement on this loop, try option -mh4
;*
;*      Minimum safe trip count     : 2 (after unrolling)
;*----------------------------------------------------------------------------*
```

Figure 5.38: Software pipelining information of function *preamble_estimation_FIXED*.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 34
;*      Loop opening brace source line   : 35
;*      Loop closing brace source line   : 38
;*      Known Minimum Trip Count         : 48
;*      Known Maximum Trip Count         : 48
;*      Known Max Trip Count Factor      : 48
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                    0        0
;*      .S units                    1        0
;*      .D units                    3*       3*
;*      .M units                    0        0
;*      .X cross paths              0        0
;*      .T address paths            3*       3*
;*      Long read paths             0        0
;*      Long write paths            0        0
;*      Logical  ops (.LS)          0        0     (.L or .S unit)
;*      Addition ops (.LSD)         0        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)            1        0
;*      Bound(.L .S .D .LS .LSD)    2        1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 4 iterations in parallel
;*      Done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 2
;*      Minimum required memory pad : 0 bytes
;*      For further improvement on this loop, try option -mh12
;*      Minimum safe trip count     : 3
;*----------------------------------------------------------------------------*
```

Figure 5.39: Software pipelining information of function *channel_estimation_FIXED*.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we presented several channel estimation methods for OFDMA uplink and downlink. To do the channel estimation, first, we used LS estimator to estimate the channel frequency response on the pilot subcarriers. Second, interpolations was used to get a rough channel estimation in the frequency domain. Third, we combined the rough estimation with some time domain improvement techniques. In uplink simulation, we used tile linear interpolation and in downlink simulation, we tried two-point, four-point and advanced four-point cluster linear interpolation.

In the case of uplink transmission, it showed that the performance of no using tile exponential averaging was better than with weighting $w = 0.9$ in mutipath channels, such as SUI-2. As for downlink transmission, although advanced 4-point cluster linear interpolation gave us a symbol latency, it had the best performance among all used methods in SUI-2. We also showed the cluster analysis on SER and MSE performance in this thesis.

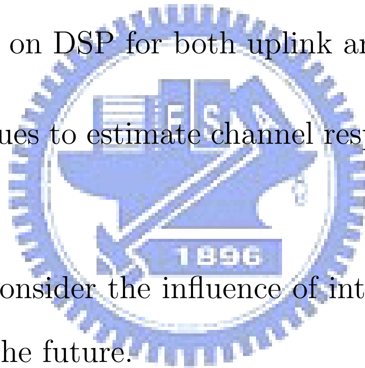
Our last work was the DSP implementation, and we implemented both uplink and downlink on TI's chip. To achieve the real-time channel estimation on CCS, we replaced all operations into 16-bit fixed point operation. We also compared the results of floating and

fixed operation. The DSP loading for 2048-FFT, 20MHz BW, and 10 used subchannels in uplink transmission is 0.016; and the average DSP cost for using major group 0, 2048-FFT, 20MHz in downlink transmission was 0.073.

6.2 Potential Future Work

There are several possible extensions for our research:

- Consider the transmission of FUSC mode in downlink, propose channel estimation method and put them on DSP.
- Optimize the performance on DSP for both uplink and downlink.
- Try other kinds of techniques to estimate channel response on pilots for less estimated errors.
- In this thesis, we do not consider the influence of intercarrier interference. The simulation can be involved in the future.



Bibliography

[1] Hongxiang Li and Hui Liu, "An analysis on uplink OFDMA optimality," *in Proc. IEEE VTC*, vol. 3, 2006, pp. 1339–1343.

[2] Liangshan Ma and Dongyan Jia, "The competition and cooperation of WiMAX, WLAN and 3G," *Inter. Conf. Applica. Sys., Mobile Tech.*, Nov. 15-17, 2005, pp. 1–5.

[3] Man-On Pun, Michele Morelli, and C.-C. Jay Kuo, "Maximum-likelihood synchronization and channel estimation for OFDMA uplink transmissions," *IEEE Trans. Commun.*, vol. 54, no. 4, pp. 726–736, April 2006.

[4] Lior Eldar, M. R. Raghavendra, S. Bhashyam, Ron Bercovich, and K. Giridhar, "Parametric channel estimation for pseudo-random user-allocation in uplink OFDMA," *IEEE Int. Conf. Commun.*, 2006, vol. 7, pp. 3035–3039.

[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 24, 2004.

[6] IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor 1-2005, *IEEE Standard for Local and metropolitan area networks–Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems–Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands and Corrigendum 1.* New York: IEEE, Feb. 28, 2006.

[7] Texas Instruments, *TMS320C6000 CPU and Instruction Set.* Literature number SPRU189F, Oct. 2000.

[8] Texas Instruments, *TMS320C6000 DSP Cache Users Guide.* Literature number SPRU656A, May. 2003.

[9] Texas Instruments, *Code Composer Studio User's Guide.* Literature number SPRU328B, Feb. 2000.

[10] Texas Instruments, *TMS320C6000 Code Composer Studio Getting Started Guide.* Literature number SPRU509D, Aug. 2003.

[11] Texas Instruments, *TMS320C64x DSP Library Programmer's Reference.* Literature number SPRU565B, Oct. 2003.

[12] Texas Instruments, *TMS320C6000 Programmer's Guide.* Literature number SPRU198G, Oct. 2002.

[13] M.-H. Hsieh, "Synchronization and channel estimation techniques for OFDM systems," Ph.D. dissectation, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., May 1998.

[14] S. Coleri, M. Ergen, A. Puri, and A. Bahai, "Channel estimation techniques based on pilot arrangement in OFDM systems," *IEEE Trans. Broadcasting,* vol. 48, no. 3, pp. 223–229, Sep. 2002.

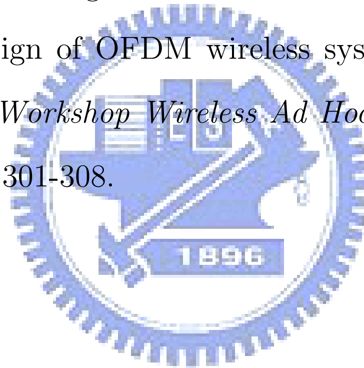
[15] T. S. Rappaport, *Wireless Communications Principles and Practice.* Upper Saddle River, New Jersey: Prentice Hall, 1996.

[16] V. Erceg et al., "Channel models for fixed wireless applications," IEEE 802.16.3c-01/29r4, July 2001.

[17] Chih-Chieh Wang, "Research in channel estimation techniques and DSP implementation for IEEE 802.16e OFDM uplink and OFDMA downlink," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2006.

[18] Ruu-Ching Chen, "Techniques for the DSP software implementation of IEEE 802.16a TDD OFDMA downlink pilot-symbol-aided channel estimation," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2005.

[19] Tien-Hsiang Lo, Kun-Chien Hung and David W. Lin, "Role of channel estimation in physical layer protocol design of OFDM wireless systems and relay-type cooperative communication," in *Proc. Workshop Wireless Ad Hoc Sensor Networks*, Chungli, Taiwan, ROC, Aug. 2006, pp. 301-308.



作者簡歷

學生王依翎，民國七十一年二月出生於台灣台南市。民國九十四年六月畢業於國立成功大學電機工程學系，並於同年九月進入國立交通大學電子工程研究就讀，從事通訊系統方面相關研究。民國九十六年六月取得碩士學位，碩士論文題目為『IEEE 802.16e OFDMA 上行及下行通道估測技術之探討與數位訊號處理器實現』。研究範圍與興趣包括：通訊系統、通道估測、數位信號處理等。