國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

IEEE 802.16a 分時雙工正交分頻多重進接下行導引訊號輔助式通道估測之技術與數位訊號處理器軟體實現

IEEE 802.16a TDD OFDMA Downlink Pilot-Symbol-Aided Channel Estimation: Techniques and DSP Software Implementation

研 究 生：陳汝苓

指導教授：林大衛 博士

中 華 民 國 九 十 四 年 六 月

IEEE 802.16a 分時雙工正交分頻多重進接下行導引訊號輔助式通道估測之技術與數位訊號處理器軟體實現

IEEE 802.16a TDD OFDMA Downlink Pilot-Symbol-Aided Channel Estimation: Techniques and DSP Software Implementation

研究生：陳汝芩　　Student: Ruu-Ching Chen

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

國 立 交 通 大 學

電子工程學系　電子研究所碩士班

碩士論文

A Thesis
Submitted to Department of Electronics Engineering & Institute of Electronics
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of Requirements
for the Degree of
Master of Science
in
Electronics Engineering
June 2005
Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

IEEE 802.16a 分時雙工正交分頻多重進接下行導引訊號輔助式通道估測之技術與數位訊號處理器軟體實現

研究生：陳汝芩　　　　　　　　　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

正交分頻技術近來因為能在行動環境中穩定高速傳輸而廣受注目，IEEE 802.16a 即是一個基於正交分頻多重進接技術用於無線區域網路和大都會網路的標準。

本論文主要在討論 IEEE 802.16a 下行通道估測的方法以及數位訊號處理器軟體實現。

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，因為硬體的計算方便。而內插的方法我們則研究了線性內插、二次式內插。而在用時域的資料改善的方法有下列兩種：二維內插法、以及最小平均平方差適應 (LMS adaptation)。我們的在靜態以及瑞雷通道上模擬。結合線性內插和二維內插法，我們得到較好的表現，而且運算複雜度也比較低，所以決定在數位訊號處理器軟體上實現。

我們將通道估測的技術以軟體實現在 Texas Instruments (TI) 公司製造型號為 TMS320C6416 的數位訊號處理器上（DSP）。此處理器的操作平台為 Innovative Integration 公司製名為 Quixote 的 cPCI 卡。因為我們所使用的 DSP 是專為定點運算所設計的，所以浮點數運算是很費時的。有三種方法可以加速運算速度：改變資料型態、程式語法的改良及使用 intrinsic 程式。所謂的改變資料型態就是把一開始的浮點數運算先改成 32-bit 的定點運算，再改成 16-bit 的定點運算。程式語法的改良則是把許多耗時的指令做修正，如 if-else 的指令。Intrinsic 程式是一種直接對應到 C64x 指令集的程式，可以改善我們 C 程式的表現。在依照上述步驟對原本浮點數運算的程式做改良後，我們得到了很大的進步，雖然與理論上運算的複雜度相比，成效最高只到 49%。不過在線性內插程式方面，我們至少達到了只需 0.52 個 symbol time 就能完成的速度。

IEEE 802.16a TDD OFDMA Downlink Pilot-Symbol-Aided Channel Estimation: Techniques and DSP Software Implementation

Student: Ruu-Ching Chen　　　　　　　Advisor: Dr. David W. Lin

Department of Electronics Engineering
& Institute of Electronics
National Chiao Tung University

Abstract

OFDM (orthogonal frequency division multiplexing) technique has drawn much interest recently for its robustness in the mobile transmission environment and its high transmission data rate. IEEE 802.16a is a wireless local and metropolitan area networks standard which is based on OFDMA (orthogonal frequency division multiple access) technique.

This work considers two main subjects of the downlink channel estimation under the specifications of IEEE 802.16a, the interpolation schemes and the DSP implementation.

We use LS estimator for estimations of pilot carriers because of its low computational complexity. We study the linear, the second-order interpolations in frequency domain and the LMS adaptation algorithm, the two-D interpolation in time domain. We did the simulation on both static and Rayleigh fading channels. Combination of linear interpolation and 2-D interpolation are chosen to be implemented on DSP board for its low computational complexity.

Our implementation is software-based, employing Texas Instruments' TMS320C6416 digital signal processor (DSP) housed on Innovative Integration's Quixote cPCI card. For the fixed-point DSP operation environment, floating-point operation is absolutely time-consuming. There are three ways to accelerate the DSP execution speed: changing data type, code style optimization, and using intrinsic functions. Changing data type means we replace the original floating-point operation with 32-bit fixed-point operation and then 16-bit fixed-point operation at last. Code style optimization is to modify the time-wasting parts of code, such as spared if-else instruction. Intrinsic functions are special functions that map directly to C64x instructions, to optimize our C code performance. The execution cycles of each function is improved a lot after optimized although compared with the theoretical execution cycles, the efficiency is 49% at most. At least, we reach the 0.52 multiples of real time needed per symbol in linear interpolation.

誌謝

要感謝的人太多，尤其是林老師，感謝他兩年多來對我的指導與包容，能當老師的學生是我前世修來的福氣。

此外，感謝通訊電子與訊號處理實驗室所有的成員，包含各位師長、同學、學長姐與學弟妹們。我要感謝吳俊榮學長、洪崑健學長指導與建議，還有昱昇、志凱、景中、鎮宇、、等同學，謝謝他們在這兩年來對我的幫助及帶給我歡樂。

家人對我的支持、鼓勵是我研究路上一股強大的動力，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。

陳汝芩

民國九十四年七月 於新竹

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Brief Introduction to IEEE 802.16a [1], [2]

In recent years, orthogonal frequency division multiplexing (OFDM) modulation technique has drawn much attention for its ability to deal with frequency-selective fading in high-speed wireless communication. The IEEE 802.16 standard committee has developed a group of standards for wireless metropolitan area networks (MANs). Project 802.16a is one of them. The object of this present study is the OFDMA-based interface option of this project, namely WirelessMAN-OFDMA.

The IEEE 802.16-2001 specifies the air interface of fixed (stationary) point-to-multipoint broadband wireless access systems providing multiple services. The medium access control layer is capable of supporting multiple physical layer specifications optimized for the frequency bands of application. This standard includes a particular physical layer specification applicable to systems operating between 10 and 66 GHz.

The IEEE 802.16a amends IEEE 802.16-2001 by enhancing the medium access control layer and providing additional physical layer specifications in support of broadband wireless access at frequencies from 2 to 11 GHz.

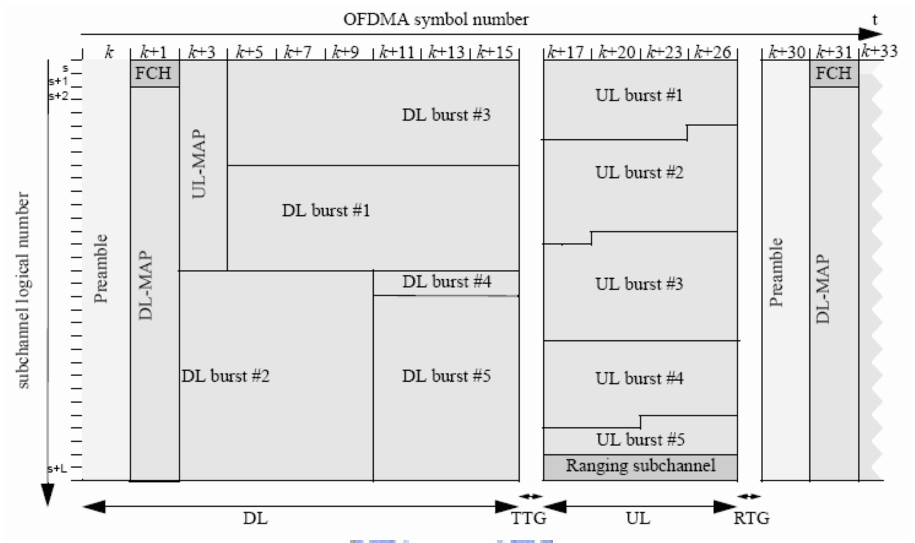
For the reason that our project started in year 2002, we have followed the specification of these two standards above. However, the IEEE 802.16 standard committee has completed a new version of the standard in year 2004, namely IEEE 802.16-2004. This standard specifies the air interface of fixed broadband wireless access (BWA) systems supporting multimedia services. The medium access control layer (MAC) supports a primarily point-to-multipoint architecture, with an optional mesh topology. The MAC is structured to support multiple physical layer (PHY) specifications, each suited to a particular operational environment. For operational frequencies of 10–66 GHz, the PHY is based on single-carrier modulation. For frequencies below 11 GHz, where propagation without a direct line of sight must be accommodated, three alternatives are provided, using OFDM, OFDMA, and single carrier modulation techniques.

Since pilot allocations are key to the study reported in this thesis, we summarize the difference between these two versions about the carrier allocations. Table 1.1 shows the pilot allocation of IEEE 802.16-2004. The variable set of pilots embedded within the symbol of each segment obeys the following rule:

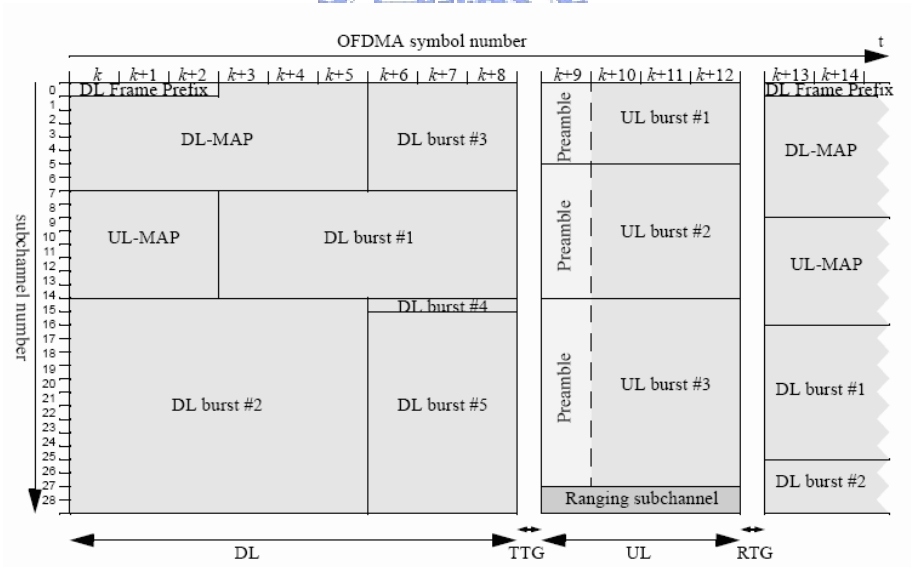
$$PilotsLocation = VariableSet\#x + 6 \cdot (FUSC\_SymbolNumber \, mod \, 2) \tag{1.1}$$

where FUSC_SymbolNumber counts the FUSC (full uasage of subchannels) symbols used in the transmission starting from 0. The arrangement is slightly different from the specification in the IEEE 802.16a-2003 (see also Fig.2.3). We have four kinds of variable location pilot arrangements in 802.16a but there are only two kinds in IEEE 802.16-2004.

As to the frame structure, the IEEE 802.16-2004 also made modification to it. From Fig. 1.1(a) we can see that in IEEE 802.16-2004, each frame begins with a preamble followed by a downlink transmission period and an uplink transmission period. This is quite different from the frame structure in IEEE 802.16a-2003, shown in Fig. 1.1(b), where preamble is used only in the uplink subframe.



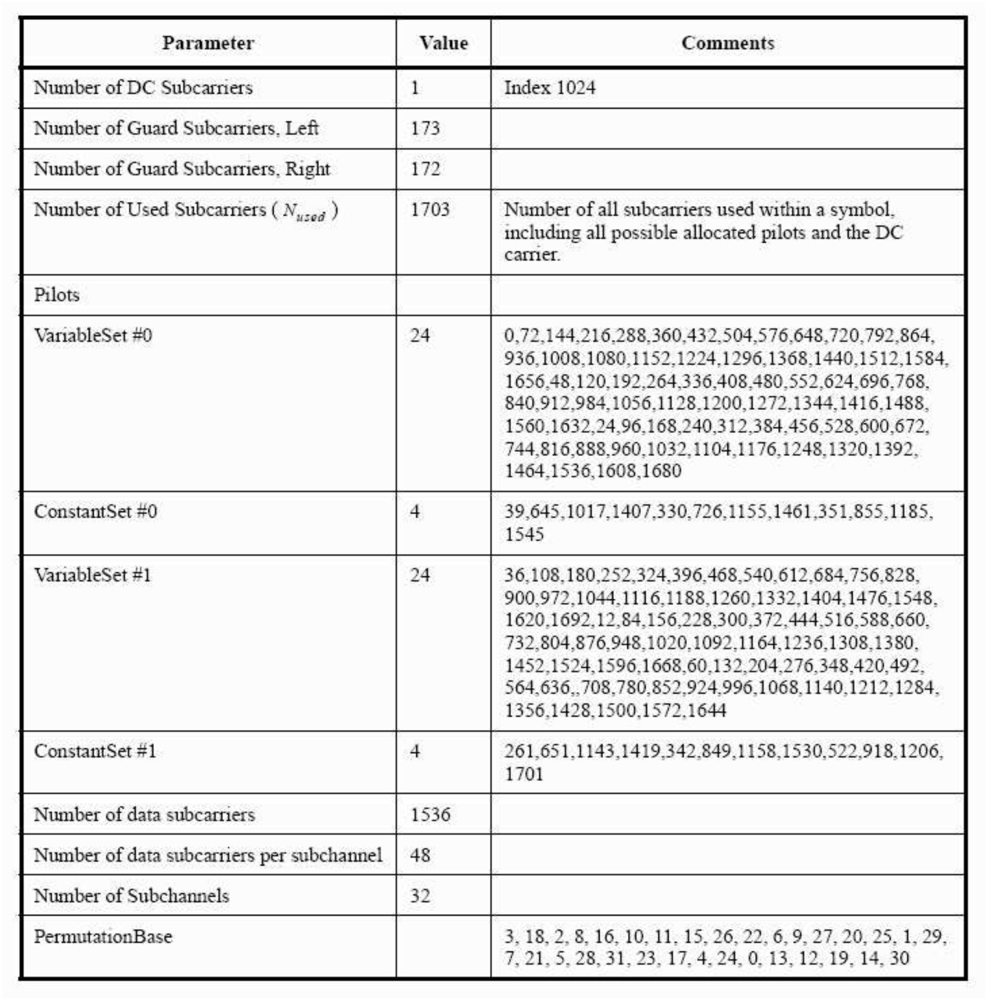
(a)



(b)

Figure 1.1: (a) Frame structure in IEEE 802.16-2004 [1]. (b) Frame structure in IEEE 802.16a-2003 [2].

Table 1.1: Carrier Allocation in the OFDMA DL (from [1])



Parameter	Value	Comments
Number of DC Subcarriers	1	Index 1024
Number of Guard Subcarriers, Left	173	
Number of Guard Subcarriers, Right	172	
Number of Used Subcarriers (N_{used})	1703	Number of all subcarriers used within a symbol, including all possible allocated pilots and the DC carrier.
Pilots		
VariableSet #0	24	0,72,144,216,288,360,432,504,576,648,720,792,864, 936,1008,1080,1152,1224,1296,1368,1440,1512,1584, 1656,48,120,192,264,336,408,480,552,624,696,768, 840,912,984,1056,1128,1200,1272,1344,1416,1488, 1560,1632,24,96,168,240,312,384,456,528,600,672, 744,816,888,960,1032,1104,1176,1248,1320,1392, 1464,1536,1608,1680
ConstantSet #0	4	39,645,1017,1407,330,726,1155,1461,351,855,1185, 1545
VariableSet #1	24	36,108,180,252,324,396,468,540,612,684,756,828, 900,972,1044,1116,1188,1260,1332,1404,1476,1548, 1620,1692,12,84,156,228,300,372,444,516,588,660, 732,804,876,948,1020,1092,1164,1236,1308,1380, 1452,1524,1596,1668,60,132,204,276,348,420,492, 564,636,,708,780,852,924,996,1068,1140,1212,1284, 1356,1428,1500,1572,1644
ConstantSet #1	4	261,651,1143,1419,342,849,1158,1530,522,918,1206, 1701
Number of data subcarriers	1536	
Number of data subcarriers per subchannel	48	
Number of Subchannels	32	
PermutationBase		3, 18, 2, 8, 16, 10, 11, 15, 26, 22, 6, 9, 27, 20, 25, 1, 29, 7, 21, 5, 28, 31, 23, 17, 4, 24, 0, 13, 12, 19, 14, 30

1.2 Motivation of This Thesis

In high data rate transmission, the imperfectness of channels, e.g., multipaths, causes more severe trouble than in low-rate transmission in demodulation. The result of data transmission over such a channel is that each received symbol is affected somewhat by adjacent symbols, thereby bringing about a common form of interference referred to as inter-symbol-interference (ISI). Inter-symbol-interference is a major source which degrades performance in the reconstructed data at receiver. In single carrier transmission, we usually employ an time domain adaptive equalizer to solve this problem. If the channel has very long impulse response compared with symbol duration, time domain equalizer may fail to handle ISI. However, in OFDM system, ISI can be easily eliminated by inserting cyclic prefix which is longer than the maximum delay spread, at the expense of some loss in capacity.

In uncoded OFDM, we only need a frequency domain equalizer with one tap at the receiver for each subcarrier. The purpose of channel estimation is to obtain the channel response at each subcarrier. Then, we can easily obtain the equalizer coefficient, the inverse of the channel gain. In channel coded OFDM, such as that in IEEE 802.16a OFDMA, equalization is not needed, but the estimated channel response is directly useful in channel decoding. Hence in this thesis, we will investigate channel estimation methods that can be employed to the IEEE 802.16a downlink transmission.

1.3 Organization of This Thesis

The contents of this thesis are as follows. In chapter 2, we give some specifications of the IEEE 802.16a OFDMA downlink system and introduce the channel estimation approaches. In chapter 3, we describe the implementation platform, which consists of Texas Instruments' TMS320C6416 digital signal processor(DSP) on a cPCI board Quixote made by Innovative Integration. Then, in chapter 4, we discuss the performance of the proposed channel estimation method as well as its DSP implementation. At last, we will give the conclusion and potential future work in chapter 5.

Chapter 2

Channel Estimation for IEEE 802.16a OFDMA Downlink Transmission

For wideband mobile communication systems, the radio channel is usually frequency selective and time variant. Therefore, our estimation schemes combine frequency domain estimation with time domain processing. In this thesis, our algorithms for channel estimation in OFDM system are intimately related to pilot sub-carrier arrangement.

2.1 Introduction to the IEEE 802.16a TDD OFDMA System

The IEEE standard 802.16a specifies the WirelessMAN air interface for wireless metropolitan area networks. There are several system modes in 802.16a: SCa (single-carrier modulation), OFDM (orthogonal frequency-division multiplexing) and OFDMA (orthogonal frequency-division multiple access). It also supports two duplex types: TDD (time division duplex) and FDD (frequency division duplex). We consider the TDD OFDMA option. Most contents in this section are taken from [2].

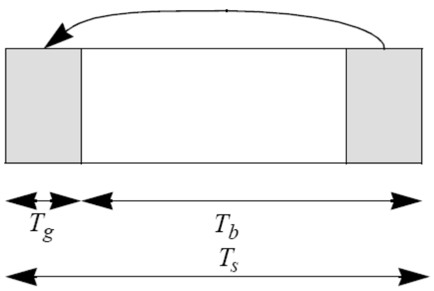


Figure 2.1: Time structure of OFDMA symbol (from [2]).

2.1.1 Generic OFDMA Symbol Description

2.1.1.1 Time Domain Description

An OFDM symbol contains the useful symbol part and the cyclic prefix (CP) part. The useful symbol time is referred to as T_b. The CP is a copy of the last T_g μs of the useful symbol period. The two together are referred to as the symbol time T_s. The ratio of CP time to useful time (T_g/T_b) that should be supported includes 1/32, 1/16, 1/8 and 1/4. In this thesis, the CP time to useful time ratio is set to 1/8. The time domain OFDMA symbol structure is shown in Fig. 2.1.

2.1.1.2 Frequency Domain Description

In frequency domain, we have 3 carrier types:

- Data carriers — for data transmission.
- Pilot carriers — for various estimation purposes.
- Null carriers — no transmission at all, for guard bands and DC carrier. (The purpose of the guard bands is to enable the signal to naturally decay and create the FFT “brick wall” shaping.)

In the OFDMA mode, active carriers are devided into subsets of carriers, and each subset is termed a subchannel. In the downlink (DL), a subchannel may be intended for different groups of receivers; similarly, a transmitter may be assigned one or more subchannels in the uplink (UL), so serveral transmitters may transmit in parallel. The symbol structure in frequency domain will be shown in detail in the following section.

2.1.2 Primitive Parameters

Four primitive parameters characterize the OFDMA symbol:

- BW. This is the nominal channel bandwidth. And it equals 10 MHz in our system simulation.
- (F_s/BW). This is the ratio of "sampling frequency" to the nominal channel bandwidth. This value is set to 8/7.
- (T_g/T_b). This is the ratio of CP time to "useful" time. We use 1/8 in our system.
- N_{FFT}. This is the number of points in the FFT. The OFDMA PHY defines this value to be 2048.

2.1.3 Derived Parameters

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

- $F_s = (F_s/BW) \cdot BW =$ sampling frequency. The value equals $10 \times 8/7 = 11.42$ MHz.
- $\triangle f = F_s/N_{FFT} =$ carrier spacing $= 5.57617$ KHz.
- $T_b = 1/\triangle f =$ useful time $= 179.33$ μs.

- $T_g = (T_g/T_b) \cdot T_b$ = CP time = 22.4 μs.

- $T_s = T_b + T_g$ = OFDM symbol time = 201.9 μs.

- $1/F_s$= sample time = 87.5657 ns.

2.1.4 Downlink Carrier Allocation

Since we focus on downlink pilot-symbol-aided channel estimation in this thesis, it is necessary to understand what the allocation of carriers is.

2.1.4.1 Pilot Allocation

The carriers allocation in a DL OFDM symbol is shown in Fig. 2.2. Null carriers are allocated in the left and right sides as well as at DC. The pilot and data carriers are termed useful carriers since they transmit useful information. The pilot tones are allocated first, and the remainder of the used carriers are divided into 32 subchannels, and then the data carriers are allocated within each subchannel.

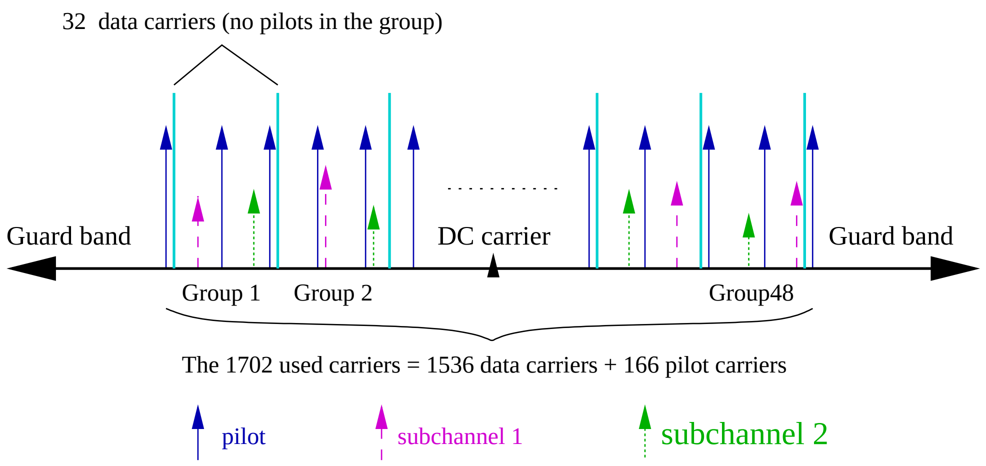

32 data carriers (no pilots in the group)
Guard band
DC carrier
Guard band
Group 1
Group 2
Group48
The 1702 used carriers = 1536 data carriers + 166 pilot carriers
pilot
subchannel 1
subchannel 2

Figure 2.2: Illustration of carrier usage in OFDMA DL (from [3]).

The pilot carriers include fixed-location pilots and variable-location pilots. The carrier indices of fixed-location pilots never change. The carrier indices of the variable-location pilots vary according to the formula $varLocPilot_k = 3L + 12P_k$, where $varLocPilot_k$ is the carrier index of a variable-location pilot, L is the symbol index that cycles through the values 0,2,1,3 periodically every 4-symbol period, and $P_k = \{0, 1, 2, ..., 141\}$. The pilot carriers allocation map is shown in Fig. 2.3.

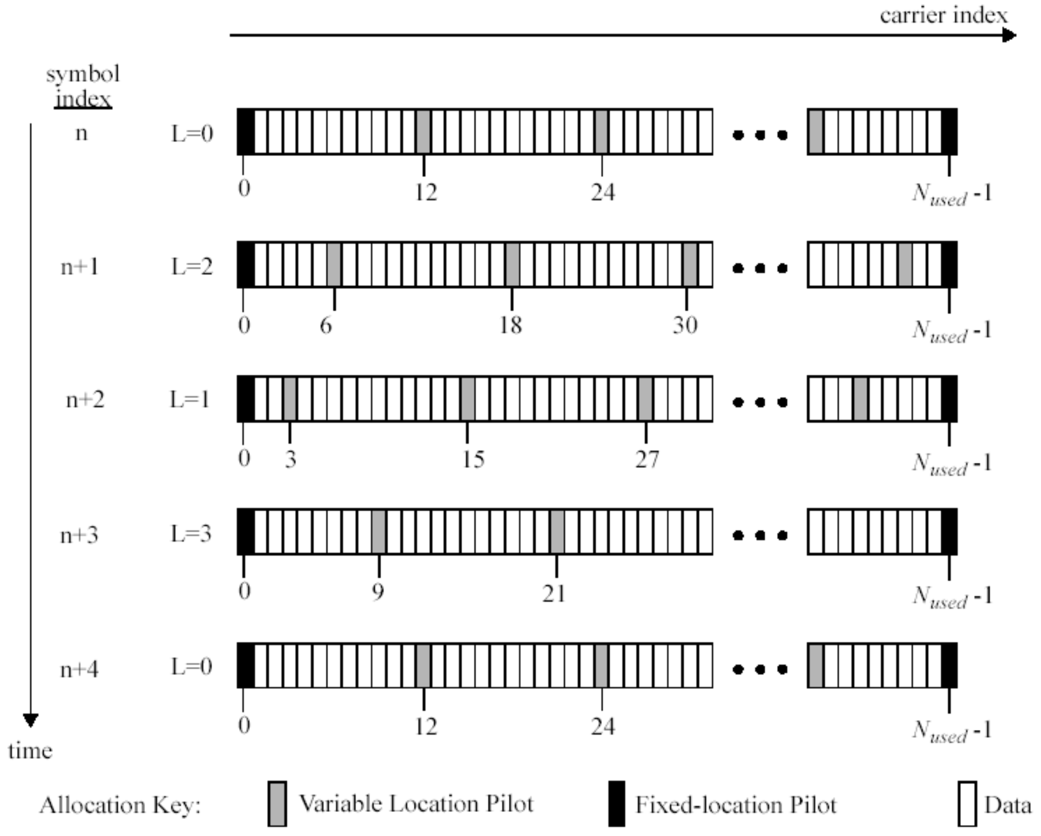


Figure 2.3: Pilot allocation in the OFDMA DL (from [2]).

2.1.4.2 Data Carrier Allocation

After inserting the pilots, the remaining space is for the useful carriers from the data subchannels. To allocate data subchannels, we partition the remaining carriers into groups of contiguous carriers. Each subchannel consists of one carrier from each of these groups respectively. The number of carriers in a subchannel is therefore equal to the number of groups, and it is denoted $N_{subcarriers}$. The number of carrier groups is equal to the number of channels, and it is denoted $N_{subchannels}$. The total number of data carriers is thus equal to $N_{subcarriers} \times N_{subchannels}$.

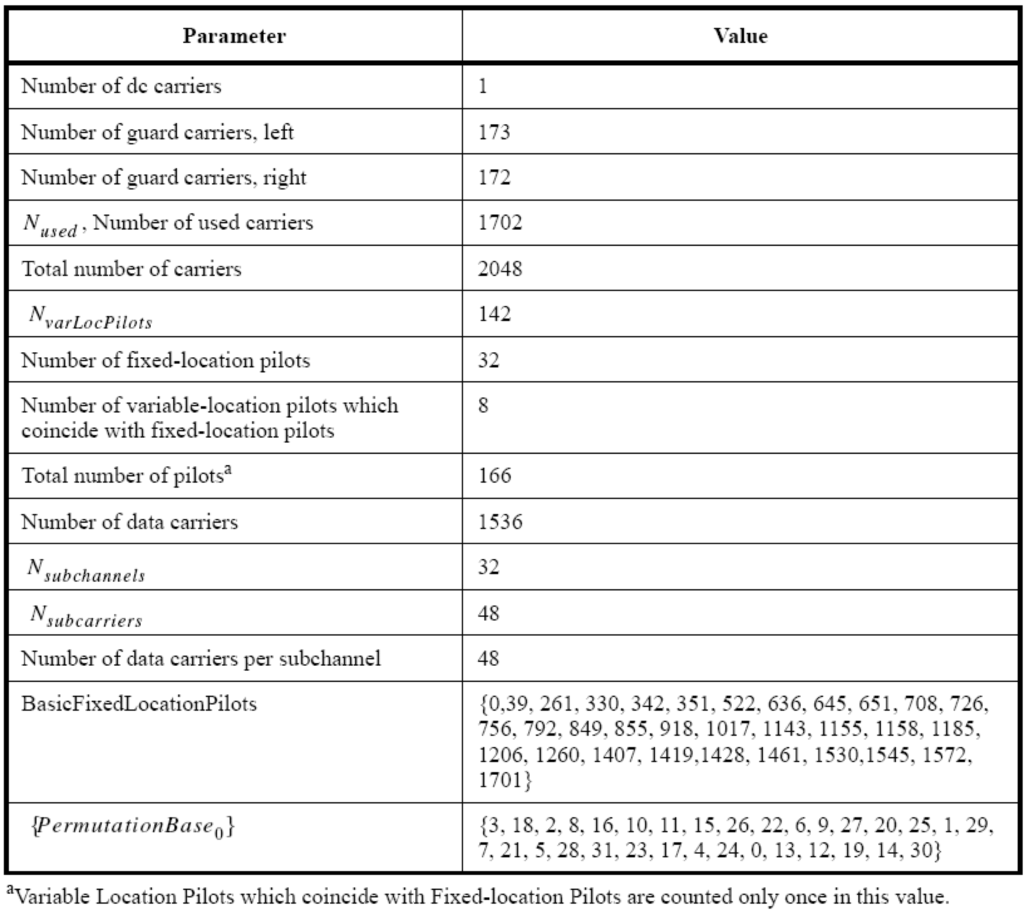
The exact partitioning into subchannels is according to the following equation called a permutation formula:

$$\begin{aligned} carrier(n,s) &= (N_{subchannels}) \cdot n + \{p_s[n_{mod(N_{subchannels})}] \\ &\quad + ID_{cell} \cdot ceil[(n+1)/N_{subchannels}]\}_{(mod(N_{subchannels}))} \end{aligned} \tag{2.1}$$

where:

- $carrier(n,s)$ = carrier index of carrier n in subchannel s.
- s = index number of a subchannel, from the set $[0, \cdots, N_{subchannels}-1]$.
- n = carrier-in-subchannel index from the set $[0, \cdots, N_{subcarriers}-1]$.
- $N_{subchannels}$ = number of subchannels.
- $p_s[j]$ = the series obtained by rotating $\{PermutationBase_0\}$, which is given in the Table 2.1, cyclically to the left s times.
- $ceil[\,]$ = ceiling function which rounds its argument up to the next integer.
- ID_{cell} = a positive integer assigned by the MAC to identify this particular base-station cell.
- $X_{mod(k)}$ = the remainder of the quotient X/k, which is at most $k-1$.

Table 2.1: Carrier Allocation in the OFDMA DL (from [2])



Parameter	Value
Number of dc carriers	1
Number of guard carriers, left	173
Number of guard carriers, right	172
N_{used}, Number of used carriers	1702
Total number of carriers	2048
$N_{varLocPilots}$	142
Number of fixed-location pilots	32
Number of variable-location pilots which coincide with fixed-location pilots	8
Total number of pilots[a]	166
Number of data carriers	1536
$N_{subchannels}$	32
$N_{subcarriers}$	48
Number of data carriers per subchannel	48
BasicFixedLocationPilots	{0,39, 261, 330, 342, 351, 522, 636, 645, 651, 708, 726, 756, 792, 849, 855, 918, 1017, 1143, 1155, 1158, 1185, 1206, 1260, 1407, 1419,1428, 1461, 1530,1545, 1572, 1701}
$\{PermutationBase_0\}$	{3, 18, 2, 8, 16, 10, 11, 15, 26, 22, 6, 9, 27, 20, 25, 1, 29, 7, 21, 5, 28, 31, 23, 17, 4, 24, 0, 13, 12, 19, 14, 30}

[a]Variable Location Pilots which coincide with Fixed-location Pilots are counted only once in this value.

The numerical parameters are given in Table 2.1.

2.1.5 Data Modulation and Pilot Modulation

2.1.5.1 Data Modulation

The data modulation schemes in 802.16a are shown in Fig. 2.4. The data bits are entered serially to the constellation mapper. Gray-mapped QPSK and 16-QAM must be supported, whereas the support of 64-QAM is optional.

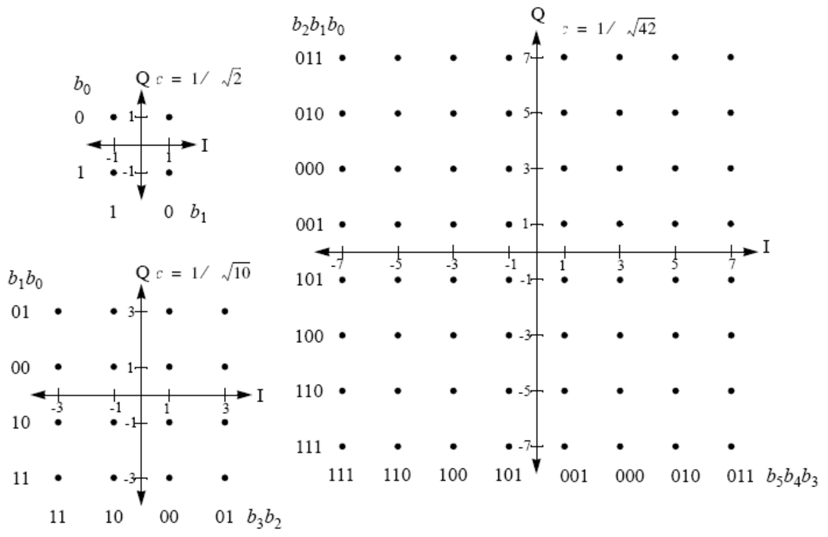


Figure 2.4: QPSK, 16-QAM and 64-QAM constellations (from [2]).

2.1.5.2 Pilot Modulation

Pilot carriers are inserted into each data burst in order to constitute the symbol and they are modulated according to their carrier locations within the OFDMA symbol. A PRBS (pseudo-random binary sequence) generator is used to produce a sequence w_k where k corresponds to the carrier index. The value of the pilot modulation on carrier k is then derived from w_k. The polynomial for the PRBS generator is $X^{11} + X^9 + 1$, as shown in Fig. 2.5.

Symbols in the TDD OFDMA system DL transmission can be separated into two different types. The first three symbols are preamble symbols, and other symbols are normal symbols. The initialization vector of the PRBS in the DL normal symbols is [11111111111], while the initialization vector of the PRBS in the DL preamble symbol is [01010101010]. The PRBS shall be initialized so that its first output bit coincides with the first usable carrier. A new value shall be generated by the PRBS on every usable carrier. Each pilot shall be transmitted with a boosting of 2.5 dB

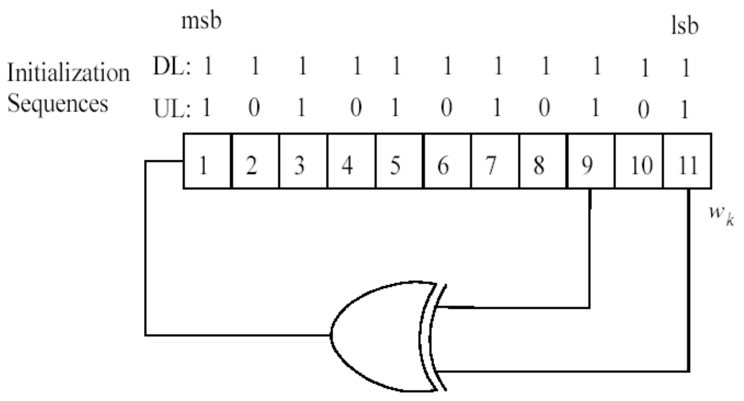


Figure 2.5: Pseudo random binary sequence (PRBS) generator for pilot modulation (from [2]).

over the average power of each data tone. The pilot carriers shall be modulated as

$$Re\{c_k\} = \frac{8}{3}(\frac{1}{2} - w_k), \quad Im\{c_k\} = 0. \tag{2.2}$$

2.2 DL Channel Estimation Methods

Interpolation plays an significant role in pilot-symbol-aided channel estimation. Our interpolation schemes work in both frequency and the time domains. Linear and second-order interpolation are applied in the frequency domain, while 2-D interpolation and LMS (least mean square adaptation) optimize their performance in the time domain.

2.2.1 Pilot-Symbol-Aided Channel Estimation

Channel estimators usually need some kind of pilot information as a point of reference. A fading channel requires constant tracking, so pilot information has to be transmitted more or less continuously. Decision-directed channel estimation can also be used. But even in these types of schemes, pilot information has to be transmitted regularly to mitigate error propagation [4].

In general, the fading channel can be viewed as a two-dimensional (2-D) signal (time and frequency), which is sampled at pilot positions and the channel coefficients between pilots may be estimated by interpolation.

Based on a priori known data, we can estimate the channel information on pilot carriers roughly by the least-square (LS) or the minimum mean square error (MMSE) estimator. An LS estimator minimizes the following squared error [5]:

$$||\mathbf{Y} - \hat{\mathbf{H}}_{LS}\mathbf{X}||^2 \tag{2.3}$$

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 OFDM FFT size. $\hat{\mathbf{H}}_{LS}$ is an $N \times N$ matrix whose values are 0 except at pilot locations m_i where $i = 0, \cdots, N_p - 1$:

$$\hat{\mathbf{H}}_{LS} = \begin{bmatrix} H_{m_0,m_0} & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & H_{m_1,m_1} & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & H_{m_2,m_2} & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & H_{m_{N_p-1},H_{m_{N_p-1}}} \end{bmatrix}. \tag{2.4}$$

Therefore, (2.3) can be rewritten as

$$[Y(m) - \hat{H}_{LS}(m)X(m)]^2, \text{ for all } m = m_i. \tag{2.5}$$

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

$$\hat{H}_{LS}(m) = \frac{Y(m)}{X(m)} = \frac{X(m)H(m) + N(m)}{X(m)} = H(m) + \frac{N(m)}{X(m)} \tag{2.6}$$

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

$$\begin{aligned} \hat{\mathbf{H}}_{p,LS} &= [H_{p,LS}(0)\ H_{p,LS}(1)\ \cdots H_{p,LS}(N_p-1)]^T \\ &= \mathbf{X}_p^{-1}\mathbf{Y}_p \qquad (2.7) \\ &= [\tfrac{Y_p(0)}{X_p(0)}, \tfrac{Y_p(1)}{X_p(1)}, \ldots, \tfrac{Y_p(N_p-1)}{X_p(N_p-1)}]^T, \end{aligned}$$

where $\mathbf{X}_p$ and $\mathbf{Y}_p$ are the collections of the transmitted and the received signal on the pilot subcarriers respectively. The LS estimate of $\mathbf{H}_p$ based on one OFDM symbol only is susceptible to Gaussian noise, and thus an estimator better than the LS estimator is preferable.

The minimum mean-square error (MMSE) estimate has been shown to be better than the LS estimate for channel estimation in OFDM systems, but the major drawback of the MMSE estimate is its high complexity. A low-rank approximation results in a linear minimum mean squared error (LMMSE) estimator that uses the frequency-domain correlation of the channel [6]. The mathematical representation for the LMMSE estimator of pilot signals is

$$\begin{aligned} \hat{\mathbf{H}}_{p,lmmse} &= \mathbf{R}_{H_p H_{p,LS}} \mathbf{R}^{-1}_{H_{p,LS} H_{p,LS}} \hat{\mathbf{H}}_{p,LS} \\ &= \mathbf{R}_{H_p H_p} (\mathbf{R}_{H_p H_p} + \sigma_n^2 (\mathbf{X}_p \mathbf{X}_p^H)^{-1})^{-1} \hat{\mathbf{H}}_{p,LS} \end{aligned} \tag{2.8}$$

where $\hat{\mathbf{H}}_{p,LS}$ is the least-square estimate of $\mathbf{H}_p$ in (2.7), σ_n^2 is the variance of the Gaussian white noise, and the covariance matrices are defined by

$$\mathbf{R}_{H_p H_{p,LS}} = E\{\mathbf{H}_p \mathbf{H}^H_{p,LS}\}, \tag{2.9}$$

$$\mathbf{R}_{H_{p,LS} H_{p,LS}} = E\{\mathbf{H}_{p,LS} \mathbf{H}^H_{p,LS}\}, \tag{2.10}$$

$$\mathbf{R}_{H_p H_p} = E\{\mathbf{H}_p \mathbf{H}_p^H\}. \tag{2.11}$$

Note that there is a matrix inverse involved in the MMSE estimator, which must be calculated every time, and the computation of matrix inversion requires $O(N_p^3)$ arithmetic operations [7]. We also need to use the statistical properties of the unknown channel. Therefore, we use the LS estimator which requires only $O(N_p)$ operations instead of the LMMSE due to the concerns of complexity and unknown information.

2.2.2 Frequency Domain Interpolation Methods

2.2.2.1 Linear Interpolation

Linear interpolation is a commonly used method of interpolation. It does the interpolation simply with two known data, and interpolates those unknown data between them. It is given by [8]

$$H_e(k) = H_e(m+l) = (H_p(m+1) - H_p(m))\frac{l}{L} + H_p(m) \tag{2.12}$$

where $H_p(k), k = 0, 1, \cdots, N_p$, are the channel frequency responses at pilot subcarriers, L is the distance between the two given data, that is, the pilot sub-carriers spacing, and $0 \leq l < L$.

2.2.2.2 Second-Order Interpolation

Theoretically, using higher-order polynomial interpolation may fit the channel response better than linear interpolation [9]. However, the computational complexity grows as the order is increased. Here we consider the second order polynomial interpolation, and it has also been called Gaussian second order estimation. It is given as a solution to the second order polynomial with respect to l/L by using three reference signal points. The interpolation is obtained using three successive pilot subcarriers signal as follows [10]:

$$\begin{aligned} H_e(k) &= H_e(m+l) \\ &= c_1 H_p(m-1) + c_0 H_p(m) + c_{-1} H_p(m+1) \end{aligned} \tag{2.13}$$

where

$$\begin{cases} c_1 = \frac{\alpha(\alpha-1)}{2}, \\ c_0 = -(\alpha-1)(\alpha+1), \\ c_{-1} = \frac{\alpha(\alpha+1)}{2}, \\ \alpha = \frac{l}{L}. \end{cases}$$

The notations are the same as they are in linear interpolation.

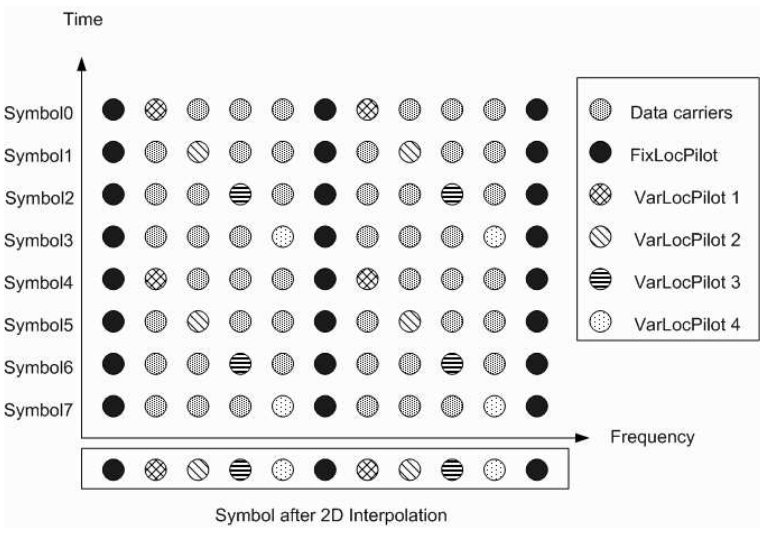


Figure 2.6: Illustration of 2D interpolation.

2.2.3 Time Domain Improvement Methods

As Table 2.1 shows, we can only use 166 pilots in one symbol to interpolate the channel in the frequency domain. It is not sufficient because the pilot spacings are too wide in our system. Since the channel does not change abruptly over time, here we propose two methods to improve the performance.

2.2.3.1 Two-Dimensional Interpolation [11]

Recall the downlink variable pilot allocation in IEEE 802.16a in Fig. 2.3. The equation of the allocation formula is

$$varLocPilot_k = 3L + 12P_k \tag{2.14}$$

where:

- $varLocPilot_k$ = carrier index of a variable-location pilot.
- $L \in 0, \cdots, 3$ is a function of the symbol index, modulo 4.
- $P_k \in \{0, 1, 2, \cdots, N_{varLocPilots} - 1\}$.

Because the positions of the variable location pilots vary with a period of four symbols, we could make use of the four sets of pilot locations to help channel estimation. The maximum number of pilot locations that we can use is

$$(N_{varLocPilots} - N_{CoincidentPilots}) \times 4 + N_{fixLocPilots} = (142 - 8) \times 4 + 32 = 568 \tag{2.15}$$

where $N_{ConincidentPilots}$ is the number of the variable location pilots which are coincident with the fixed location pilots. For example, we can use extrapolation in the time domain to estimate the channel frequency response at the pilot locations of other symbols. It should work the best when transmitting through a static channel. The method is illustrated in Fig. 2.6.

One possible way of interpolation (extrapolation) is

$$\begin{aligned} \tilde{h}^{2D-extrap-p}_{4sets}(f) &= \tfrac{1}{2}\tilde{h}^{p}_{0}(f) + \tfrac{1}{2}\tilde{h}^{p}_{-4}(f) \\ &+ \tfrac{1}{2}\tilde{h}^{p}_{-1}(f) + \tfrac{1}{2}\tilde{h}^{p}_{-5}(f) \\ &+ \tfrac{1}{2}\tilde{h}^{p}_{-2}(f) + \tfrac{1}{2}\tilde{h}^{p}_{-6}(f) \\ &+ \tfrac{1}{2}\tilde{h}^{p}_{-3}(f) + \tfrac{1}{2}\tilde{h}^{p}_{-7}(f) \end{aligned} \tag{2.16}$$

where $\tilde{h}^{p}_{-n}(f)$, $n = 0, 1, \cdots, 7$, are the channel frequency responses at pilot carriers in the nth previous symbol. We can use interpolations again in the frequency domain after obtaining $\tilde{h}^{2D-extrap-p}(f)$. Since the equivalent number of pilots becomes $568/166 = 3.421$ times that of the original case, better estimation is expected.

However, there are seven extra registers needed to store the channel frequency response at pilot carriers. Except for the hardware concern, a fast fading channel might seriously affect the accuracy of the extrapolations in the time domain, because we need to use the information from the seven previous symbols. Thus, an alternative is use less previous symbols, say only 3 or 2. Then the extrapolation formula becomes

$$\begin{aligned}\tilde{h}_{3sets}^{2D-extrap-p}(f) &= \tfrac{1}{2}\tilde{h}_0^p(f) + \tfrac{1}{2}\tilde{h}_{-4}^p(f)\\ &+\tfrac{1}{2}\tilde{h}_{-1}^p(f) + \tfrac{1}{2}\tilde{h}_{-5}^p(f)\\ &+\tfrac{1}{2}\tilde{h}_{-2}^p(f) + \tfrac{1}{2}\tilde{h}_{-6}^p(f)\end{aligned} \tag{2.17}$$

and

$$\begin{aligned}\tilde{h}_{2sets}^{2D-extrap-p}(f) &= \tfrac{1}{2}\tilde{h}_0^p(f) + \tfrac{1}{2}\tilde{h}_{-4}^p(f)\\ &+\tfrac{1}{2}\tilde{h}_{-1}^p(f) + \tfrac{1}{2}\tilde{h}_{-5}^p(f),\end{aligned} \tag{2.18}$$

respectively.

When dealing with fading channels, we consider replacing the formulas above with

$$\begin{aligned}\tilde{h}_{4sets}^{2D-extrap-p}(f) &= \tilde{h}_0^p(f)\\ &+\tfrac{5}{4}\tilde{h}_{-1}^p(f) - \tfrac{1}{4}\tilde{h}_{-5}^p(f)\\ &+\tfrac{3}{2}\tilde{h}_{-2}^p(f) - \tfrac{1}{2}\tilde{h}_{-6}^p(f)\\ &+\tfrac{7}{4}\tilde{h}_{-3}^p(f) - \tfrac{3}{4}\tilde{h}_{-7}^p(f),\end{aligned} \tag{2.19}$$

$$\begin{aligned}\tilde{h}_{3sets}^{2D-extrap-p}(f) &= \tilde{h}_0^p(f)\\ &+\tfrac{5}{4}\tilde{h}_{-1}^p(f) - \tfrac{1}{4}\tilde{h}_{-5}^p(f)\\ &+\tfrac{3}{2}\tilde{h}_{-2}^p(f) - \tfrac{1}{2}\tilde{h}_{-6}^p(f),\end{aligned} \tag{2.20}$$

and

$$\begin{aligned}\tilde{h}_{2sets}^{2D-extrap-p}(f) &= \tilde{h}_0^p(f)\\ &+\tfrac{5}{4}\tilde{h}_{-1}^p(f) - \tfrac{1}{4}\tilde{h}_{-5}^p(f),\end{aligned} \tag{2.21}$$

where we emphasize the weighting of $\tilde{h}_n^p(f)$ $n = -1, -2, -3$ nearier to $\tilde{h}_0^p(f)$ in a linear fashion, because when time variation of the channel is not overly fast, the channel coefficients can be modelled to a first-order approximation as varying linearly with time in a short-enough time span.

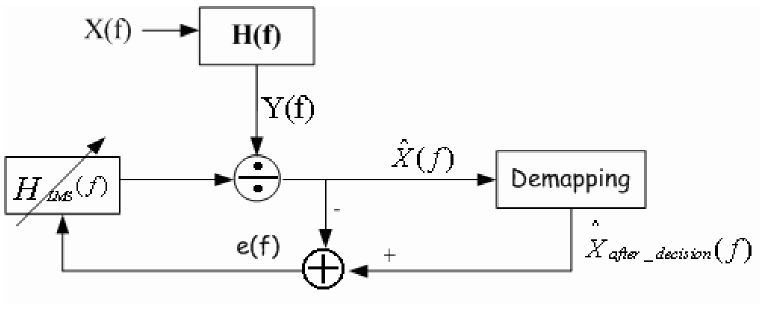


Figure 2.7: Adaptive channel estimation using the LMS algorithm

2.2.3.2 Least Mean Square (LMS) Adaptation [12], [14]

The LMS algorithm is the most widely used adaptive filtering algorithm in practice for its simplicity. Meanwhile, it is stable and robust against different channel conditions.

The LMS channel estimation process is illustrated in Fig. 2.7, where $X(f)$ is the input signal sent into the channel, $H(f)$ is channel frequency response, and $Y(f)$ is the channel output. The following equations apply to our work where $\mathbf{H}^n_{LMS}(f)$ is the estimated channel response at the nth symbol.

- Filtering by channel:

$$y(n) = h(n) * x(n), \tag{2.22}$$

$$\mathbf{Y}(f) = \mathbf{H}(f) \cdot \mathbf{X}(f). \tag{2.23}$$

- Estimated error:

$$\mathbf{e}(f) = \hat{\mathbf{X}}_{after\_decision}(f) - \hat{\mathbf{X}}(f), \tag{2.24}$$

$$\hat{\mathbf{X}}(f) = \frac{\mathbf{Y}(f)}{\mathbf{H}^n_{LMS}(f)}. \tag{2.25}$$

- Cost function:

$$\hat{\xi}(f) = \mathbf{e}^2(f) = |\hat{\mathbf{X}}_{after\_decision}(f) - \hat{\mathbf{X}}(f)|^2. \tag{2.26}$$

- Channel frequency response adaptation:

$$\mathbf{H}^{n+1}_{LMS}(f) = \mathbf{H}^{n}_{LMS}(f) + \mu \mathbf{e}^{*}(f)\hat{\mathbf{X}}(f), \tag{2.27}$$

  where μ is the step size which affects the speed of convergence. With a larger step size, the estimated channel converges more quickly to the real channel response. However, if it is too big, then it may lead to a unstable condition.

To minimize the error shown in (2.24), we try to minimize the expected value of (2.26). For this, we can tune the estimated channel weights adaptively. In our simulation, we use the interpolated channel estimation $\tilde{H}^{0}_{interp}(f)$ as $\mathbf{H}^{0}_{LMS}(f)$ and $\mathbf{H}^{n}_{LMS}(f)$ is obtained by (2.27) when $n > 0$. Following the algorithm, only the first symbol's pilot information is used in the whole flow, thus the pilot information in other symbols is wasted. So we try to combine the interpolated channel and the $\mathbf{H}^{n}_{LMS}(n)$ which is the estimated channel by using LMS algorithm when $n > 0$. The combination is given by

$$\tilde{H}^{n}_{modified\ LMS}(f) = \begin{cases} \alpha \cdot \mathbf{H}^{n}_{LMS}(f) + (1-\alpha) \cdot \tilde{H}^{n}_{interp}(f), & n > 0, \\ \tilde{H}^{n}_{interp}(f), & n = 0, \end{cases} \tag{2.28}$$

where $\mathbf{H}^{n}_{LMS}(f)$ is the channel estimated by the LMS adaptation algorithm and $\tilde{H}^{n}_{interp}(f)$ is the channel estimated by interpolation. The α and the $(1-\alpha)$ are the weighting factors for $\mathbf{H}^{n}_{LMS}(f)$ and $\tilde{H}^{n}_{interp}(f)$, respectively. Therefore, $\tilde{H}^{n}_{modified\ LMS}(f)$ is the combination of these two kinds of estimation outcomes and may be more correct. Then, we use $\tilde{H}^{n}_{modified\ LMS}(f)$ in place of $\mathbf{H}^{n}_{LMS}(f)$ in the right-hand side of (2.27) to calculate the estimated channel response for the next symbol.

Chapter 3

DSP Introduction

DSP implementation is the final goal of our work. The DSP platform that we use is the Quixote board produced by Innovation Integration. The DSP on the board is TMS320C6416 made by Texas Instruments. In this chapter, we introduce the architectures of the Quixote board and the DSP chip.

3.1 Introduction to TMS320C6416 DSP [16]

3.1.1 TMS320C6416 Features

The TMS320C64x DSPs are the highest-performance fixed-point DSP generation of the TMS320C6000 DSP devices with a performance of up to 6000 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. These 8 function units contain:

- Two multipliers.
- Six ALUs.

Features of C6000 devices include :

- Advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic units:
  - Executes up to eight instructions per cycle.
  - 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.
  - Reduces code size, program fetches, and power consumption.
- Conditional execution of all instructions:
  - Reduces costly branching.
  - Increases parallelism for higher sustained performance.
- Efficient code execution on independent functional units:
  - Efficient C compiler on DSP benchmark suite.
  - Assembly optimizer for fast development and improved parallelization.
- 8/16/32-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 Fig. 3.1. The DSP contains:

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

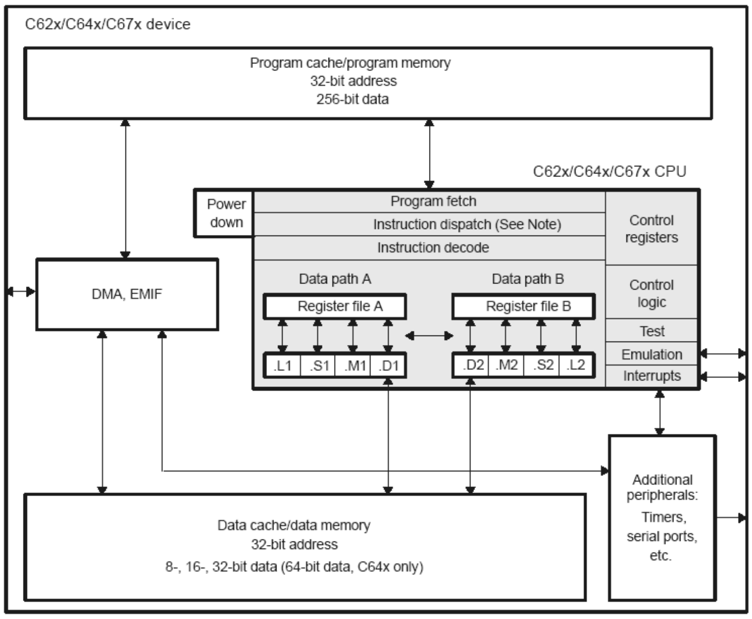


Figure 3.1: Block diagram of the TMS320C6416 DSP [16].

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

The TMS320C64x DSP pipeline provides flexibility to simplify programming and improve performance. The pipeline can dispatch eight parallel instructions every cycle. These two factors provide this flexibility:

- Control of the pipeline is simplified by eliminating pipeline interlocks.
- Increased pipelining eliminates traditional architectural bottlenecks in program fetch, data access, and multiply operations. This provides single cycle throughput.

3.1.2.1 Pipeline

The pipeline phases are divided into three stages as shown in Fig. 3.2:

- Fetch has 4 phases:

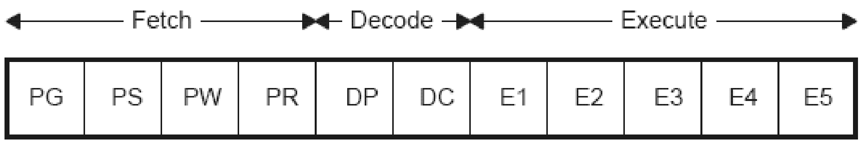


Figure 3.2: Pipeline phases of TMS320C6416 DSP [16].

  - 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.

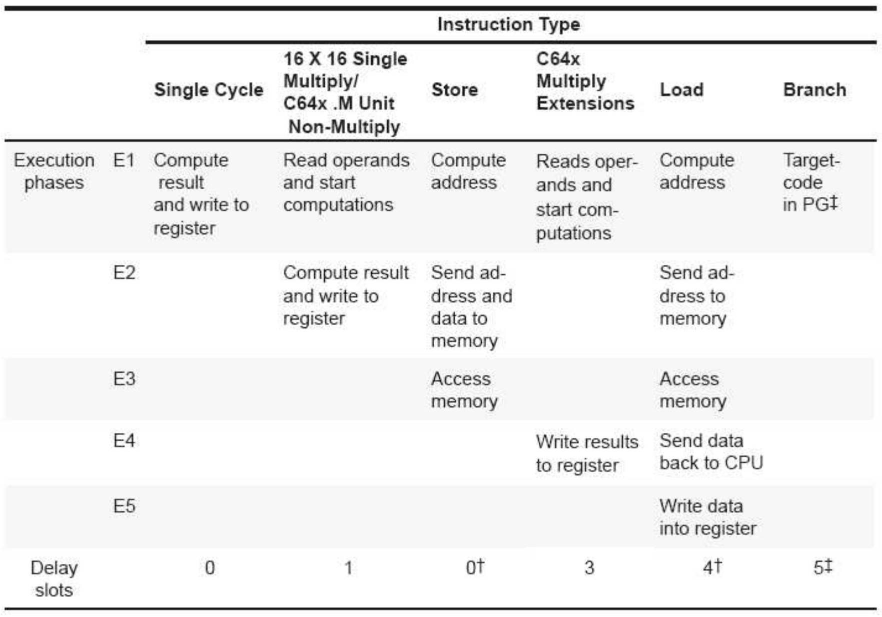
- Decode has two phases:
  - DP (instruction dispatch): The next execute packet in the fetch packet is determined and sent to the appropriate functional units to be decoded.
  - DC (instruction decode): Instructions are decoded in functional units.

- 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 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.

Table 3.1: Execution Stage Length Description for Each Instruction Type [16]



		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‡

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. 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. 3.3). 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.

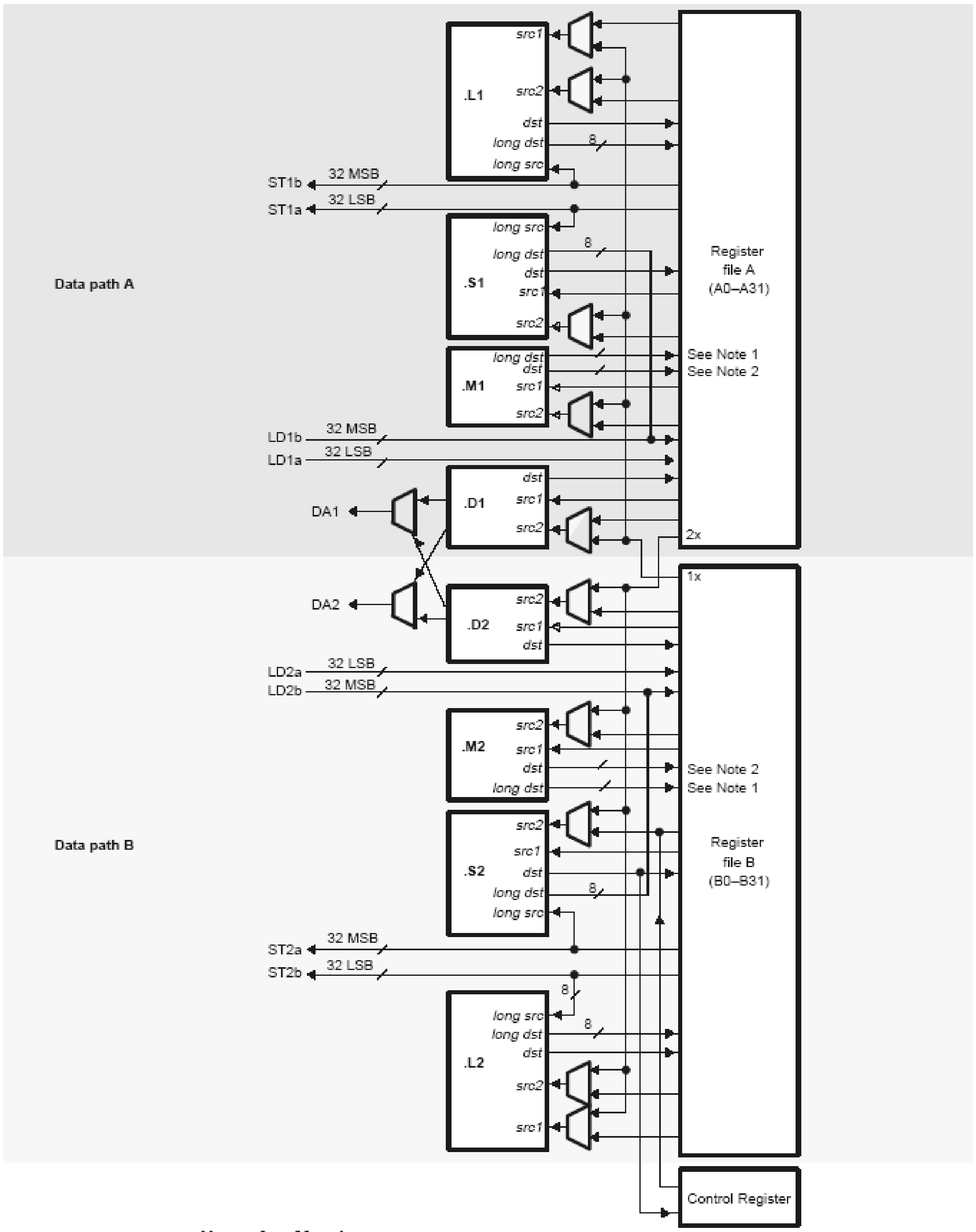
3.1.3 Memory Architecture

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,

Table 3.2: Functional Units and Operations Performed [16]

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



Notes for .M unit:

1. *long dst* is 32 MSB
2. *dst* is 32 LSB

Figure 3.3: TMS320C64x CPU data path [16].

the memory types we can use are:

- On-chip RAM, up to 875 MB.
- Program cache.
- 32-bit external memory interface supports SDRAM, SBSRAM, SRAM, and other asynchronous memories.
- Two-level caches [20]. 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 DSP is a 32 MB SDRAM.

3.2 Introduction to the Quixote cPCI Board [15]

The Quixote is one of Innovative Integration's Velocia-family baseboard for applications requiring speed and processing power. Quixote features a processing core built around Texas Instruments' fixed-point TMS320C6416 and Xilinx Virtex2 with 32 MB of DSP RAM and 2 MB of FPGA computation RAM (optional). The TI C6416 DSP operating at 600 MHz offers a processing power of 4800 MIPS. The analog IO features of the board include dual channels of 105 MHz A/D and D/A (2 in, 2 out). A block diagram of Quixote board is shown in Fig. 3.4.

The Quixote card has a 32 MB SDRAM for use by the DSP. When used with the advanced cache controller on the 'C6416, the SDRAM provides a large, fast external memory pool for DSP data and code. The Quixote has a serial EEPROM for storing data such as board identification, calibration coefficients, and other data that needs

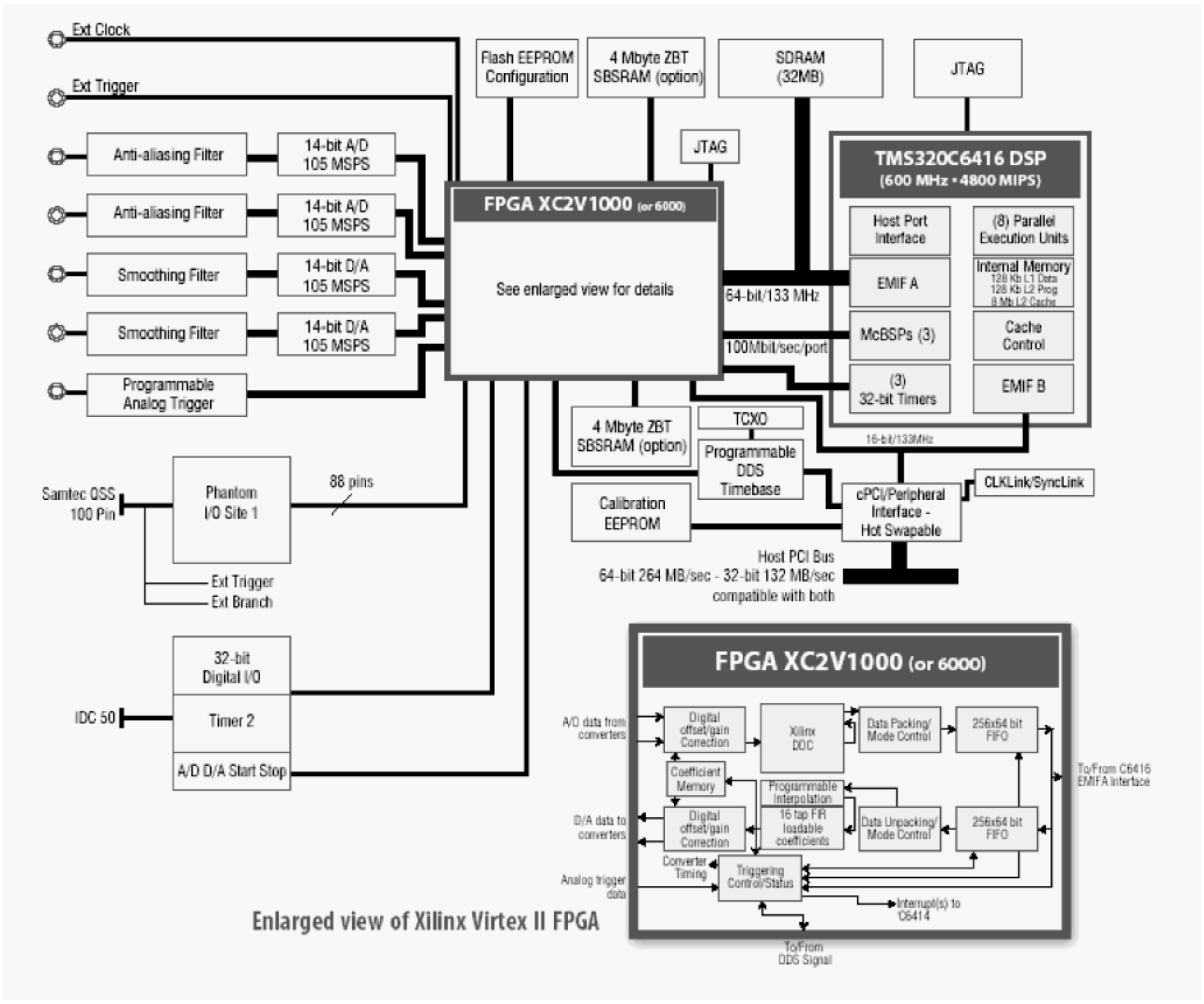


Figure 3.4: Block diagram of Quixote [15].

to be stored permanently on the card. This memory is 16K bits in size. Functions for using the serial EEPROM are included in the Pismo Toolset that allow the software application programmer to easily write and read from the memory without controlling the low-level interface.

The Caliente subsystem handles the details of interacting with the baseboard in streaming mode. There are 3 ways for data transmission between host PC and DSP: data streaming, block mode data streams and message packet I/O.

Data Streaming. To address high-bandwidth data transfer applications, Quixote is capable of continuous transmission and reception of data via the PCI bus, using a mechanism called streaming. When streaming, the target DSP, which must be running a downloaded DSP application, transfers data between target DSP memory and host PC memory automatically with no host intervention. Streaming input is independent of streaming output. It is possible to acquire data from any number and mix of input devices at a programmed rate. Simultaneously, data may be streamed out to a variety of output devices at a different programmed rate. Data flow is fully controlled by use of device drivers called from within the DSP target application.

During data streaming on baseboards, data flows between peripherals and a dedicated, onboard, digital signal processor (DSP) while simultaneously flowing data between the DSP and the host application software. The dedicated DSP can extensively process data as it travels between peripherals and the host application. Fig. 3.5 illustrates the data streaming operation.

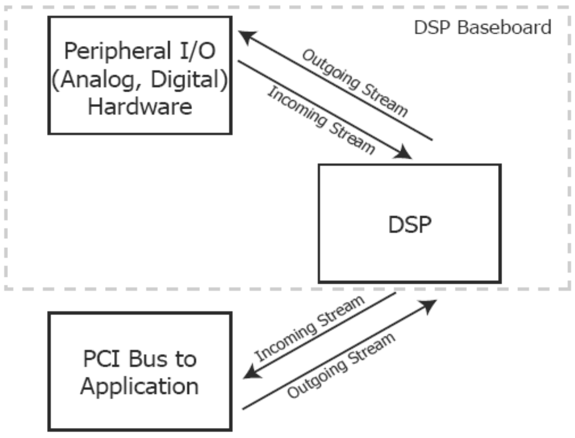


Figure 3.5: Block diagram of DSP streaming mode [15].

Block Mode Data Streams. An alternate data flow paradigm is supported for non-channelized peripherals. This mode is referred to as block mode streaming. In block mode, the splitter/merger features of Caliente are bypassed, and raw, binary data in peripheral-specific format is consumed and supplied by the application program. Devices that produce data that can be channelized may elect to use block mode because of its higher inherent efficiency. For very high rate applications, any processing done to each point may result in a reduction in the maximum data rate that can be achieved. Since block mode does no implicit processing on a point-by-point basis, the fastest data rates are achievable using this mode.

Message Packet I/O. In many applications, there is a need for additional, low bandwidth channels in addition to a high rate data stream. Velocia baseboards feature a means to support the asynchronous interchange of low-bandwidth data in conjunction with high-bandwidth streaming mode I/O. Messages packets consist of a command code and channel number plus up to 14 additional 32-bit parametric data values. Messages may be asynchronously transmitted and received from any number of distinct channels by any number of threads running on both the target DSP and the host PC. Message transfers have no deleterious effect on data streaming and consume virtually none of the bandwidth of the DSP, so they may be freely used even in conjunction with full rate data streaming.

In our implementations, we use block mode data streams the most and also use message packet I/O [24]. TheVirtex2 FPGA includes 18×18 hardware multipliers and contains up to 12 digital clock managers, each providing 256 subdivisions of phase shifting and frequency synthesis capabilities to deliver flexibility in managing both on-chip and off-chip clock domains and synchronization. On-chip memory blocks in the Virtex-II fabric provide convenient high-speed memory elements for FIFOs, dual-port RAM and local processing memory that are invaluable in efficient logic design.

3.3 Introduction to the Code Composer Studio Development Tools[17], [18]

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 Texas Instruments. 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 [19]. The routines included in the DSP library are organized as follows:

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

3.4 Code Optimization Methods [21]

The recommended code development flow involves utilizing the C6000 code generation tools to aid in optimization rather than forcing the programmer to code by hand in assembly. These advantages allow the compiler to do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation. These features simplify the maintenance of the code, as everything resides in a C framework that is simple to maintain, support, and upgrade.

The recommended code development flow for the C6000 involves the phases described in Fig. 3.6. The tutorial section of the Programmer's Guide [21] 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. The following items list the goal for each phase in the software development flow shown in Fig. 3.6.

- 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 [21] 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.

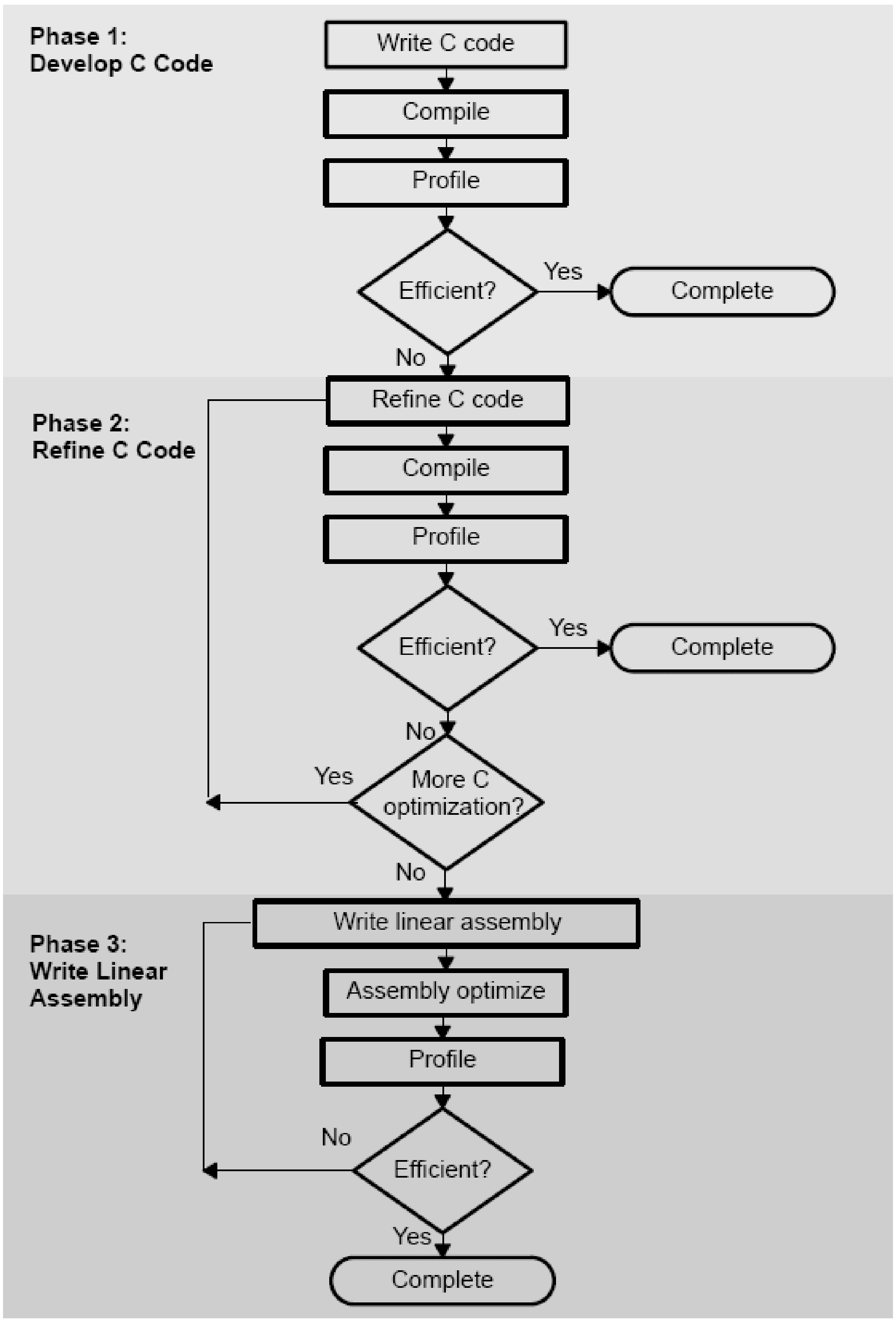


Figure 3.6: Code development flow for TI C6000 DSP [21].

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.

3.4.1 Compiler Optimization Options [17], [18]

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.
  - Inline 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.

3.4.2 Using Intrinsics

The C6000 compiler provides intrinsics, special functions that map directly to C64x instructions, to optimize our 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 [21]. The intrinsics used in our program are introduced in chapter 4.

Chapter 4

Simulation and DSP Implementation

Our work and results can be separated into two parts. The first part concerns the performance of each channel estimation approach, such as symbol error rate (SER), mean square error (MSE), etc. The second part concerns the DSP implementation which emphasizes the execution efficiency.

4.1 Comparison Between 2-D Interpolation and LMS Adaptive Methods

Fig. 4.1 illustrates the block diagram of the simulated system. We assume perfect synchronization and omit it in the simulation. After channel estimation, we get MSE between the real channel response and the estimated one. Also, the SER can be calculated after de-mapping, i.e., de-QAM.

The channel estimation contains several steps:

- Channel response estimation at each pilot location.
- Interpolation for the whole channel response using the estimated values at pilot locations, which may include use of the LMS alogorithm.
- Estimating the transmitted signal using a divider.

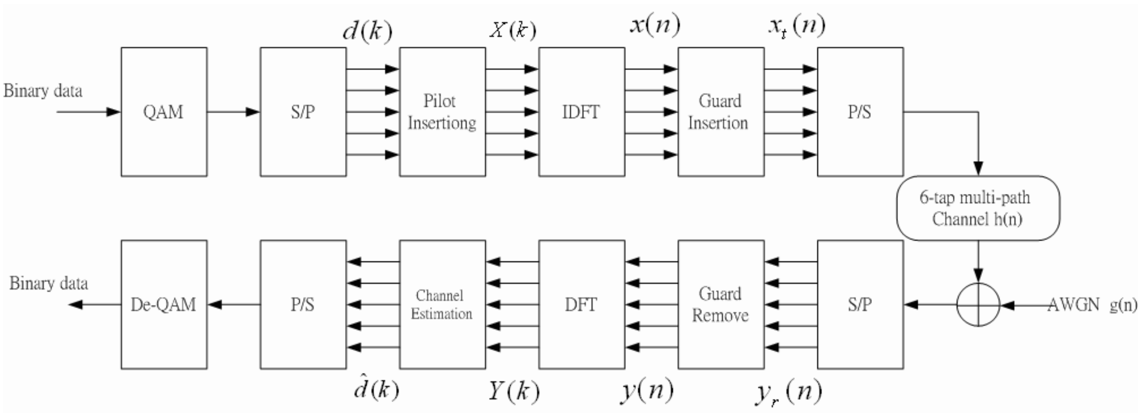


Figure 4.1: Block diagram of the simulated system.

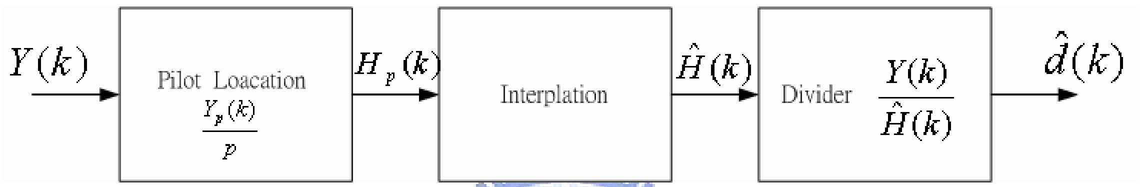


Figure 4.2: Channel estimation steps.

These steps are illustrated in Fig. 4.2.

4.1.1 Simulation Results for AWGN Channel

Before considering multipath channels, we do simulation with an AWGN channel, which means we transmit the data through a one-path channel with $h[0] = 1$, and then add AWGN to it. The theoretical symbol error rate with Gaussian noise power N_0 for M-ary QAM can be obtained by [23]

$$P_e = 4(1 - \frac{1}{\sqrt{M}})Q(\sqrt{\frac{3NE_b}{(M-1)N_0}}) \tag{4.1}$$

where $N = \log_2 M$ and for 64-QAM we have $N = 6$ with $M = 64$ here. The E_b is $E_s/6$ and the E_s is normalized to be 1 in our simulation. If we substitute $E[|\hat{X}_i - X_i|^2]$ for N_0, we can get a theoretical symbol error rate. The result is shown in Figs. 4.3 and 4.4, where we call (2.16) formula 1 and (2.19) formula 2 and linear interpolation is used. The modulation scheme is 64QAM. We can see that the

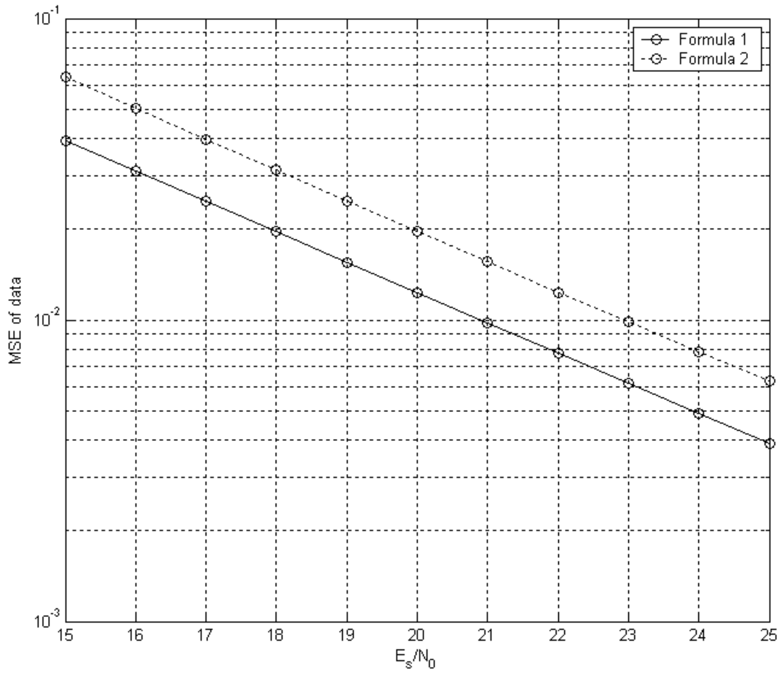


Figure 4.3: MSE of $|\hat{X}_i - X_i|$ for AWGN channel.

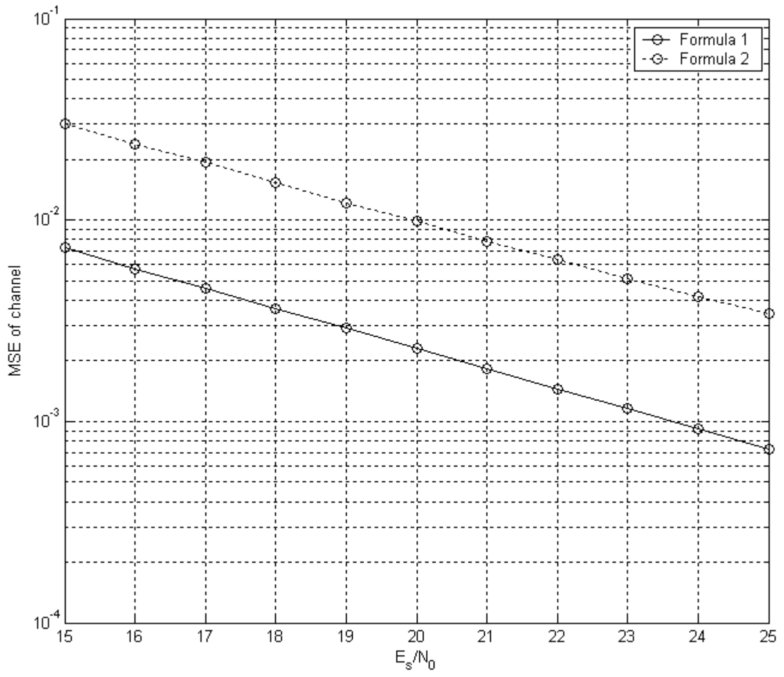
theoretical SERs are closed to the simulated ones whatever the formula we use. We also see that formula 1 works better than formula 2. We calculate the ratio between the coefficients of formula 1 and formula 2 this way:

$$\frac{(\frac{1}{2})^2 \times 8}{1^2 + (\frac{5}{4})^2 + (\frac{1}{4})^2 + (\frac{3}{2})^2 + (\frac{1}{2})^2 + (\frac{7}{4})^2 + (\frac{3}{4})^2} = 0.2286 \qquad (4.2)$$

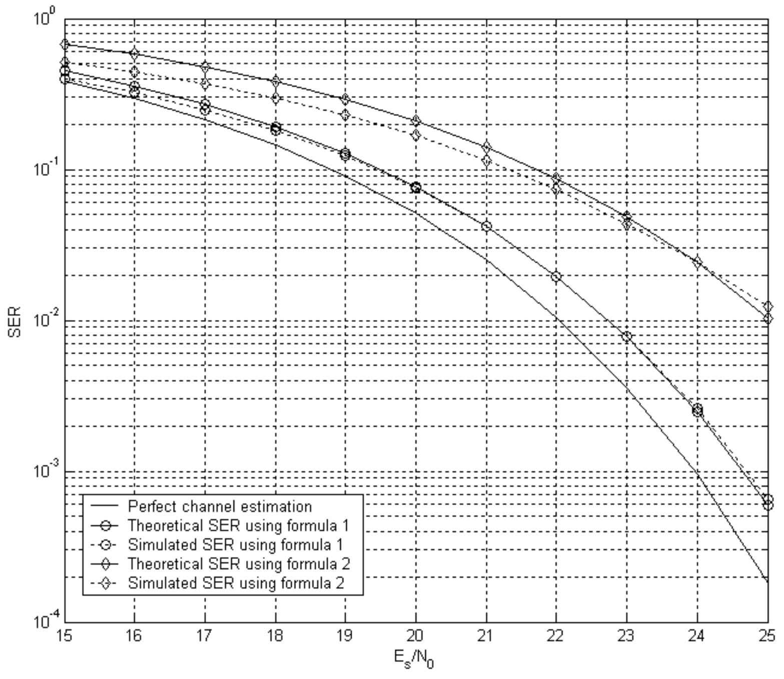
and the simulated ratio is listed in Table 4.1. We can find that those simulated ratios are closed to 0.2286.

4.1.2 Simulation Results for Static Multipath Channel

We employ the ATTC (Advanced Television Technology Center) and the Grande Alliance DTV Laboratory's ensemble E mode channel response, assuming the channel is static. The response is given in Table 4.2. The phase in time domain is $\pi/4$. The amplitude and phase response of this channel response are shown in Fig. 4.5.

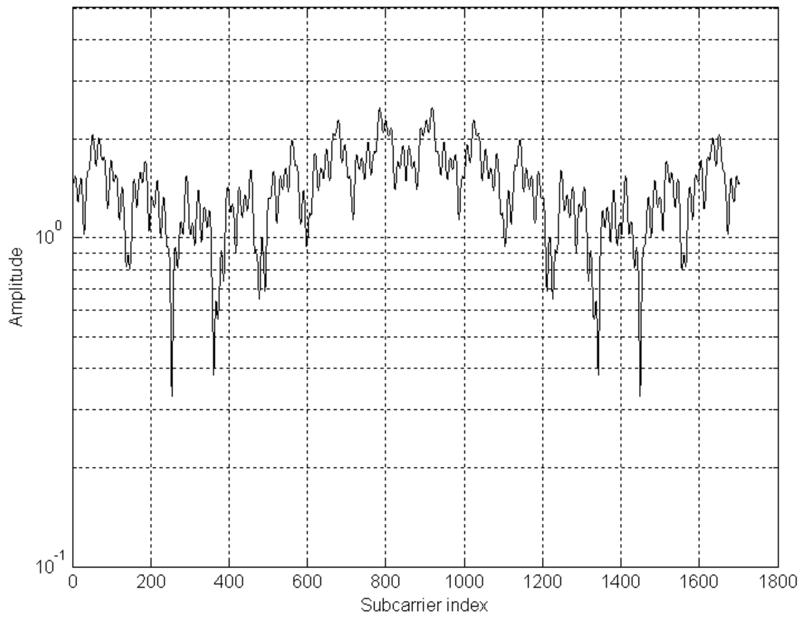


(a)

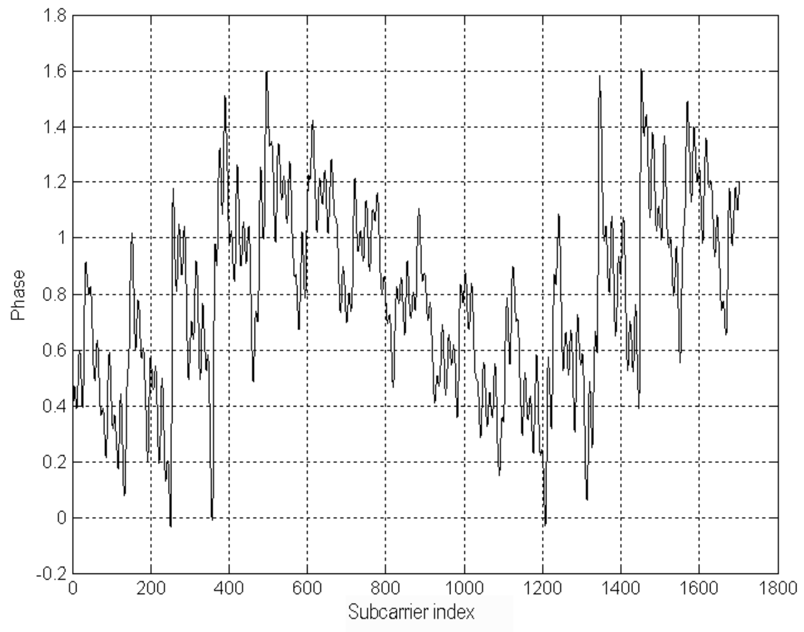


(b)

Figure 4.4: The (a) MSE and (b) SER for AWGN channel simulation



(a)



(b)

Figure 4.5: (a)Amplitude response and (b) phase response of the channel given in Table 4.2.

Table 4.1: MSE Ratio Between Formula 1 and Formula 2 for AWGN Channel

$\frac{E_s}{N_0}$	15	17.5	20	22.5	25	27.5	30	32.5	35	37.5	40
$\frac{MSE_{formula\ 1}}{MSE_{formula\ 2}}$	0.24055	0.23916	0.23756	0.23776	0.23769	0.23417	0.23407	0.22929	0.22764	0.22158	0.21259

Table 4.2: Channel Impulse Response

Tap	**Delay (OFDM Samples)**	**Average Power**	**Average Power (in dB)**
1	0	1	0
2	2	0.3162	-5
3	17	0.1995	-7
4	36	0.1296	-8.87
5	75	0.1	-10
6	137	0.1	-10

4.1.2.1 Two-Dimensional Interpolation

In this section, we will do comparison between the two interpolation schemes proposed in chapter 2. We use different sets in these two formulas, which means different amount of previous symbols' information will be employed.

To verify the correspondence between the simulation results and the theory, we calculate the average $|\hat{X}_i - X_i|^2$ on subcarrier 1 (see Fig.2.7; note that subcarrier indexes run from 0 to 1701) and subcarrier 1700 by simulating 1000 symbols. The theoretical symbol error is taken by following (4.1). Fig. 4.17 shows the MSE of $|\hat{X}_i - X_i|$ on the subcarrier 1 where we use formula 1 and linear interpolation. Fig. 4.7 gives the MSE and SER on the subcarrier 1. The theoretical values are calculated with $N_0 = |\hat{X}_i - X_i|^2$ in (4.1) whether in low SNR or high SNR. We find that the theoretical results are closed to the simulated ones, and we conclude the simulation results seen correct. Figs. 4.9 shows the MSE of $|\hat{X}_i - X_i|$, MSE, SER of the carrier 1700. It responses similar results.

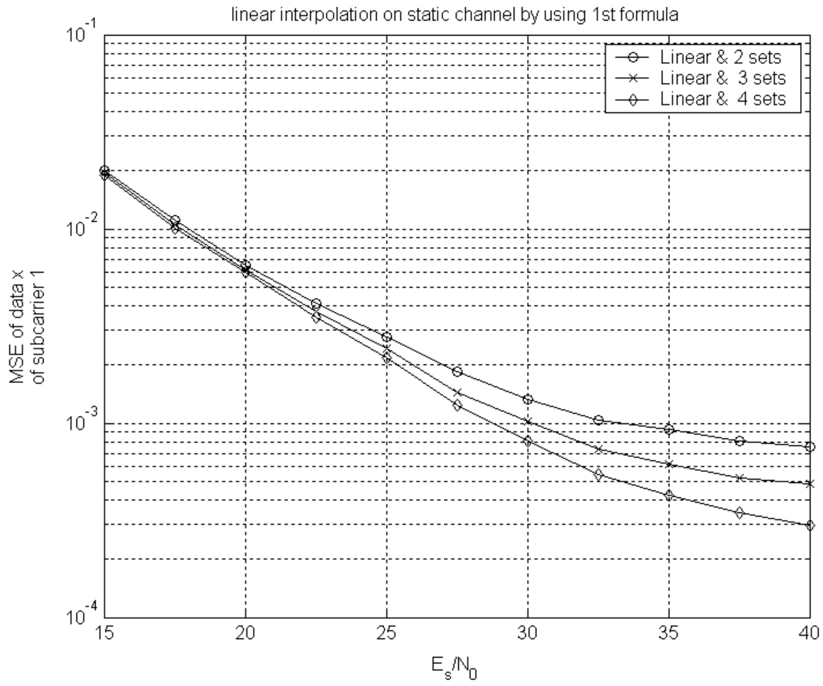
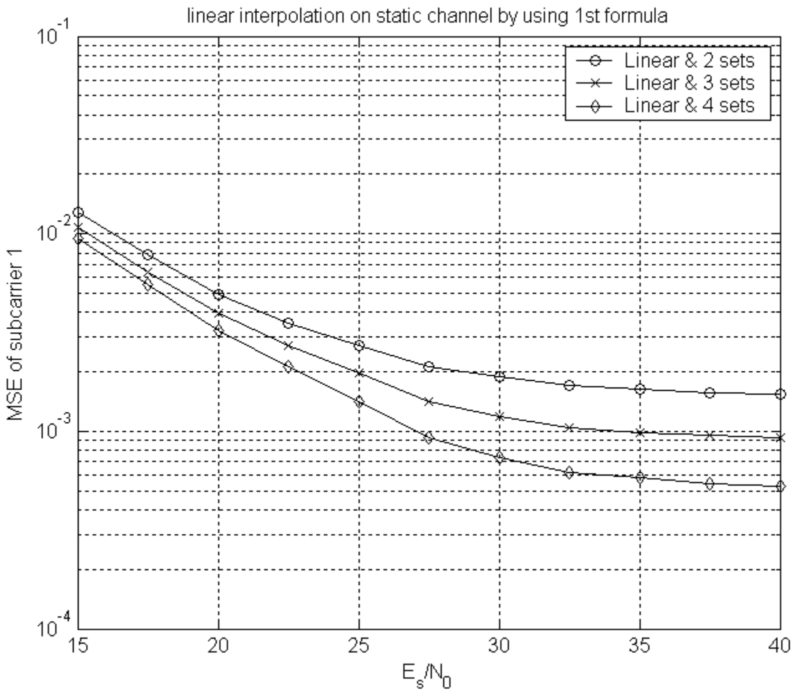


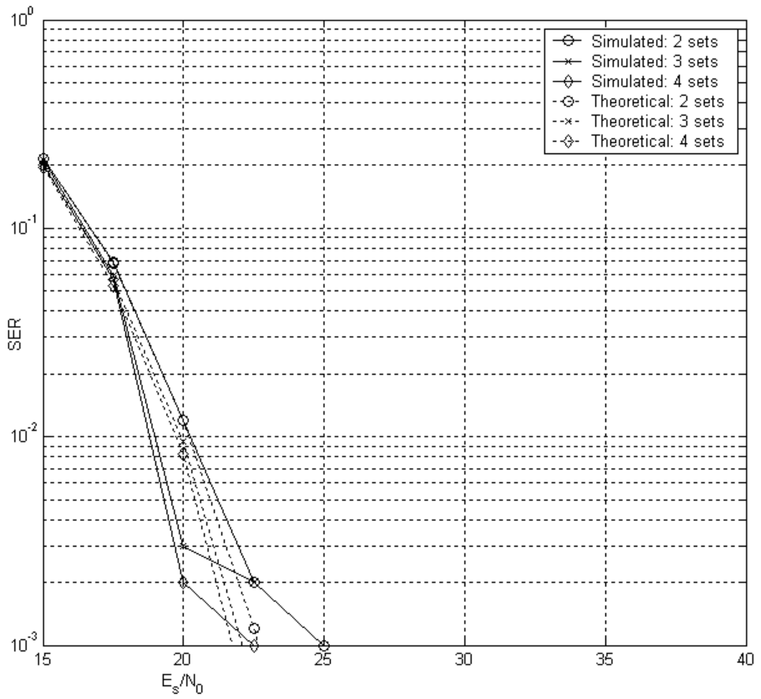
Figure 4.6: MSE of $|\hat{X}_i - X_i|$ on subcarrier 1.

Fig. 4.10 shows the outcomes of formula 1 with both linear and 2nd-order interpolations. Obviously, if we use more sets of pilot information, we get better performance. The MSE and SER of 2nd-order interpolation method decrease faster than the linear one for $E_s/N_0 > 22.5$ dB. The SER of 4 sets interpolation decreases to zero because we have only run 1000 symbols. Thus, it proves that 2-D interpolation is useful in the static channel condition. On the whole, the difference between these two interpolation methods is small but the 2nd-order interpolation is of more complexity than the linear one. Formula 2 yields results with many similar properties, which are given in Fig. 4.11.

We now compare the performance between formula 1 and formula 2. We can find that formula 1 works better than formula 2 in Fig. 4.12; here the linear interpolation is used. This is because we weight the 2 pilot-symbol information equally in formula 1 and it is reasonable doing so in a static channel. In formula 2, we emphasize the



(a)



(b)

Figure 4.7: The (a) MSE and (b) SER on the subcarrier 1 of the 2-D interpolation using formula 1 with linear interpolation in the frequency domain respectively.

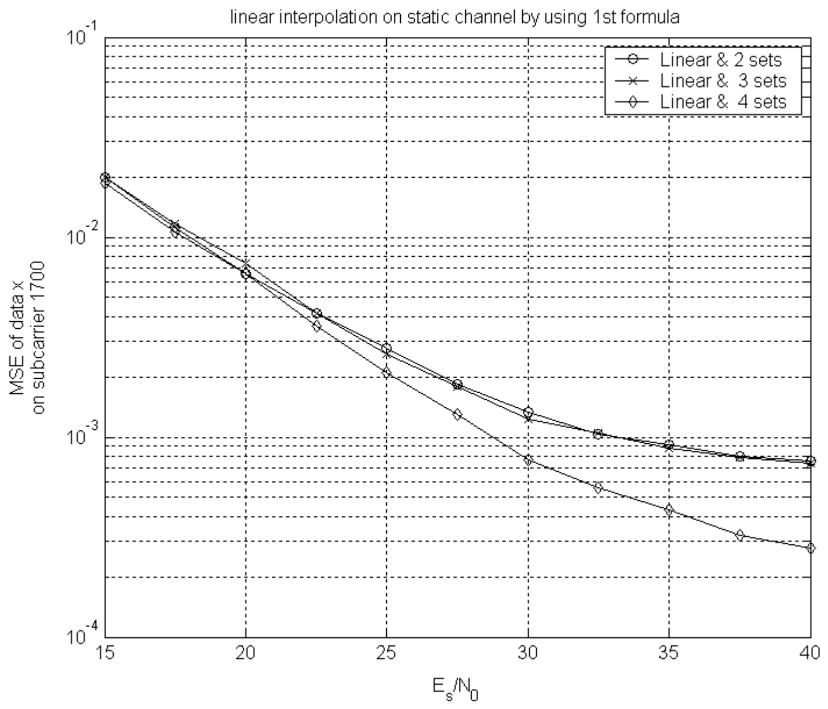
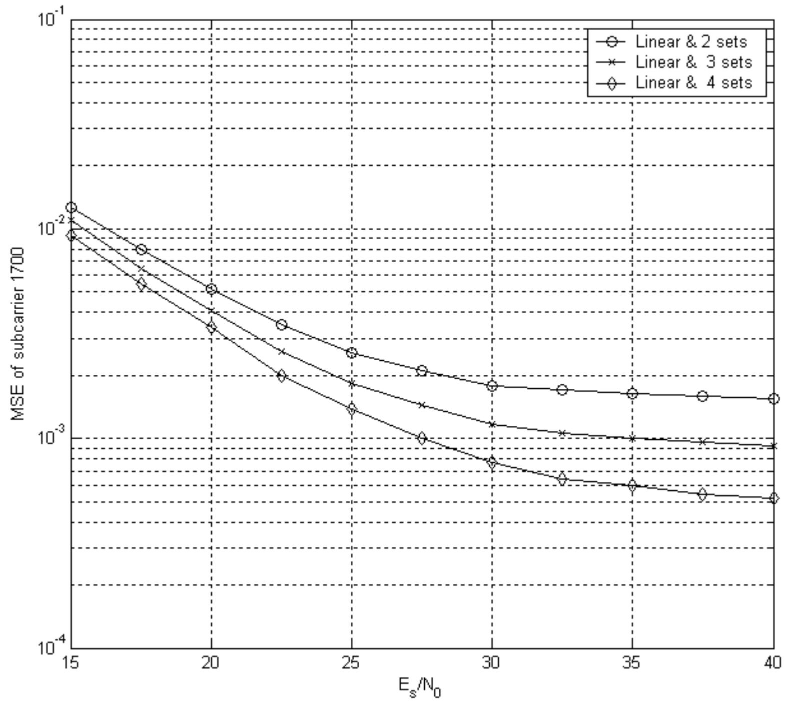
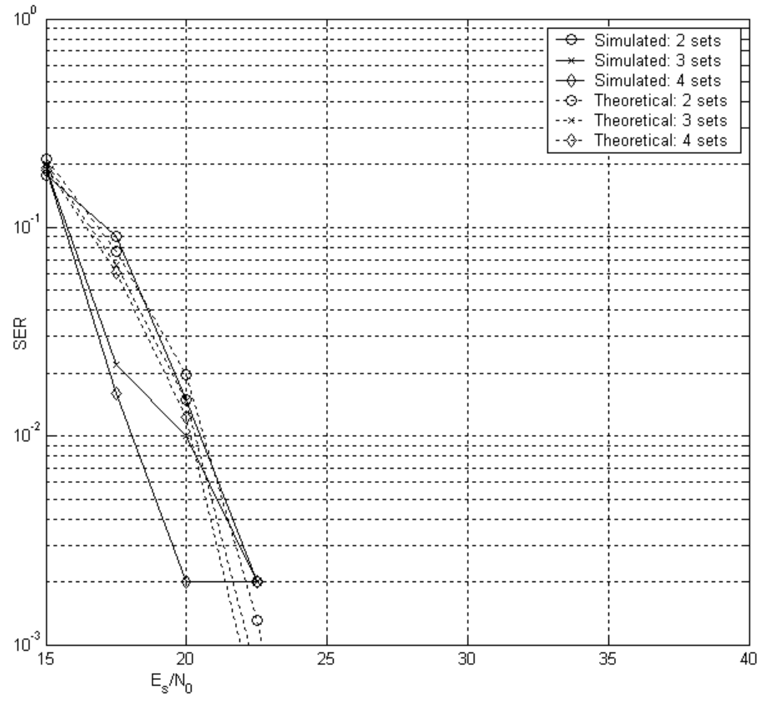


Figure 4.8: MSE of $|\hat{X}_i - X_i|$ on subcarrier 1700.

pilot-symbol information closer to the present symbol. Objectively, it may not be effective in estimating a static channel response because sometimes the information of the symbols which are away from the present symbol may be more correct due to the different AWGN. We also calculate the MSE ratio between formula 1 and formula 2 in Table 4.3 and find the simulated ratios are closed to 0.2286 at low $\frac{E_s}{N_0}$. The same comparison is given in Fig. 4.13 with 2nd-order interpolation. Both these two figure shows that formula 1 causes the SER drops to zero by 2.5 dB earlier than formula 2 with 4 sets of pilot symbols employed. The reason for the zero-dropping is also that we have only run 1000 symbols.

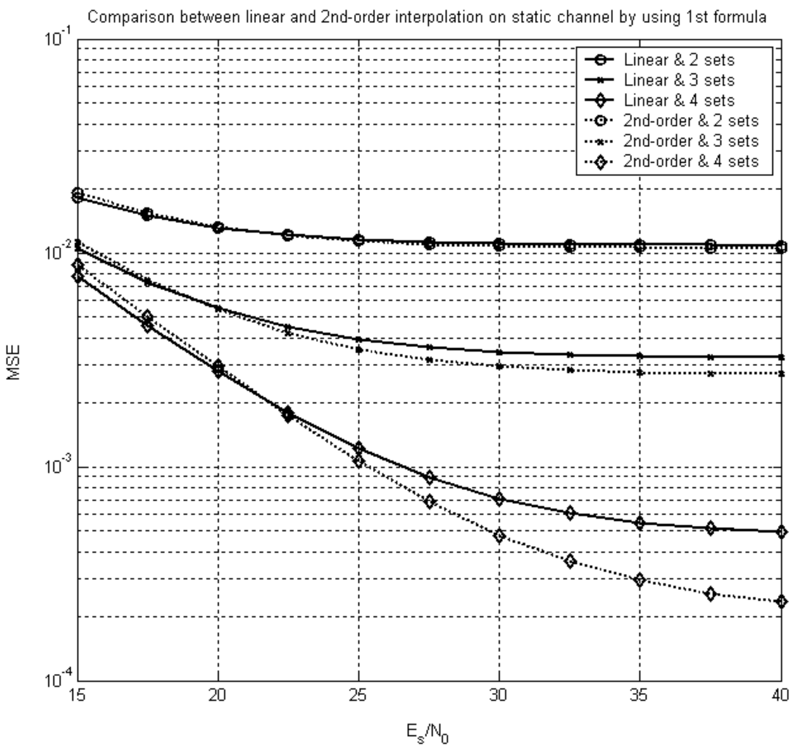


(a)

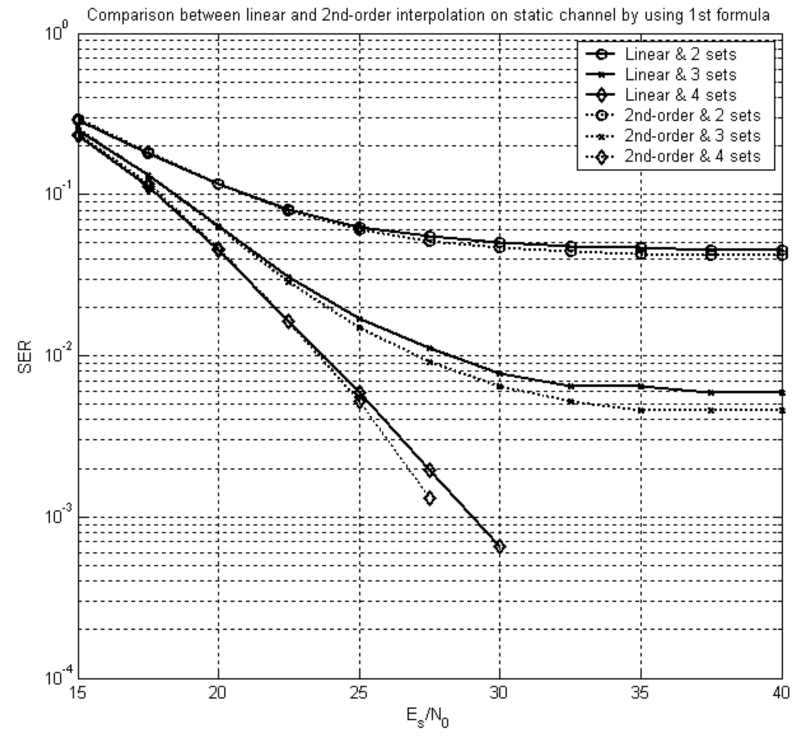


(b)

Figure 4.9: The (a) MSE and (b) SER on the subcarrier 1700 of the 2-D interpolation using formula 1 with linear interpolation in the frequency domain respectively.

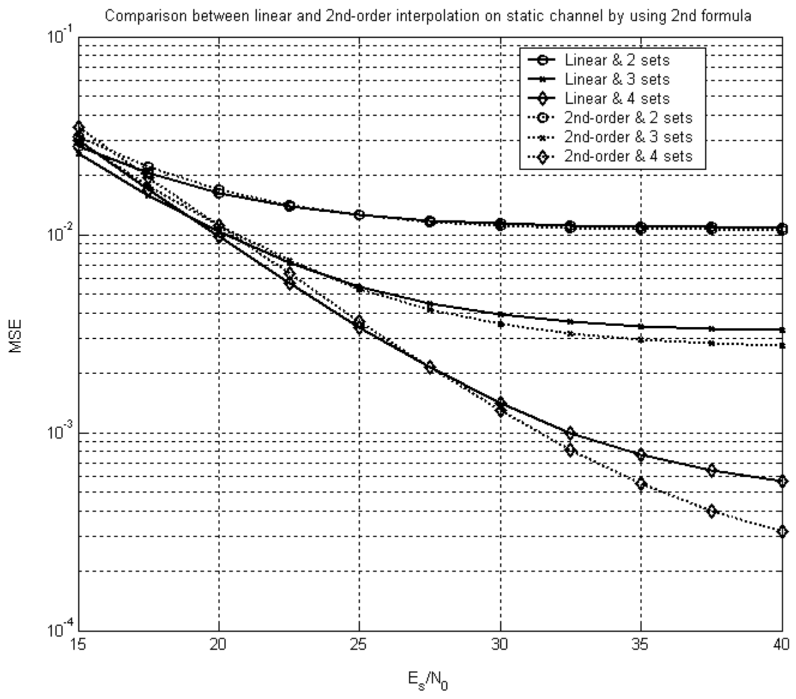


(a)

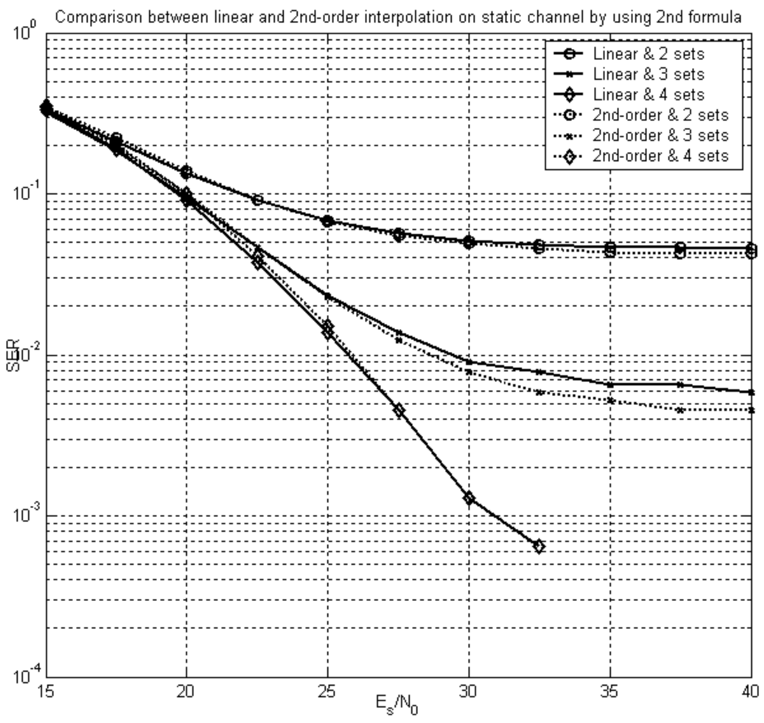


(b)

Figure 4.10: The (a) MSE and (b) SER of the 2-D interpolation using formula 1 with linear and 2nd-order interpolation in the frequency domain respectively.

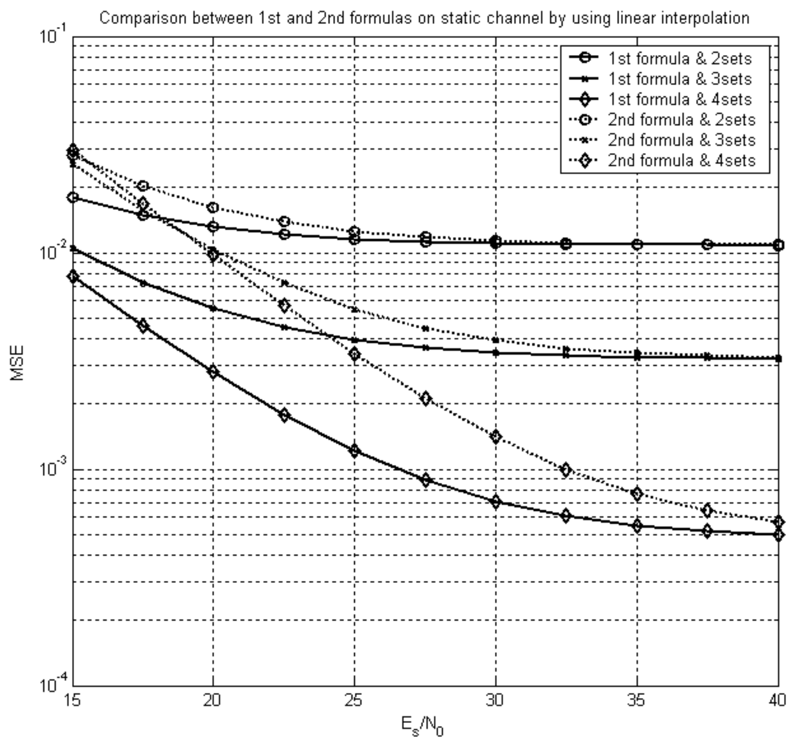


(a)

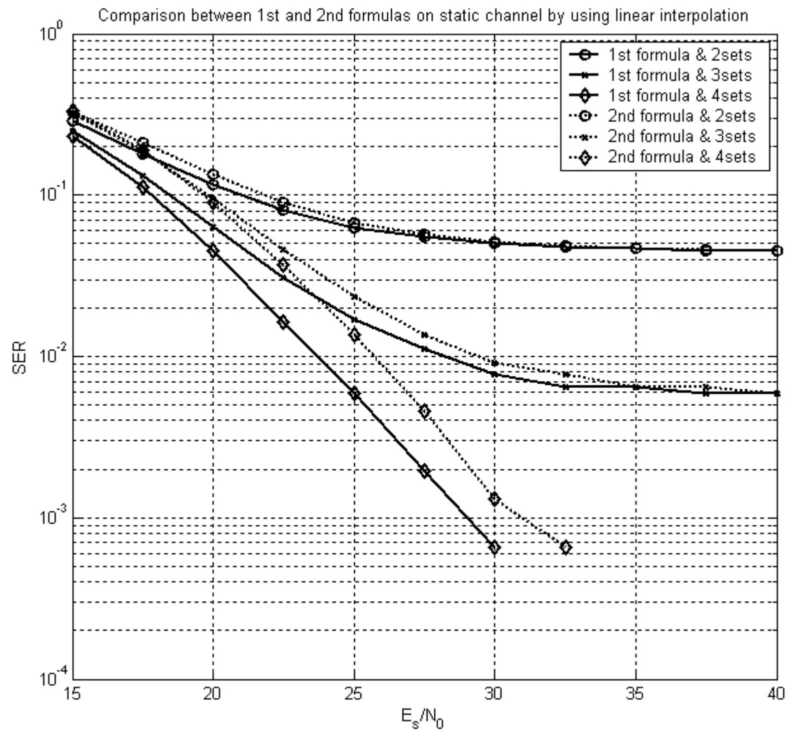


(b)

Figure 4.11: The (a) MSE and (b) SER of the 2-D interpolation using formula 2 with linear and 2nd-order interpolation in the frequency domain respectively.

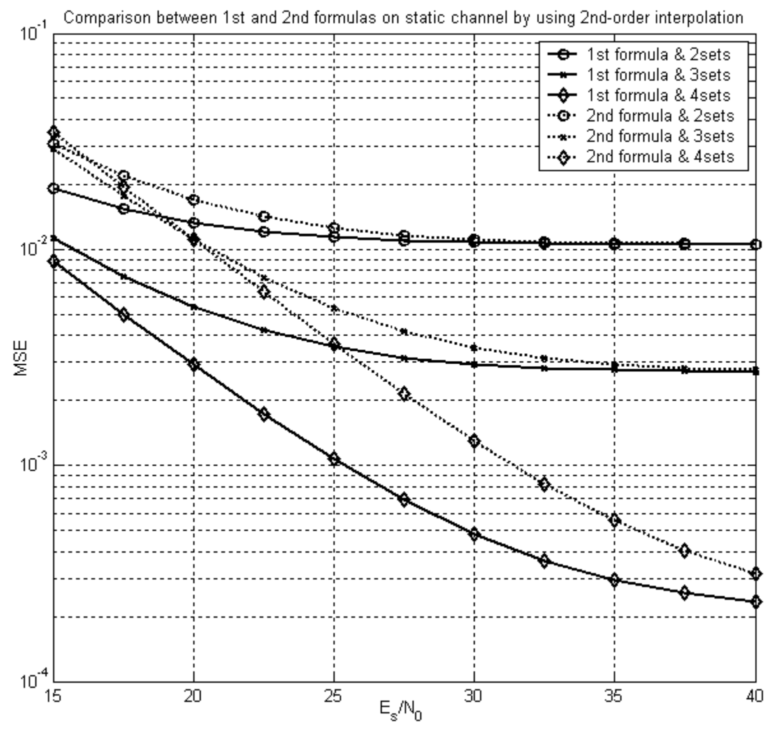


(a)

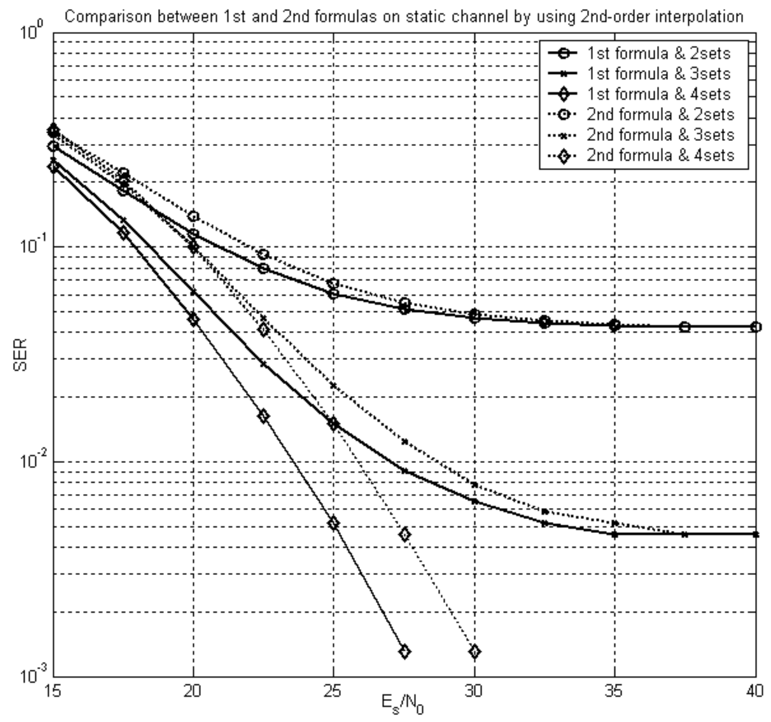


(b)

Figure 4.12: The (a) MSE and (b) SER of using formula 1 and 2 in the 2-D interpolation respectively with linear interpolation in the frequency domain.



(a)



(b)

Figure 4.13: The (a) MSE and (b) SER of using formula 1 and 2 in the 2-D interpolation respectively with 2nd-order interpolation in the frequency domain.

Table 4.3: MSE Ratio Between Formula 1 and Formula 2 for Multipath Channel

$\frac{E_s}{N_0}$	15	17.5	20	22.5	25	27.5	30	32.5	35	37.5	40
$\frac{MSE_{formula\ 1}}{MSE_{formula\ 2}}$	0.26065	0.26941	0.28589	0.3106	0.35563	0.41757	0.50178	0.60782	0.71261	0.80499	0.87676

4.1.2.2 LMS Adaptive Algorithm

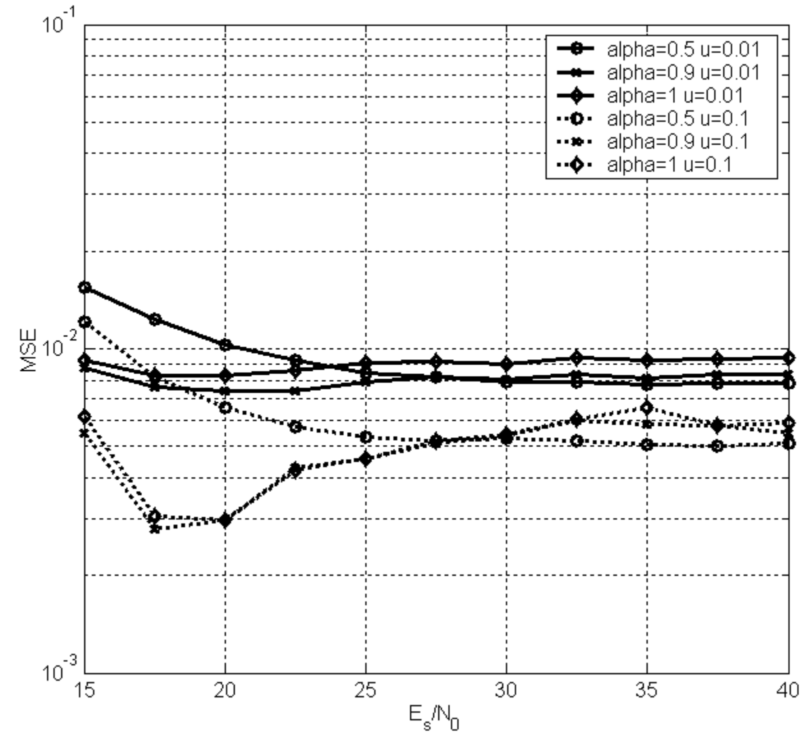
In this section we combine the LMS adaptation with linear interpolation. The step-size parameter μ of the LMS algorithm obeys [12], [13]

$$0 < \mu < \frac{2}{3tr[\mathbf{R}]}, \tag{4.3}$$

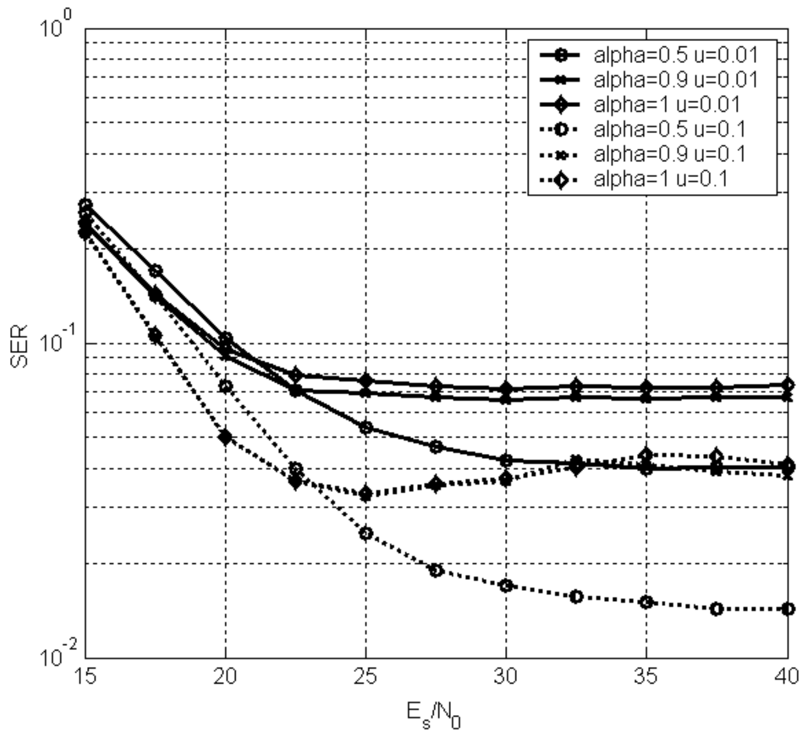
where tr[$\mathbf{R}$] is the sum of the powers of the signal samples at the filter tap inputs, which are the powers of the channel impulse response taps in our case. Since $tr[\mathbf{R}] = 1 + 0.3162 + 0.1995 + 0.1296 + 0.1 + 0.1 = 1.1766$, we choose μ to be 0.1 and 0.01. Recall the adaptive equation $\hat{\mathbf{H}}_{p,LMS}(n+1) = \hat{\mathbf{H}}_{p,LMS}(n) + \mu\varepsilon(n)^*\mathbf{X}(n)$, where $\mathbf{X}$(n) is obtained by dividing the received signal by the estimated channel response i.e., $\mathbf{X}$(n)=$\frac{\mathbf{Y}(n)}{\hat{\mathbf{H}}_{p,LMS}(n)}$. This essentially assumes that our decision is correct. We also compare the different weights α for the LMS filter output and the interpolated data. The results are shown in Figs. 4.14 and 4.15. The MSE in Fig. 4.15 is the error we want to minimize in this algorithm. From these simulation results, we get steady convergence if we set $\mu = 0.1$ and $\alpha = 0.5$. However, when compared with 2-D interpolation, the outcome of LMS adaptive algorithm is not sufficient for our requirements.

4.1.3 Multipath Rayleigh Fading Channel Simulations

We simulate block type Rayleigh fading in our work. The Rayleigh fading is simulated as in [22], which is an improved Jakes' model and the below equations are its



(a)



(b)

Figure 4.14: The (a) MSE and (b) SER for different weighting and different step-size parameters in LMS adaptive method.

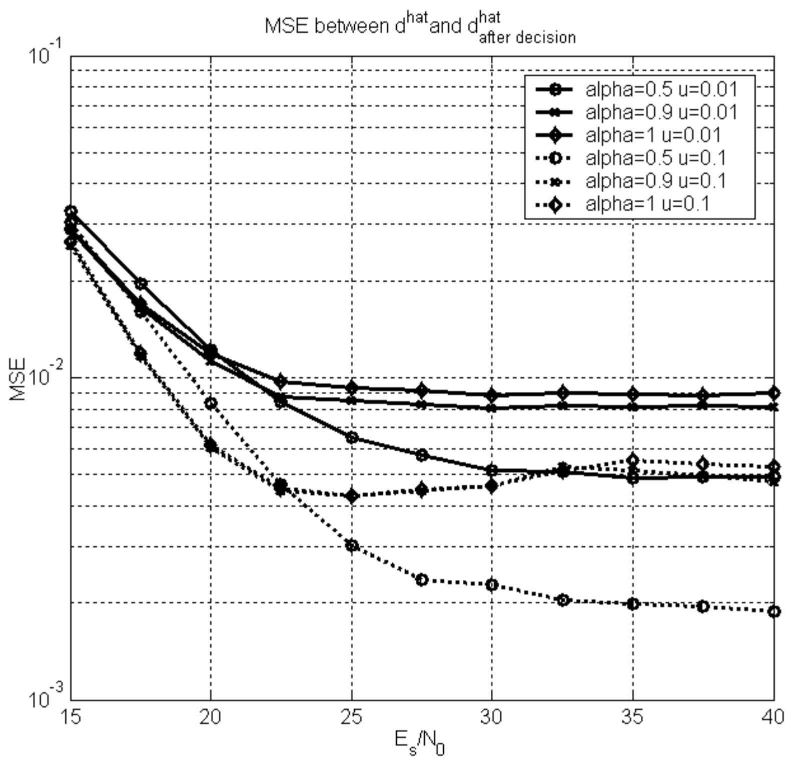


Figure 4.15: MSE between $\hat{X}$ and $\hat{X}_{after\_decision}$ for different weighting and different step-size parameters in LMS adaptive method.

mathematical expressions.

$$R(t) = R_c(t) + jR_s(t), \tag{4.4}$$

$$R_c(t) = \frac{2}{\sqrt{M}} \sum_{n=1}^{M} \cos(\psi_n) \cdot \cos(w_d t \cdot \cos\alpha_n + \phi), \tag{4.5}$$

$$R_s(t) = \frac{2}{\sqrt{M}} \sum_{n=1}^{M} \sin(\psi_n) \cdot \cos(w_d t \cdot \cos\alpha_n + \phi), \tag{4.6}$$

where

$$\alpha_n = \frac{2\pi n - \pi + \theta}{4M}, \qquad n = 1, 2, \cdots, M, \tag{4.7}$$

with θ, ϕ, and ψ_n being statistically independent and uniformly distributed over $[-\pi, \pi)$ for all n, and $M = 8$ in our simulation.

The Rayleigh fading channel is created following the equation:

$$h[n] = \sum_{i=1}^{k} R_i[n] \times \alpha_i \delta[n - d_i]; \quad H(f) = FFT\{h[n]\} \tag{4.8}$$

where k is the total path number, α_i is the static power of the ith path, $R_i[n]$ is the Rayleigh parameter calculated by (4.4), and the d_i is the delay spread of the ith path. Because we simulate channel transmission in the frerquency domain as $Y(f) = X(f) \times H(f) + N(f)$, therefore we have to transform $h[n]$ to the frequency domain. In Rayleigh channel simulation, the $R_i[n]$ varies every symbol, thus we need to do the FFT computation every symbol. To reduce the complexity and the computation time, we simplify (4.8) into:

$$Base_i(f) = FFT\{\alpha_i \delta[n - d_i]\}, \tag{4.9}$$

$$H(f) = \sum_{i=1}^{k} R_i[n] \times Base_i(f). \tag{4.10}$$

Then, we compute FFT only 6 times at the first, and the latter channel frequency response can be obtained by the linear summizing the products of the Rayleigh parameters and the $Base_i(f)$. Compared with (4.8), we have to compute FFT for one time per symbol, but now only 6 times in total because we have $Base_i(f)$,$i = 1 \cdots 6$. It does save a lot of time.

The Doppler shift is given by [23]

$$f_d = \frac{v}{c} \cdot f_c \cdot \cos\theta \tag{4.11}$$

where v is the velocity of vehicles km/hr, c is the velocity of light, and θ is the angle between the direction of v and line-of-sight of transmitter and receiver. The simulation parameters are listed below:

- $f_c = 2$ GHz.
- $T = T_s = 201.9\mu$ s.

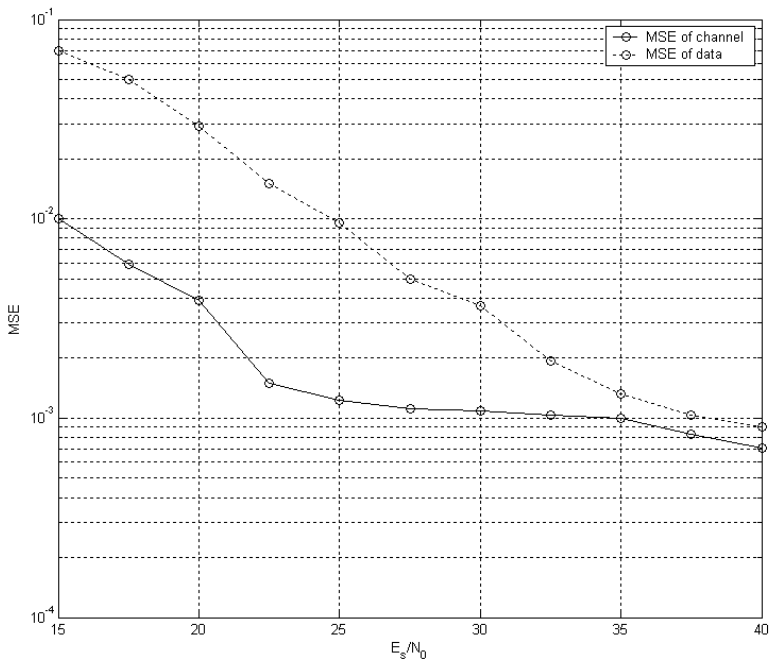
Table 4.4: Relation Between Speed and Maximum Doppler Shift

Speed (km/hr)	Max. Doppler Shift (Hz)	f_dT_s
27	50	0.01
54	100	0.02

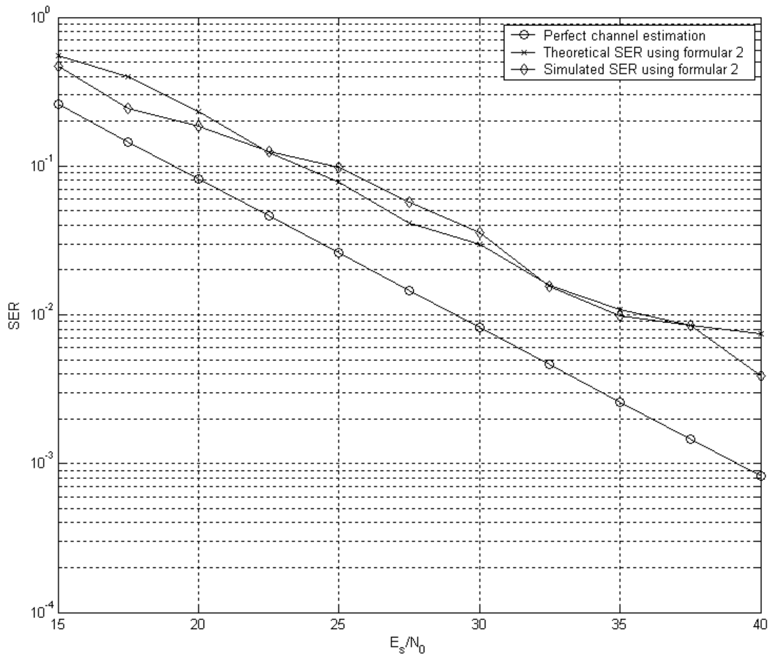
Table 4.4 shows the relation between some simulation parameters. Here we only present the result of using in 2-D interpolation since its performance is much better than that of LMS adaptive algorithm, as seen in the previous discussion. Besides, for a Rayleigh fading channel, it would be useful emphasizing the estimated channel information at pilots in symbols which are closer to the present one. For this reason, we decide to do multipath fading channel estimation with formula 2.

First we do simulation on one-path Rayleigh fading channel with linear interpolation and formula 2. The results are given in Figs. 4.16, where the theoretical curve is obtained using the MSE of $|\hat{X}_i - X_i|$ in (4.1), as we did in the cases of static one-path and multipath channels. The reason why the simulated SER is much different from the theoretical one may be that the MSE of $|\hat{X}_i - X_i|$ cannot be seemed as N_0.

Now we turn to the multipath Rayleigh fading channel. To verify our the correction of oue simulation, we only have comparison between the simulation results and the theory. For subcarrier 1, the MSE and SER are shown in Figs. 4.17 and 4.18. The other verification for subcarrier 1700 are shown in Figs. 4.19 and 4.20. We can see that there is much different between the simulated results and the theoretical ones. The reason for this may be that the $|\hat{X}_i - X_i|^2$ cannot replace the N_0 in the (4.1) as a AWGN in the multipath Rayleigh fading channel. Then, we do other comparison in Figs. 4.21 and 4.22. We can observe that 2nd-order interpolation still performs better than the linear one with $f_dT = 0.01$ as before. But it loses its superiority with $f_dT = 0.02$. Both MSE and SER do not decease smoothly as E_s/N_0 increases, because the 6-tap multipath Rayleigh fading channel is hard to



(a)



(b)

Figure 4.16: The (a) MSE of $|\hat{x}_i - x_i|$, MSE and (b) SER for one-path Rayleigh fading channel, where $V = 27$ km/h, $fdT = 0.01$.

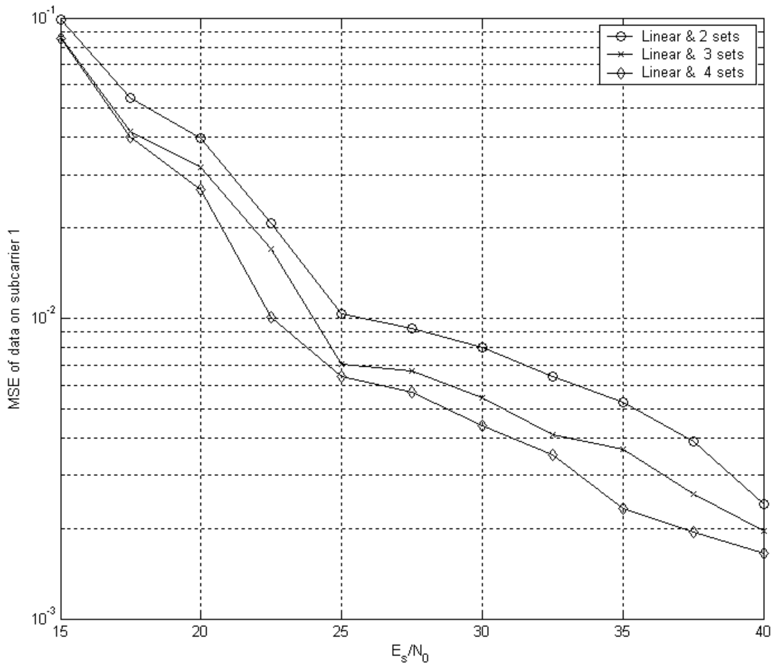
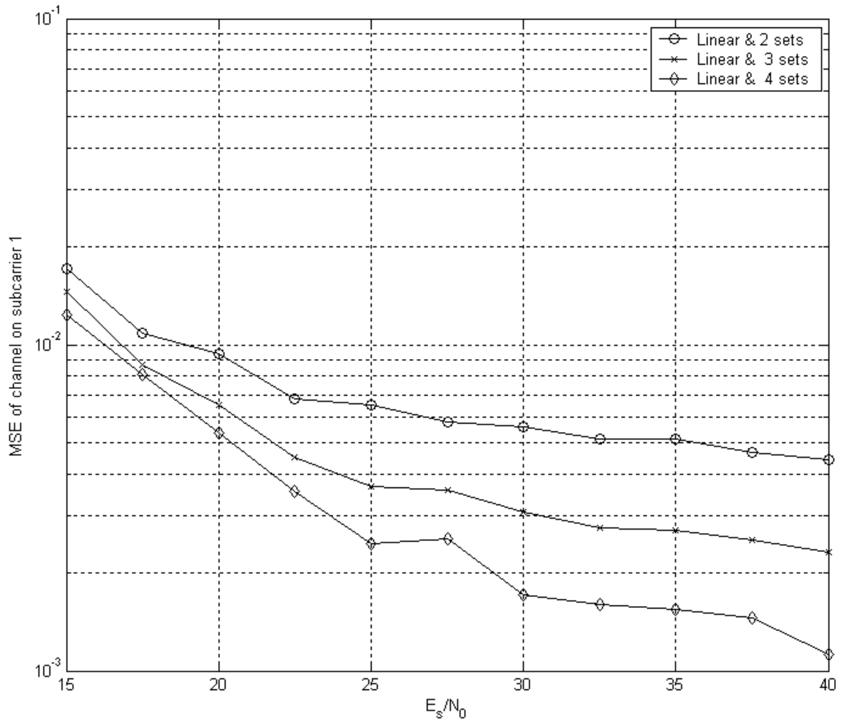


Figure 4.17: MSE of $|\hat{X}_i - X_i|$ on subcarrier 1 for multipath Rayleigh fading channel.

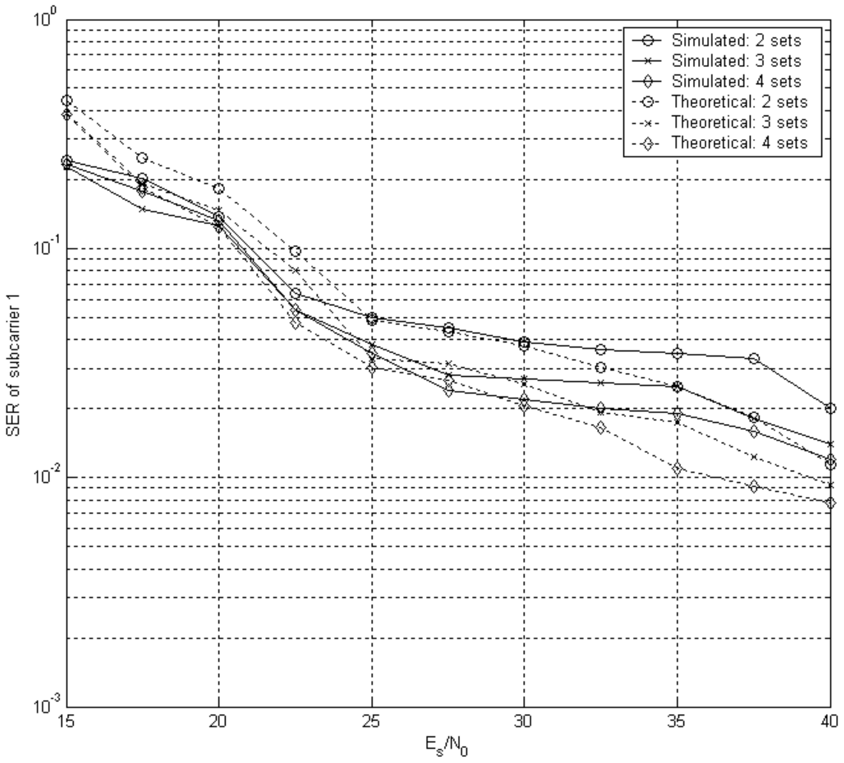
deal with. Among all, interpolating with 4 sets of pilot-symbol information is the most effective method. However, it costs the most memory and complexity at the same time.

4.2 DSP Implementation

According to the results in the previous sections, we have a conclusion. For performance and complexity reasons, it would be better combining 2-D interpolation in time domain with linear interpolation in frequency domain. Although linear interpolation is not as good as 2nd-order one, it is of smaller code size and lower complexity.



(a)



(b)

Figure 4.18: The (a) MSE and (b) SER of carrier 1 with 2-D interpolation using formula 2 with linear interpolation in the frequency domain respectively. $V = 27$ km/h, $fdT = 0.01$.

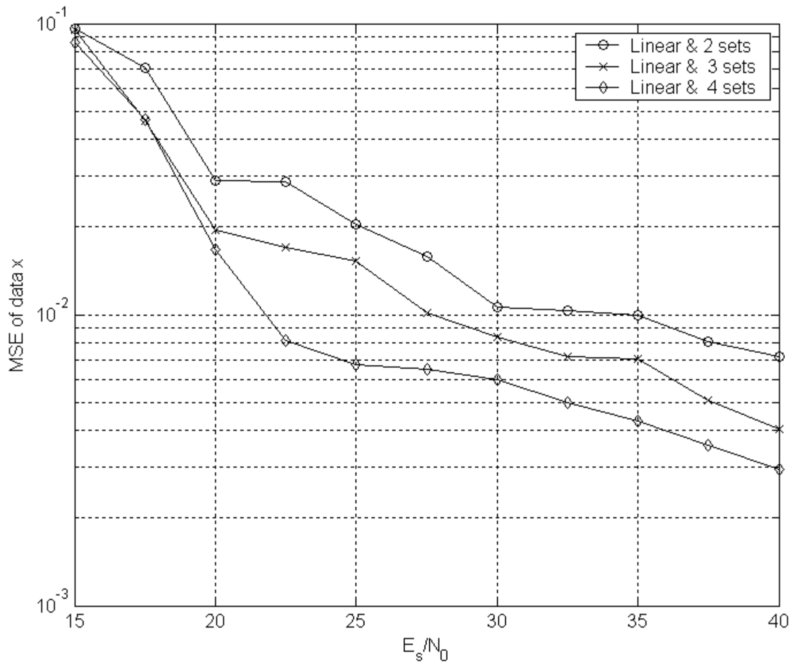


Figure 4.19: MSE of $|\hat{X}_i - X_i|$ on subcarrier 1700 for multipath Rayleigh fading channel.

4.2.1 Introduction to Program Structure

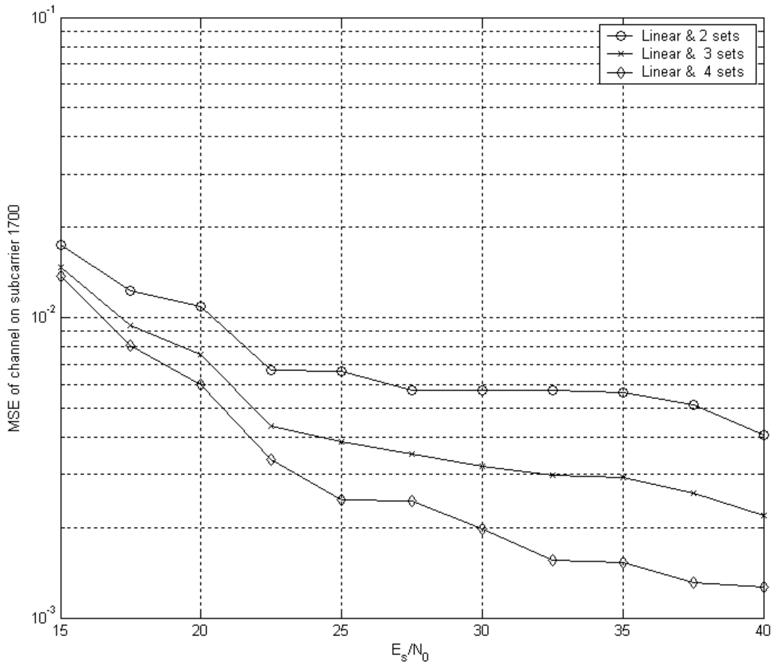
Fig. 4.23 shows program structure of the implemented system, where the key function in channel estimation is Linear-Interp; other are in supporting role.

Function *Modulation(QPSK, 16-QAM, 64-QAM)* maps binary data to the constellation points. We only show the original code for QPSK in Fig. 4.24 for example.

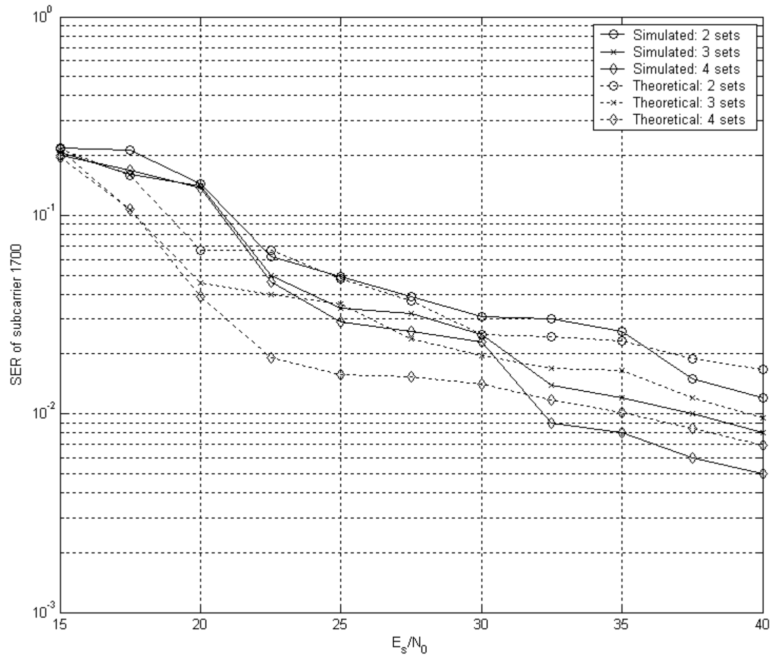
Function *Complex_Mul* is a multiplier which computes complex multiplication to simulate channel effects. The original code is shown in Fig. 4.25.

We add AWGN in the main function instead of an individual function.

The operation in the block *Pilot Location* is that received signal $\mathbf{Y(f)}$ is divided by p=4/3 or −4/3 at pilot locations, i.e., the LS estimator. This function is for convenience in simulation; in real system implementation it can be absorbed into later block.

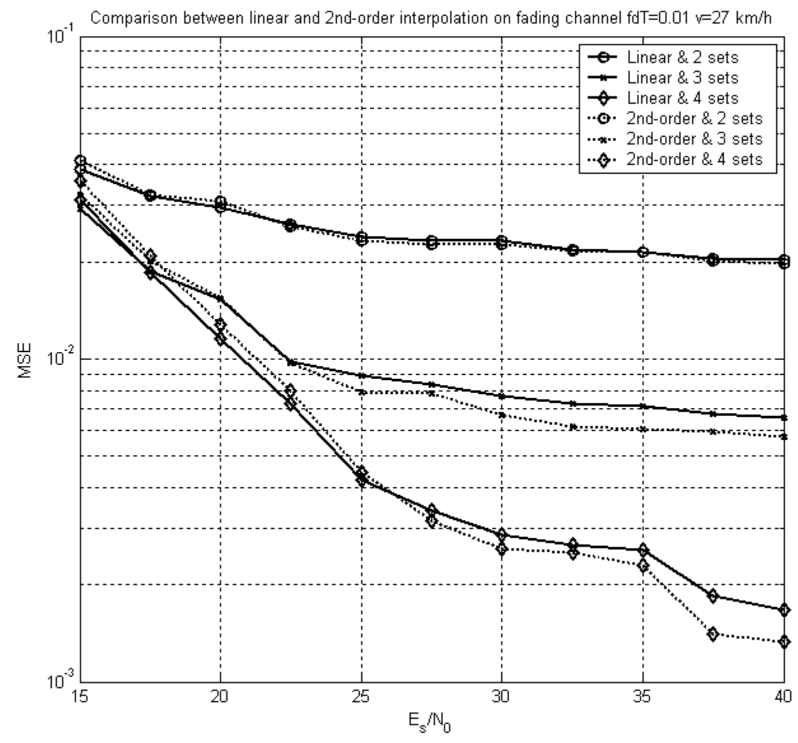


(a)

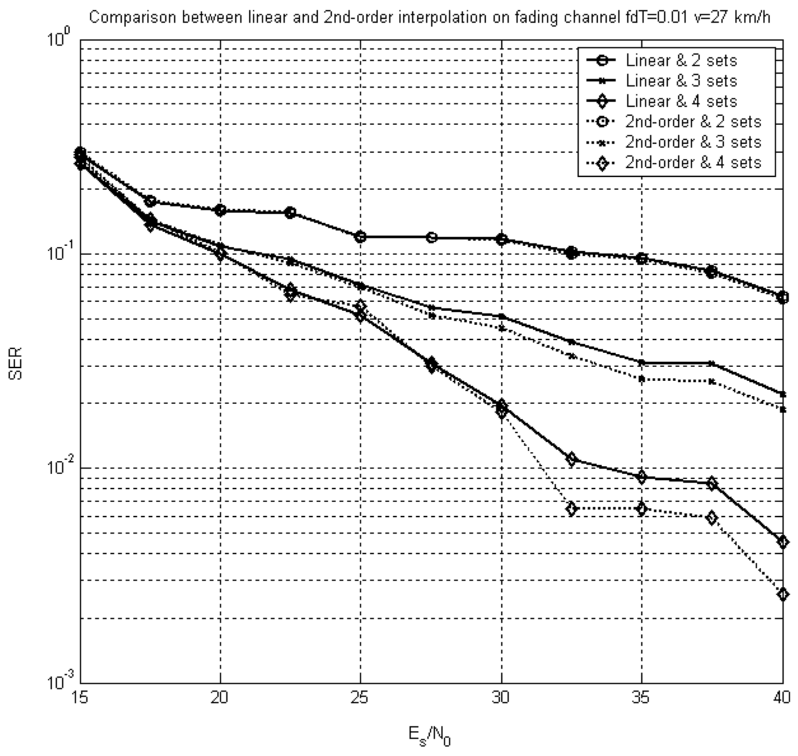


(b)

Figure 4.20: The (a) MSE and (b) SER carrier 1700 with 2-D interpolation using formula 2 with linear and 2nd-order interpolation in the frequency domain respectively. $V = 27$ km/h, $fdT = 0.01$.

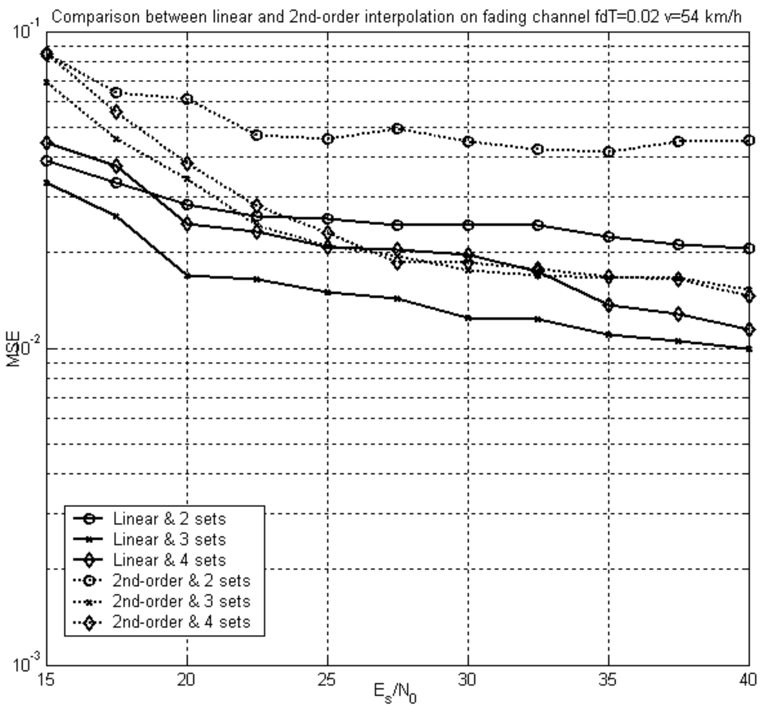


(a)

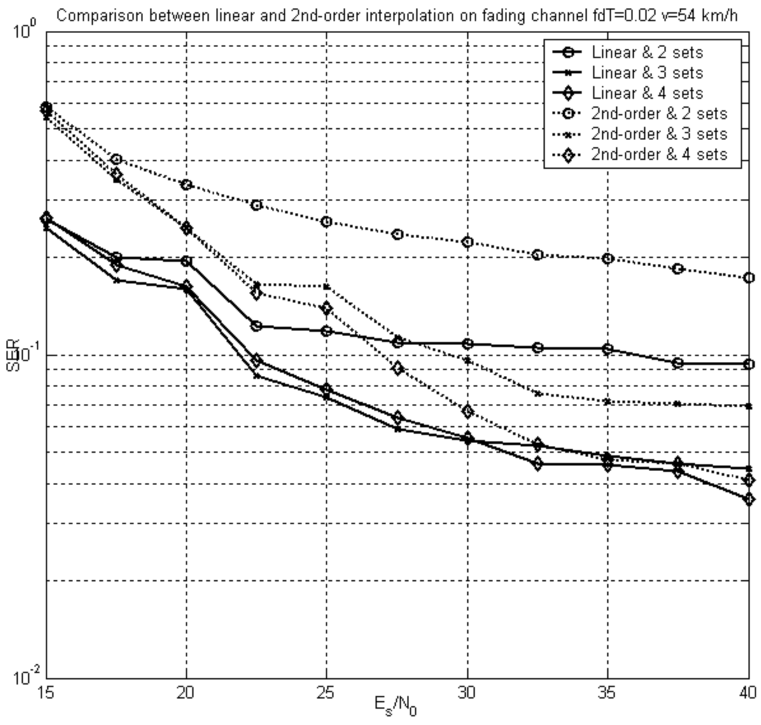


(b)

Figure 4.21: The (a) MSE and (b) SER of the 2-D interpolation using formula 2 with linear and 2nd-order interpolation in the frequency domain respectively. $V = 27$ km/h, $fdT = 0.01$.



(a)



(b)

Figure 4.22: The (a) MSE and (b) SER of the 2-D interpolation using formula 2 with linear and 2nd-order interpolation in the frequency domain respectively. $V = 54$ km/h, $f_dT = 0.02$.

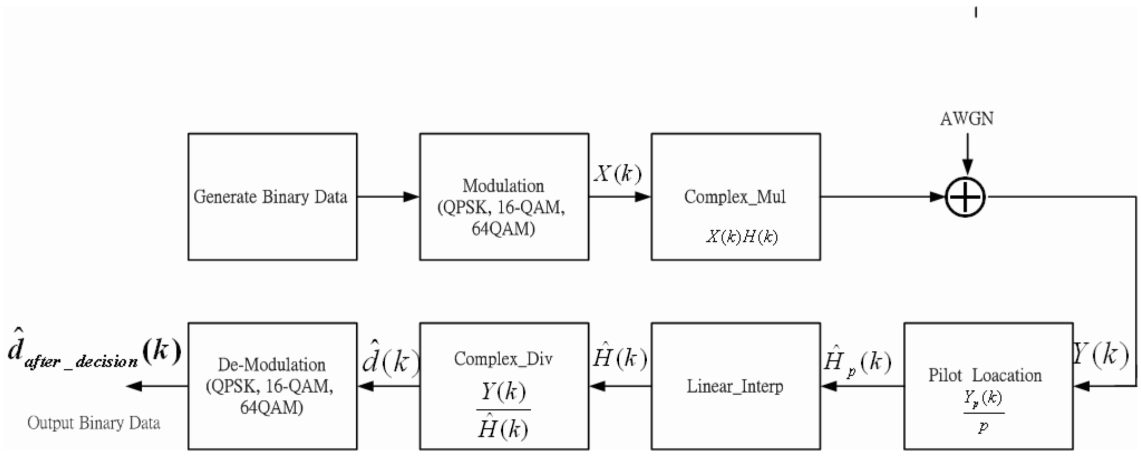


Figure 4.23: Program structure for channel estimation.

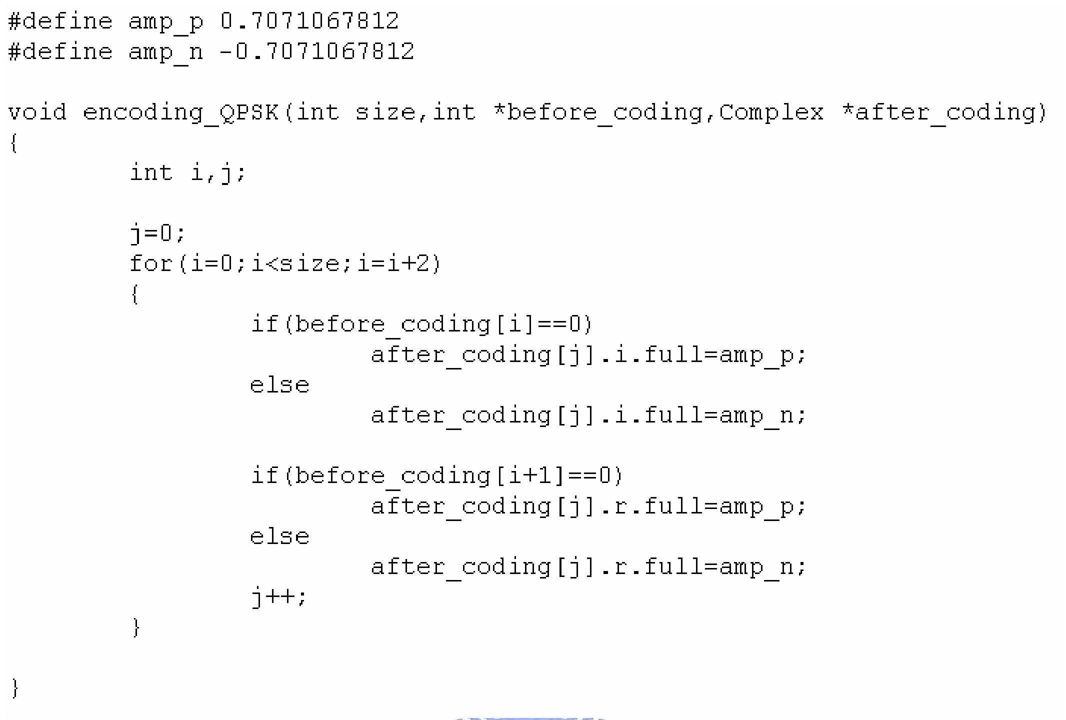
Function *Linear_Interp* is the interpolation part which plays an important role in the channel estimation scheme. The original code is shown in Fig. 4.26.

Function *Complex_Div* is an equalizer where received signal is divided by the estimated channel response and $\hat{d}(k)$ is the output. The code is shown in Fig. 4.27. This function is also for convenience of simmulation; in real implementation its function can be absorbed in the demodulator and the decoder.

Function *De_Modulation* is the de-mapping function which outputs binary data and the mapped data $\hat{d}_{after\_decision}$ in the constellation. The original code is shown in Fig. 4.28.

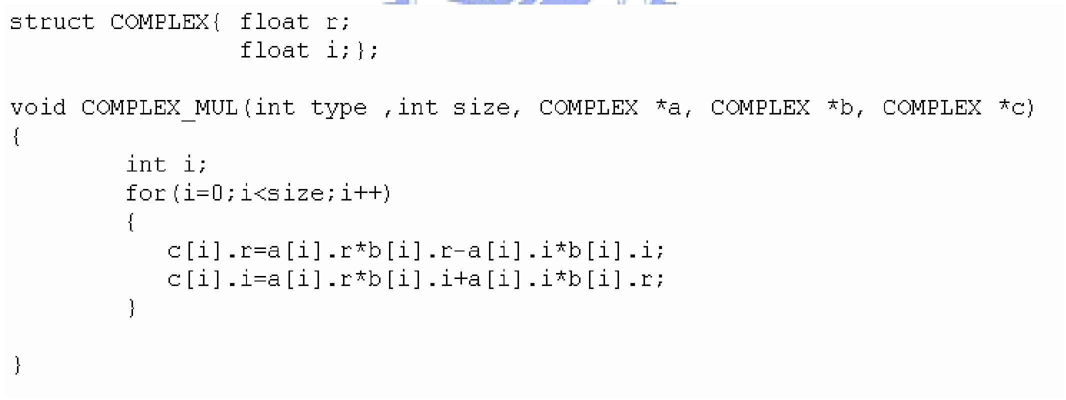
4.2.2 Performance of the Original Program

We use -o3 level optimization in all the following DSP simulations. Originally, we develop the whole system with floating-point computation. Table 4.5 shows the code size, the maximum, minimum, average execution cycles, and the multiples of real-time needed, where multiples of real-time is the execution cycles divided by the available DSP cycles. In our system, one symbol duration is 201.9 μs and there are 2304 samples in a symbol. The clock frequency of the DSP is 600 MHz. Hence the



```
#define amp_p 0.7071067812
#define amp_n -0.7071067812

void encoding_QPSK(int size,int *before_coding,Complex *after_coding)
{
        int i,j;

        j=0;
        for(i=0;i<size;i=i+2)
        {
                if(before_coding[i]==0)
                        after_coding[j].i.full=amp_p;
                else
                        after_coding[j].i.full=amp_n;

                if(before_coding[i+1]==0)
                        after_coding[j].r.full=amp_p;
                else
                        after_coding[j].r.full=amp_n;
                j++;
        }

}
```

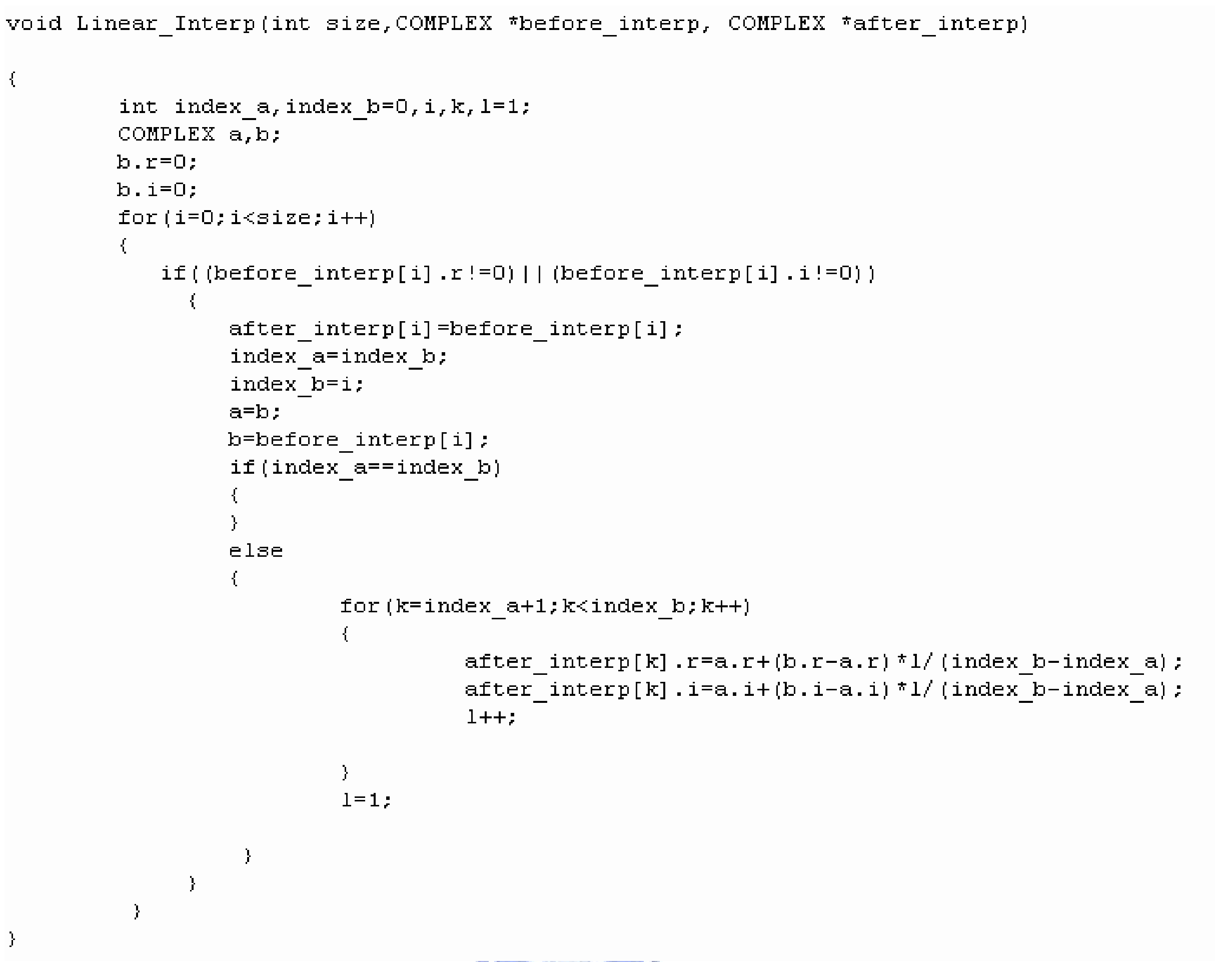
Figure 4.24: Function *Modulation (QPSK).*



```
struct COMPLEX{ float r;
                float i;};

void COMPLEX_MUL(int type ,int size, COMPLEX *a, COMPLEX *b, COMPLEX *c)
{
        int i;
        for(i=0;i<size;i++)
        {
           c[i].r=a[i].r*b[i].r-a[i].i*b[i].i;
           c[i].i=a[i].r*b[i].i+a[i].i*b[i].r;
        }

}
```

Figure 4.25: Function *Complex_Mul.*

available execution cycles are 121140 in a symbol duration, averaging to 52.6 in a sample duration.



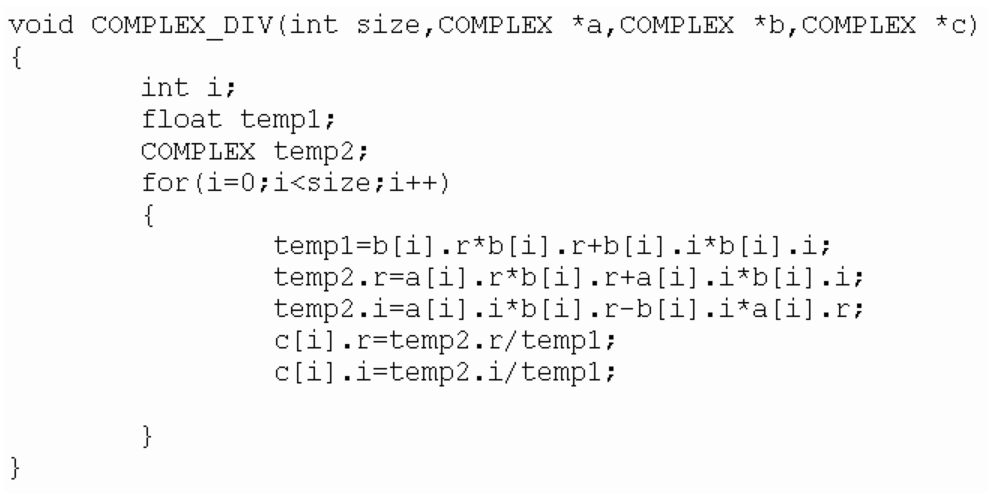
```
void Linear_Interp(int size,COMPLEX *before_interp, COMPLEX *after_interp)

{
        int index_a,index_b=0,i,k,l=1;
        COMPLEX a,b;
        b.r=0;
        b.i=0;
        for(i=0;i<size;i++)
        {
           if((before_interp[i].r!=0)||(before_interp[i].i!=0))
             {
                after_interp[i]=before_interp[i];
                index_a=index_b;
                index_b=i;
                a=b;
                b=before_interp[i];
                if(index_a==index_b)
                {
                }
                else
                {
                        for(k=index_a+1;k<index_b;k++)
                        {
                                after_interp[k].r=a.r+(b.r-a.r)*l/(index_b-index_a);
                                after_interp[k].i=a.i+(b.i-a.i)*l/(index_b-index_a);
                                l++;

                        }
                        l=1;

                 }
             }
         }
}
```

Figure 4.26: Function *Linear_Interp*.

4.2.3 Choice of the Fixed-Point Data Formats

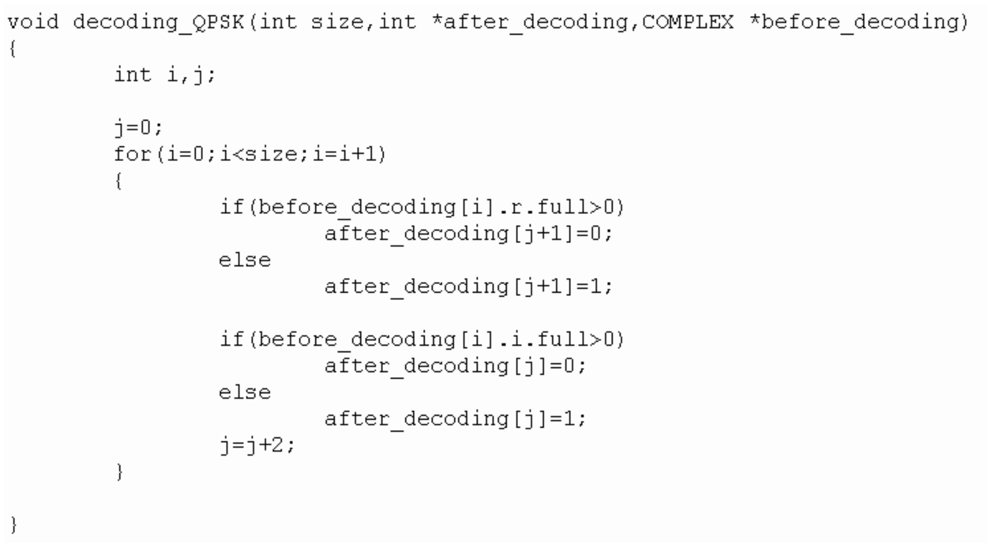
4.2.3.1 32-bit Fixed-Point Operation

Since the C6416 is a fixed-point DSP, floating-point operations are time-consuming. For this reason, employing fixed-point computation will be beneficial to the execution speed. Therefore, we modify the data type of all functions to 32-bit fixed-point data type, i.e., int, in the beginning. To satisfy what is needed for synchronization [24], we define the data format in our system as Q16.15. The Q16.15 format places the sign bit in the leftmost bit, followed by 16 integer bits and then 15 bits fractional bits (Table 4.6). This could support a large dynamic range for all data and is quite



```
void COMPLEX_DIV(int size,COMPLEX *a,COMPLEX *b,COMPLEX *c)
{
        int i;
        float temp1;
        COMPLEX temp2;
        for(i=0;i<size;i++)
        {
                temp1=b[i].r*b[i].r+b[i].i*b[i].i;
                temp2.r=a[i].r*b[i].r+a[i].i*b[i].i;
                temp2.i=a[i].i*b[i].r-b[i].i*a[i].r;
                c[i].r=temp2.r/temp1;
                c[i].i=temp2.i/temp1;

        }
}
```

Figure 4.27: Function *Complex_Div*.



```
void decoding_QPSK(int size,int *after_decoding,COMPLEX *before_decoding)
{
        int i,j;

        j=0;
        for(i=0;i<size;i=i+1)
        {
                if(before_decoding[i].r.full>0)
                        after_decoding[j+1]=0;
                else
                        after_decoding[j+1]=1;

                if(before_decoding[i].i.full>0)
                        after_decoding[j]=0;
                else
                        after_decoding[j]=1;
                j=j+2;
        }

}
```

Figure 4.28: Function *De-modulation(QPSK)*.

sufficient. Table 4.7 shows the profile using 32-bit fixed-point operations.

It has improved very much for most functions except the *Modulation* functions. This is because we add multiplication in these functions to translate the mapped data from floating-point to fixed-point and not only mapping operation (Fig. 4.29). Simple modification of this part should be able to reduce the amount of computation.

The reason why *Complex_Mul* improves the most is that software pipelining is

Table 4.5: Floating-Point Profile of 802.16a DL Channel Estimation Function Blocks

Block	Code Size (Bytes)	Max. Count (Cycles)	Min. Count (Cycles)	Avg. Count (Cycles)	Multiples of Real Time
Modulation (QPSK)	136	24,585	24,585	24,585	0.20
Modulation (16QAM)	332	74,933	73,947	74,391	0.61
Modulation (64QAM)	580	92,212	92,131	92,169	0.76
Complex_Mul	284	899,946	899,231	899,603	7.42
Linear_Interp	548	625,067	467,233	579,794	4.78
Complex_Div	404	1901,807	1,898,949	1,900,051	15.67
De_Modulation	2742	1,372,176	1,223,976	1,305,239	10.77

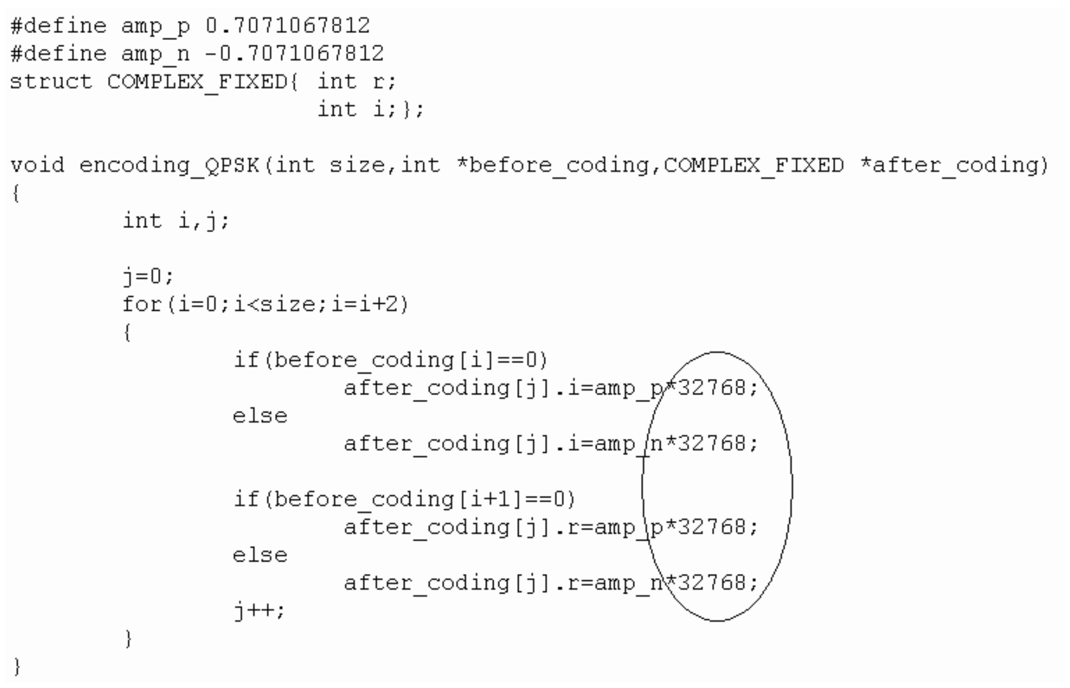
Table 4.6: Q16.15 Bit Fields

Bits	31	30	29	...	15	14	...	1	0
Value	S	I15	I14	...	I0	Q14	...	Q1	Q0

Table 4.7: Fixed-Point 32-Bit Operation Profile of 802.16a DL Channel Estimation Function Blocks

Block	Code size (Bytes)	Max. Count (Cycles)	Min. Count (Cycles)	Avg. Count (Cycles)	Improvement Compared with Table 4.5	Multiples of Real Time
Modulation (QPSK)	126	28,586	28,423	28,545	-16.27%	0.23
Modulation (16QAM)	302	85.549	85.549	85.549	-15.01%	0.70
Modulation (64QAM)	456	104,590	104,274	104,458	-13.33%	0.86
Complex_Mul	472	15,338	15,338	15,338	98.29%	0.12
Linear_Interp	632	504,921	398,510	441,423	23.87%	3.64
Complex_Div	556	554,340	551,557	552,807	70.89%	4.56
De_Modulation	1088	228,666	222,227	225,016	82.76%	1.85

performed in the function *Complex_Mul* which accelerates the execution speed a lot compared to floating-point operations. Fig. 4.30 gives the compile feedback information that provides this information. The loop kernel is shown in Fig. 4.31.

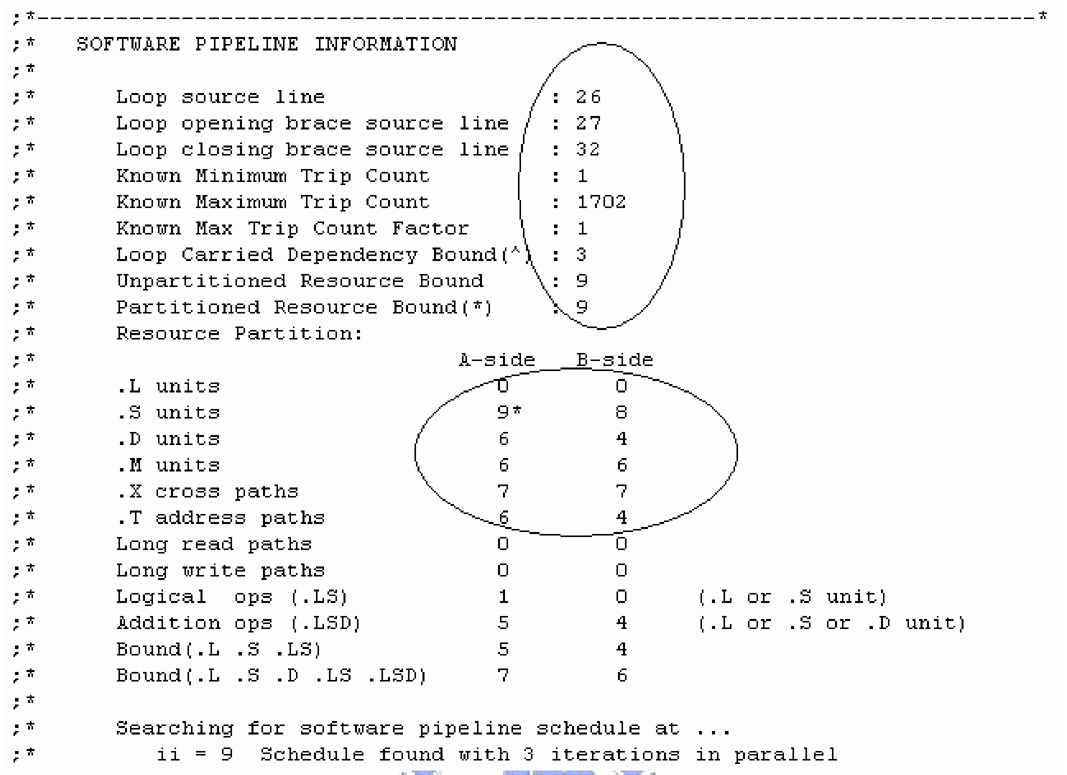


```
#define amp_p 0.7071067812
#define amp_n -0.7071067812
struct COMPLEX_FIXED{ int r;
                      int i;};

void encoding_QPSK(int size,int *before_coding,COMPLEX_FIXED *after_coding)
{
        int i,j;

        j=0;
        for(i=0;i<size;i=i+2)
        {
                if(before_coding[i]==0)
                        after_coding[j].i=amp_p*32768;
                else
                        after_coding[j].i=amp_n*32768;

                if(before_coding[i+1]==0)
                        after_coding[j].r=amp_p*32768;
                else
                        after_coding[j].r=amp_n*32768;
                j++;
        }
}
```

Figure 4.29: Function *Modulation*(*QPSK*) of 32-bit fixed-point operation.

4.2.3.2 16-Bit Fixed-Point Operation

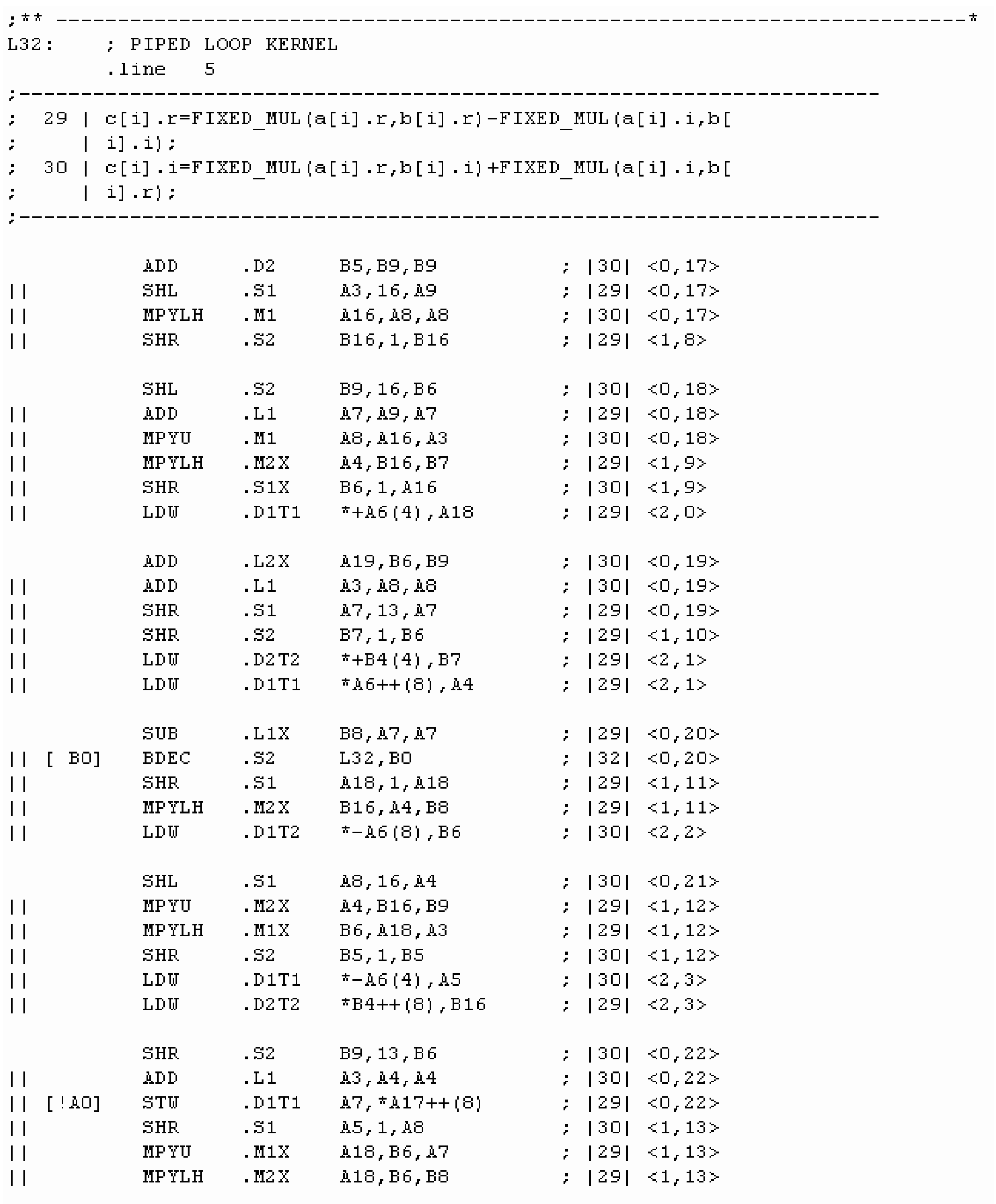
However, we face a problem when optimizing the performance further by adding intrinsic functions into these functions. Most intrinsic functions are specified for the 16-bit data format, but very few for the 32-bit one. Later simulation results will show that 16-bit fixed-point operations are enough to provide the performance near that of floating-point computation. Therefore, we decided to change the data type to 16-bit, i.e., *short*. Because the absolute value of the modulated signal is not larger than 2, one or two bits are enough for the integer part. The data after modulation function is set to Q1.14. Q1.14 format places the sign bit in the leftmost followed by 1 integer bit, and the remainder 14 bits are fraction component (Table 4.8). We can notice that the number of the integer decreases from 15 to 1 and this means the dynamic range supported by Q1.14 would not be sufficient for functions in the follows. To solve this problem, we have to change the bit-field in different functions according to their data values simulated in the floating-point



```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 26
;*      Loop opening brace source line   : 27
;*      Loop closing brace source line   : 32
;*      Known Minimum Trip Count         : 1
;*      Known Maximum Trip Count         : 1702
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 3
;*      Unpartitioned Resource Bound     : 9
;*      Partitioned Resource Bound(*)    : 9
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     9*       8
;*      .D units                     6        4
;*      .M units                     6        6
;*      .X cross paths               7        7
;*      .T address paths             6        4
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        0     (.L or .S unit)
;*      Addition ops (.LSD)          5        4     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             5        4
;*      Bound(.L .S .D .LS .LSD)     7        6
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 9  Schedule found with 3 iterations in parallel
```

Figure 4.30: Software pipelining information of 32-bit fixed-point *Complex_Mul*.

version. Then, we have different types of bit-field setting such as Q5.10, Q4.11, etc. Fig. 4.32 shows the fixed-point data formats used in the simulation. The output bit-field of *Modulation* $X(k)$ is Q1.14. Since the channel gain is no higher than 8, we set the bit-field of $H(k)$ to Q3.12. Therefore, the output of *Complex_Mul* has to be Q4.11 . After adding AWGN, the integer part is right shifted 1 bit and the bit-field setting is Q5.10 ($Y(k)$) to prevent from overflow. Since $p = \pm 4/3$ is of Q1.14 format by our design, so Q5.10/Q1.14 ($Y_p(k)/p$) outputs in Q4.11. In the function *Linear_Interp*, all bit-field format is Q4.11. In the function *Complex_Div*, Q5.10/Q4.11 ($Y(k)/\hat{H}(k)$) outputs Q1.14, for the reason that absolute values of de-modulation constellation points are no greater than 2.



```
;** --------------------------------------------------------------------------*
L32:    ; PIPED LOOP KERNEL
        .line   5
;----------------------------------------------------------------------
;  29 | c[i].r=FIXED_MUL(a[i].r,b[i].r)-FIXED_MUL(a[i].i,b[
;     | i].i);
;  30 | c[i].i=FIXED_MUL(a[i].r,b[i].i)+FIXED_MUL(a[i].i,b[
;     | i].r);
;----------------------------------------------------------------------

           ADD     .D2     B5,B9,B9          ; |30| <0,17>
||         SHL     .S1     A3,16,A9          ; |29| <0,17>
||         MPYLH   .M1     A16,A8,A8         ; |30| <0,17>
||         SHR     .S2     B16,1,B16         ; |29| <1,8>

           SHL     .S2     B9,16,B6          ; |30| <0,18>
||         ADD     .L1     A7,A9,A7          ; |29| <0,18>
||         MPYU    .M1     A8,A16,A3         ; |30| <0,18>
||         MPYLH   .M2X    A4,B16,B7         ; |29| <1,9>
||         SHR     .S1X    B6,1,A16          ; |30| <1,9>
||         LDW     .D1T1   *+A6(4),A18       ; |29| <2,0>

           ADD     .L2X    A19,B6,B9         ; |30| <0,19>
||         ADD     .L1     A3,A8,A8          ; |30| <0,19>
||         SHR     .S1     A7,13,A7          ; |29| <0,19>
||         SHR     .S2     B7,1,B6           ; |29| <1,10>
||         LDW     .D2T2   *+B4(4),B7        ; |29| <2,1>
||         LDW     .D1T1   *A6++(8),A4       ; |29| <2,1>

           SUB     .L1X    B8,A7,A7          ; |29| <0,20>
|| [ B0]   BDEC    .S2     L32,B0            ; |32| <0,20>
||         SHR     .S1     A18,1,A18         ; |29| <1,11>
||         MPYLH   .M2X    B16,A4,B8         ; |29| <1,11>
||         LDW     .D1T2   *-A6(8),B6        ; |30| <2,2>

           SHL     .S1     A8,16,A4          ; |30| <0,21>
||         MPYU    .M2X    A4,B16,B9         ; |29| <1,12>
||         MPYLH   .M1X    B6,A18,A3         ; |29| <1,12>
||         SHR     .S2     B5,1,B5           ; |30| <1,12>
||         LDW     .D1T1   *-A6(4),A5        ; |30| <2,3>
||         LDW     .D2T2   *B4++(8),B16      ; |29| <2,3>

           SHR     .S2     B9,13,B6          ; |30| <0,22>
||         ADD     .L1     A3,A4,A4          ; |30| <0,22>
|| [!A0]   STW     .D1T1   A7,*A17++(8)      ; |29| <0,22>
||         SHR     .S1     A5,1,A8           ; |30| <1,13>
||         MPYU    .M1X    A18,B6,A7         ; |29| <1,13>
||         MPYLH   .M2X    A18,B6,B8         ; |29| <1,13>
```

Figure 4.31: The loop kernel of *Complex_Mul*.

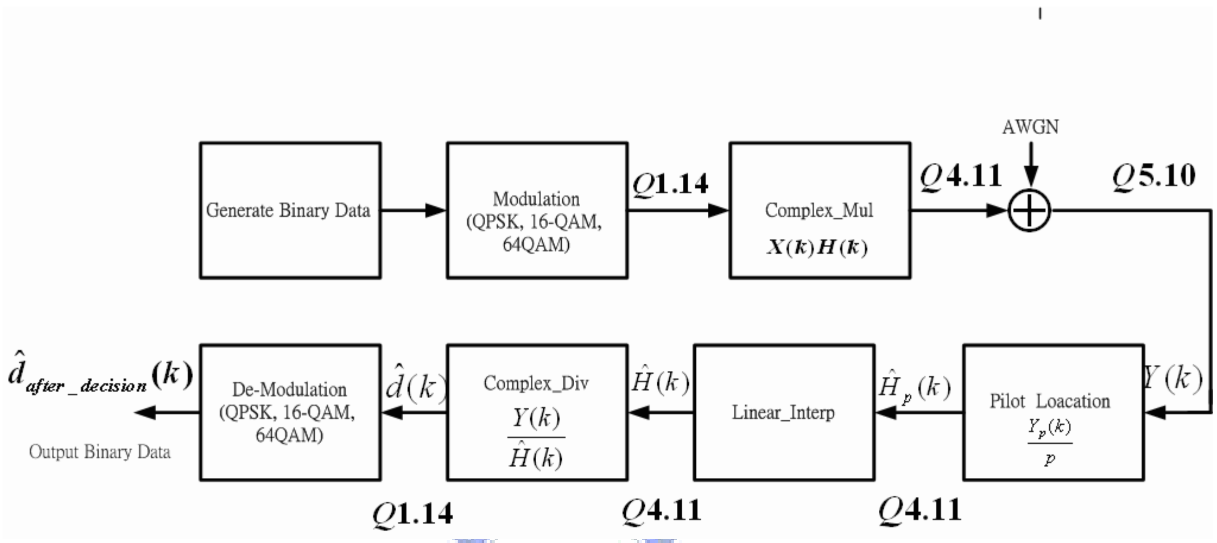
4.2.4 Code Improvement

4.2.4.1 Coding Style Improvement

In the beginning, we declare each complex number with real part and imaginary part separately in the functions *Modulation*, *Complex_Mul*, *Linear_Interp*, *Complex_Div*

Table 4.8: Q1.14 Bit Fields

Bits	15	14	13	...	1	0
Value	S	QI0	Q14	...	Q1	Q0



Generate Binary Data
Modulation (QPSK, 16-QAM, 64QAM)
$Q1.14$
Complex_Mul $X(k)H(k)$
$Q4.11$
AWGN
$Q5.10$
$Y(k)$
Pilot Loacation $\frac{Y_p(k)}{p}$
$\hat{H}_p(k)$
Linear_Interp
$\hat{H}(k)$
Complex_Div $\frac{Y(k)}{\hat{H}(k)}$
$\hat{d}(k)$
De-Modulation (QPSK, 16-QAM, 64QAM)
$\hat{d}_{after\_decision}(k)$
Output Binary Data
$Q1.14$
$Q4.11$
$Q4.11$

Figure 4.32: Fixed-point data formats used in DSP implementation.

Table 4.9: Different Ways of Variable Declaration, Where r Stands for Real Part and i Stands for Imaginary Part

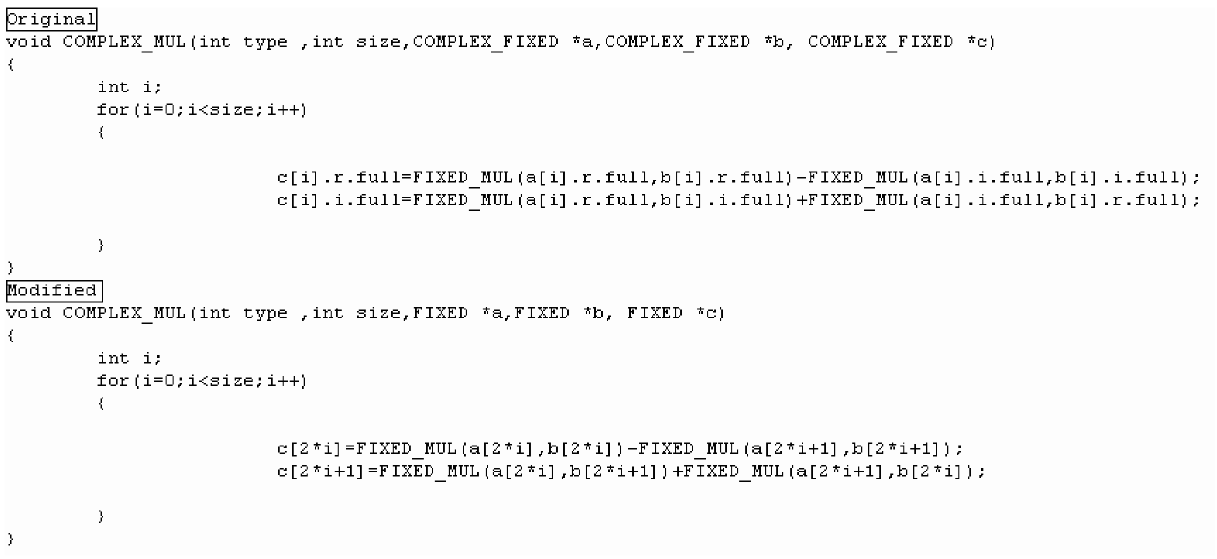
(a)Separate Declaration of Real and Imaginary Parts

Array a[1702].r	a[1701].r	a[1700].r	a[1699].r	...	a[1].r	a[0].r
Array a[1702].i	a[1701].i	a[1700].i	a[1699].i	...	a[1].i	a[0].i

(b)Combination Placement of Real and Imaginary Parts

Array a[1702*2]	a[1701*2+1]	a[1701*2]	a[1700*2+1]	...	a[1]	a[0]
Value	a[1701].i	a[1701].r	a[1700].i	...	a[0].i	a[0].r

and *De-Modulation* (Table 4.9(a)). This is time-wasting in regard to reading or writing memory. Hence, we combine both real and imaginary parts in one register (see Table 4.9(b)). Real parts are located in the even array locations whereas imaginary parts are located in the odd array locations. The previous codes can be found in section 4.2.1 and the modified codes are shown in the next section. An exampe are given in Fig. 4.33 for C code and Fig. 4.34 shows the resulting assembly code.



```
Original
void COMPLEX_MUL(int type ,int size,COMPLEX_FIXED *a,COMPLEX_FIXED *b, COMPLEX_FIXED *c)
{
        int i;
        for(i=0;i<size;i++)
        {

                        c[i].r.full=FIXED_MUL(a[i].r.full,b[i].r.full)-FIXED_MUL(a[i].i.full,b[i].i.full);
                        c[i].i.full=FIXED_MUL(a[i].r.full,b[i].i.full)+FIXED_MUL(a[i].i.full,b[i].r.full);

        }
}
Modified
void COMPLEX_MUL(int type ,int size,FIXED *a,FIXED *b, FIXED *c)
{
        int i;
        for(i=0;i<size;i++)
        {

                        c[2*i]=FIXED_MUL(a[2*i],b[2*i])-FIXED_MUL(a[2*i+1],b[2*i+1]);
                        c[2*i+1]=FIXED_MUL(a[2*i],b[2*i+1])+FIXED_MUL(a[2*i+1],b[2*i]);

        }
}
```

Figure 4.33: Example of different coding styles in C code.

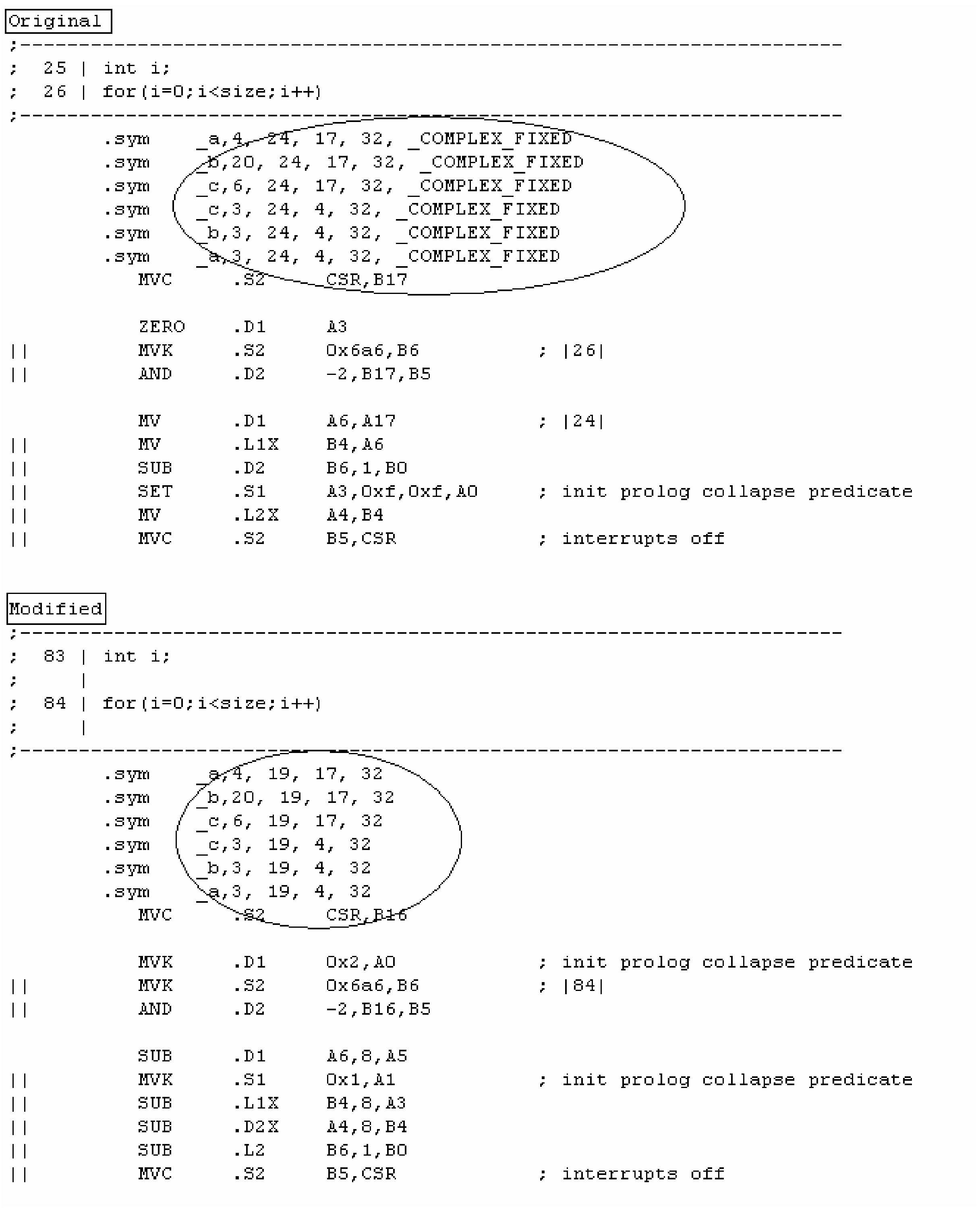
4.2.4.2 Optimization by Using Intrinsic Functions [21]

Intrinsic functions are special functions which help us accelerate the DSP execution speed. We find several useful intrinsic functions which are suitable for our system.

- The _amemd8 and _amemd8_const intrinsics tell the compiler to read the array of shorts with double-word accesses. This causes LDDW and STDW instructions to be issued for the array accesses. The _lo() and _hi() intrinsics break apart a 64-bit double into its lower and upper 32-bit halves. Each of these halves contain two 16-bit values packed in a 32-bit word. To store the results, the _itod() intrinsics assemble 32-bit words back into 64-bit doubles to be stored. Figs. 4.35 and 4.36 show these processes graphically.

- The _dotpn2 and _dotpn2 intrinsic performs real and imaginary portions of complex multiply respectively. The operation is given in Fig. 4.37. We use these functions mostly in the multiply and the divide functions.

- _add2: 32 bits adder.



Original

```
;----------------------------------------------------------------------
;   25 | int i;
;   26 | for(i=0;i<size;i++)
;----------------------------------------------------------------------
        .sym     _a,4, 24, 17, 32, _COMPLEX_FIXED
        .sym     _b,20, 24, 17, 32, _COMPLEX_FIXED
        .sym     _c,6, 24, 17, 32, _COMPLEX_FIXED
        .sym     _c,3, 24, 4, 32, _COMPLEX_FIXED
        .sym     _b,3, 24, 4, 32, _COMPLEX_FIXED
        .sym     _a,3, 24, 4, 32, _COMPLEX_FIXED
           MVC      .S2      CSR,B17

           ZERO     .D1      A3
||         MVK      .S2      0x6a6,B6            ; |26|
||         AND      .D2      -2,B17,B5

           MV       .D1      A6,A17              ; |24|
||         MV       .L1X     B4,A6
||         SUB      .D2      B6,1,B0
||         SET      .S1      A3,0xf,0xf,A0       ; init prolog collapse predicate
||         MV       .L2X     A4,B4
||         MVC      .S2      B5,CSR              ; interrupts off
```

Modified

```
;----------------------------------------------------------------------
;   83 | int i;
;      |
;   84 | for(i=0;i<size;i++)
;      |
;----------------------------------------------------------------------
        .sym     _a,4, 19, 17, 32
        .sym     _b,20, 19, 17, 32
        .sym     _c,6, 19, 17, 32
        .sym     _c,3, 19, 4, 32
        .sym     _b,3, 19, 4, 32
        .sym     _a,3, 19, 4, 32
           MVC      .S2      CSR,B16

           MVK      .D1      0x2,A0              ; init prolog collapse predicate
||         MVK      .S2      0x6a6,B6            ; |84|
||         AND      .D2      -2,B16,B5

           SUB      .D1      A6,8,A5
||         MVK      .S1      0x1,A1              ; init prolog collapse predicate
||         SUB      .L1X     B4,8,A3
||         SUB      .D2X     A4,8,B4
||         SUB      .L2      B6,1,B0
||         MVC      .S2      B5,CSR              ; interrupts off
```

Figure 4.34: Result of different coding styles in complied assembly code.

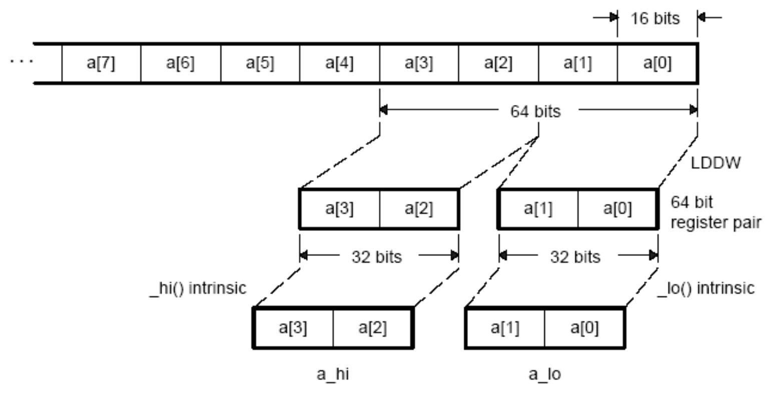


Figure 4.35: Array access in vector sum by LDDW [21].

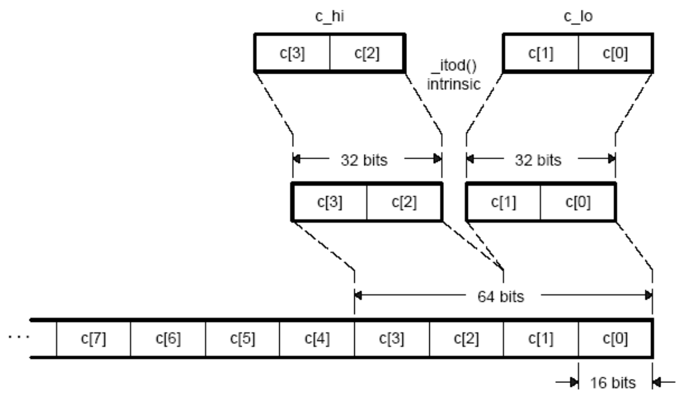


Figure 4.36: Array access in vector sum by STDW [21].

Some code using intrinsic functions is shown in Figs. 4.38 and 4.39 and note that we have added "vec" before the function names to distinguish them from the functions before. Both the functions *Complex_Mul* and *Complex_Div* use intrinsic functions _amemd8, _amemd8_const, _dotpn2, and _dotpn2. We use _add2 when adding AWGN.

4.2.5 Final Version of Fixed-Point 16-Bit Operation

Finally, we check the whole functions to see if there is any optimization could be done to our code. Then, we modify the Linear_Interp function in Fig. 4.40 to Fig.

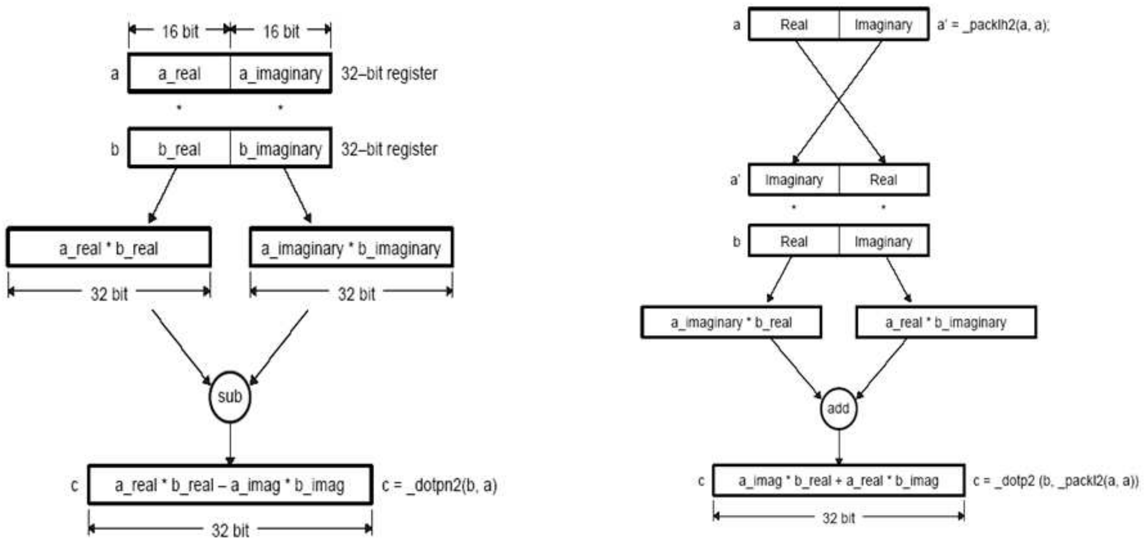


Figure 4.37: Illustration of the _dotp2 and the _dotpn2 intrinsics [21].

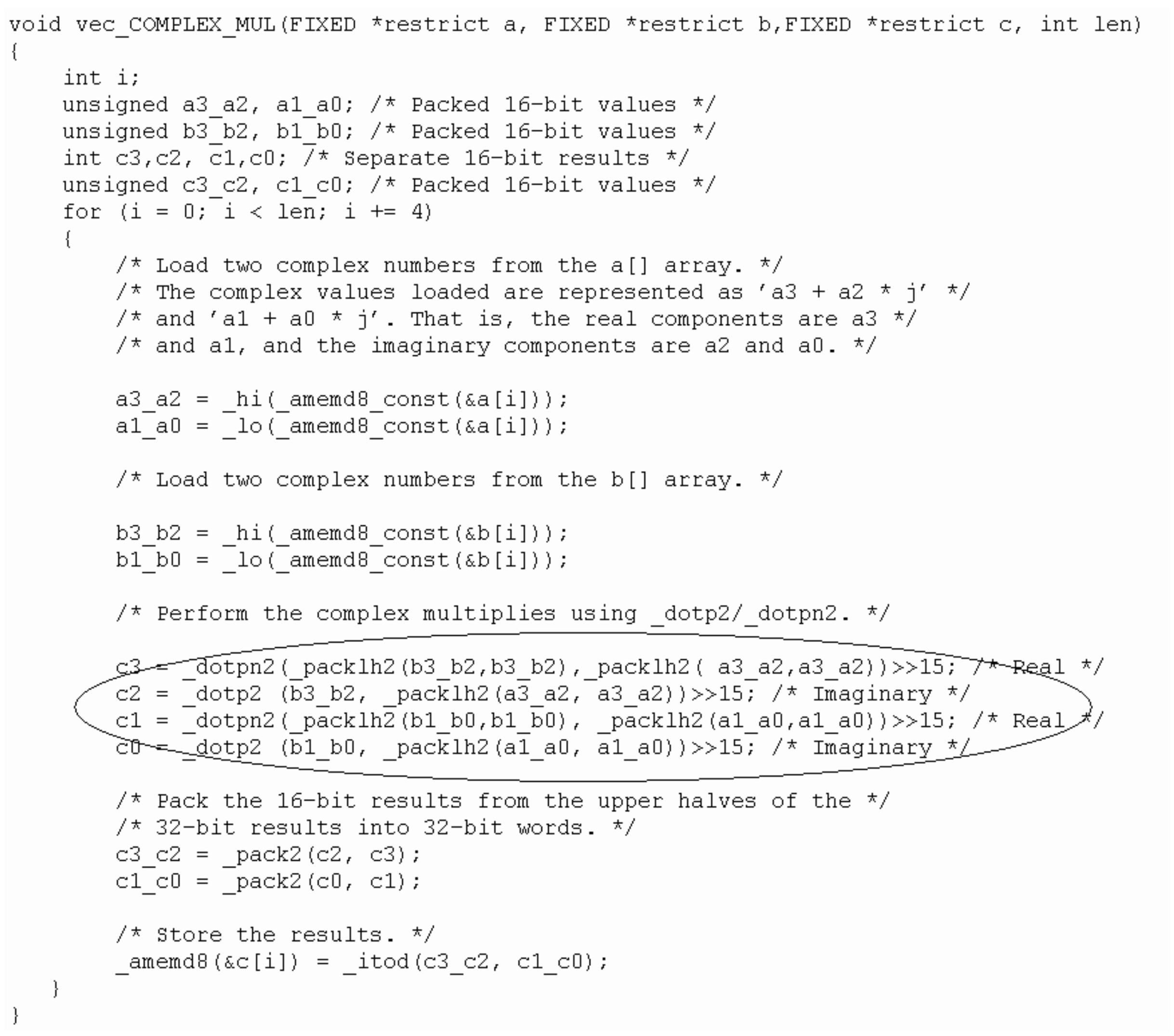
4.41. This is a kind of coding style improvement.

The software pipelining is employed to the modified part in *Linear_Interp* (Fig. 4.43) and the loop kernel is shown in Fig. 4.42. We have better performance in our simulation after this modification which is shown in Table 4.10. From the results, we can find that the original interpolation loop takes much more time but much less after we modified the code style.

Code refinement enhances our performance. Some of these functions reach the real time requirement and some almost reach it. This is because the division in *Complex_Div* is time-consuming and it costs many more cycles to execute it than multiplications.

4.2.5.1 Execution Efficiency

Our DSP can execute 6 additions and 2 multiplications in one cycles. We have 4 multiplications and 2 additions per sample in function *Complex_Mul*, and 8 complex multiplications, 4 additions, 2 division per sample in *Complex_Div*. Each division



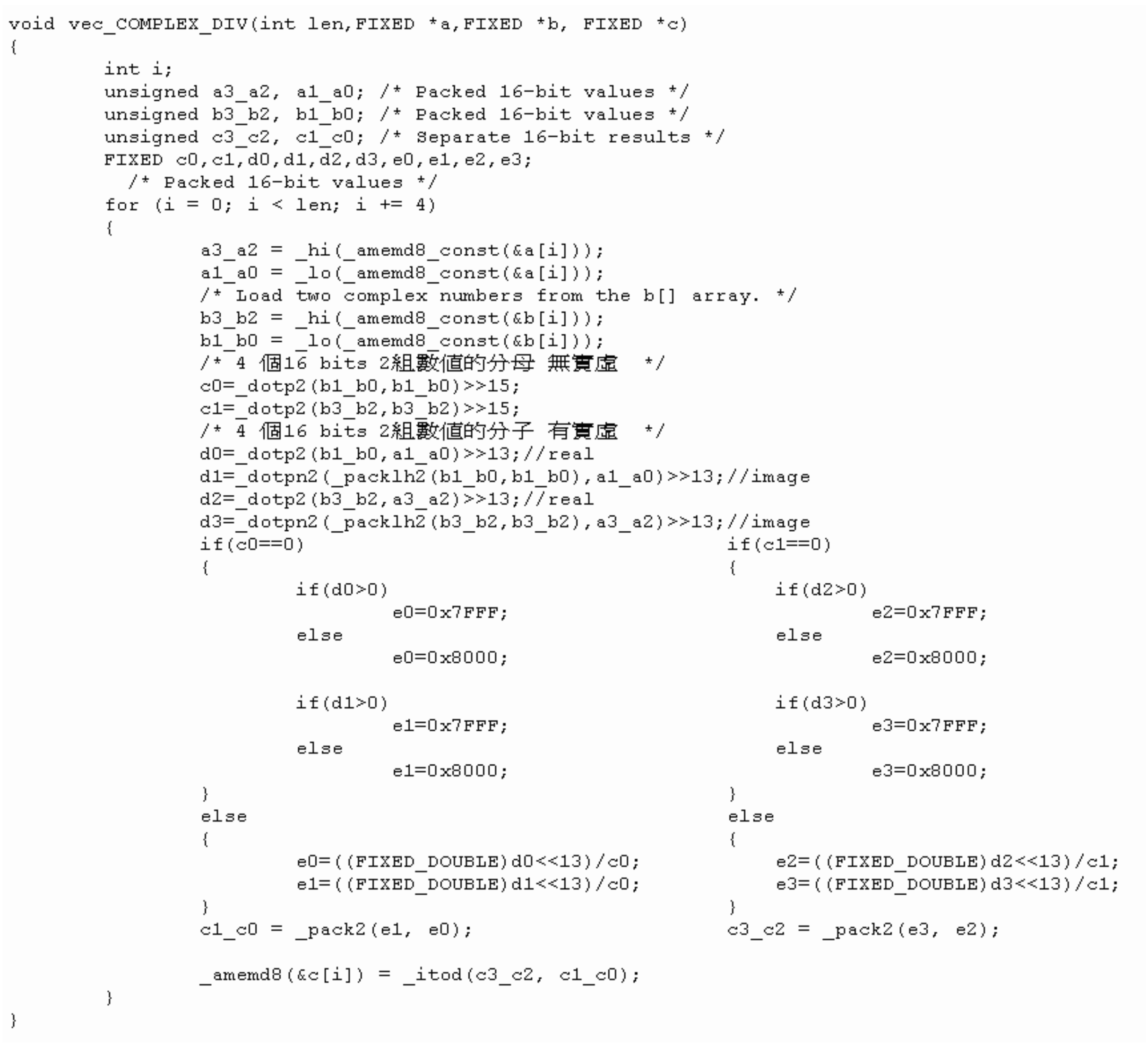
```
void vec_COMPLEX_MUL(FIXED *restrict a, FIXED *restrict b,FIXED *restrict c, int len)
{
    int i;
    unsigned a3_a2, a1_a0; /* Packed 16-bit values */
    unsigned b3_b2, b1_b0; /* Packed 16-bit values */
    int c3,c2, c1,c0; /* Separate 16-bit results */
    unsigned c3_c2, c1_c0; /* Packed 16-bit values */
    for (i = 0; i < len; i += 4)
    {
        /* Load two complex numbers from the a[] array. */
        /* The complex values loaded are represented as 'a3 + a2 * j' */
        /* and 'a1 + a0 * j'. That is, the real components are a3 */
        /* and a1, and the imaginary components are a2 and a0. */

        a3_a2 = _hi(_amemd8_const(&a[i]));
        a1_a0 = _lo(_amemd8_const(&a[i]));

        /* Load two complex numbers from the b[] array. */

        b3_b2 = _hi(_amemd8_const(&b[i]));
        b1_b0 = _lo(_amemd8_const(&b[i]));

        /* Perform the complex multiplies using _dotp2/_dotpn2. */

        c3 = _dotpn2(_packlh2(b3_b2,b3_b2),_packlh2( a3_a2,a3_a2))>>15; /* Real */
        c2 = _dotp2 (b3_b2, _packlh2(a3_a2, a3_a2))>>15; /* Imaginary */
        c1 = _dotpn2(_packlh2(b1_b0,b1_b0), _packlh2(a1_a0,a1_a0))>>15; /* Real */
        c0 = _dotp2 (b1_b0, _packlh2(a1_a0, a1_a0))>>15; /* Imaginary */

        /* Pack the 16-bit results from the upper halves of the */
        /* 32-bit results into 32-bit words. */
        c3_c2 = _pack2(c2, c3);
        c1_c0 = _pack2(c0, c1);

        /* Store the results. */
        _amemd8(&c[i]) = _itod(c3_c2, c1_c0);
    }
}
```

Figure 4.38: Function *vec_Complex_Mul*.

needs 22 cycles in the complied code for 32 bits operation and 21 cycles for 16-bit operation. Therefore, we need a minimum cycles max{4/2,2/6}×1702=3404 cycles per symbol in *Complex_Mul* and (2×22/2+max{8/2,4/6})×1702=44252 per symbol for 32-bit operation, and 1702 and 39146 cycles for 16-bit operation, respectively. 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:

$$\text{Efficiency} = \frac{\text{Minimum Cycles Needed}}{\text{Practical Execution Cycles}}, \tag{4.12}$$

which can show how much improvement is achieved after our optimization.

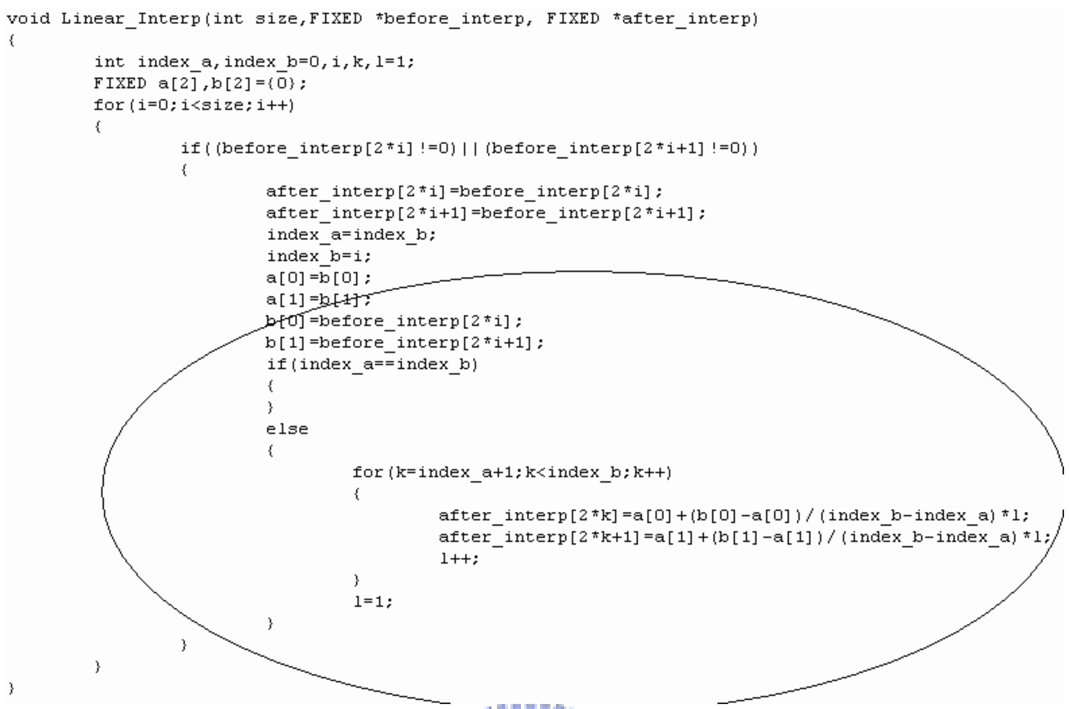


```
void vec_COMPLEX_DIV(int len,FIXED *a,FIXED *b, FIXED *c)
{
        int i;
        unsigned a3_a2, a1_a0; /* Packed 16-bit values */
        unsigned b3_b2, b1_b0; /* Packed 16-bit values */
        unsigned c3_c2, c1_c0; /* Separate 16-bit results */
        FIXED c0,c1,d0,d1,d2,d3,e0,e1,e2,e3;
          /* Packed 16-bit values */
        for (i = 0; i < len; i += 4)
        {
                a3_a2 = _hi(_amemd8_const(&a[i]));
                a1_a0 = _lo(_amemd8_const(&a[i]));
                /* Load two complex numbers from the b[] array. */
                b3_b2 = _hi(_amemd8_const(&b[i]));
                b1_b0 = _lo(_amemd8_const(&b[i]));
                /* 4 個16 bits 2組數值的分母 無實虛  */
                c0=_dotp2(b1_b0,b1_b0)>>15;
                c1=_dotp2(b3_b2,b3_b2)>>15;
                /* 4 個16 bits 2組數值的分子 有實虛  */
                d0=_dotp2(b1_b0,a1_a0)>>13;//real
                d1=_dotpn2(_packlh2(b1_b0,b1_b0),a1_a0)>>13;//image
                d2=_dotp2(b3_b2,a3_a2)>>13;//real
                d3=_dotpn2(_packlh2(b3_b2,b3_b2),a3_a2)>>13;//image
                if(c0==0)                                   if(c1==0)
                {                                           {
                        if(d0>0)                                if(d2>0)
                                e0=0x7FFF;                              e2=0x7FFF;
                        else                                    else
                                e0=0x8000;                              e2=0x8000;

                        if(d1>0)                                if(d3>0)
                                e1=0x7FFF;                              e3=0x7FFF;
                        else                                    else
                                e1=0x8000;                              e3=0x8000;
                }                                           }
                else                                        else
                {                                           {
                        e0=((FIXED_DOUBLE)d0<<13)/c0;           e2=((FIXED_DOUBLE)d2<<13)/c1;
                        e1=((FIXED_DOUBLE)d1<<13)/c0;           e3=((FIXED_DOUBLE)d3<<13)/c1;
                }                                           }
                c1_c0 = _pack2(e1, e0);                     c3_c2 = _pack2(e3, e2);

                _amemd8(&c[i]) = _itod(c3_c2, c1_c0);
        }
}
```

Figure 4.39: Function *vec_Complex_Div*.

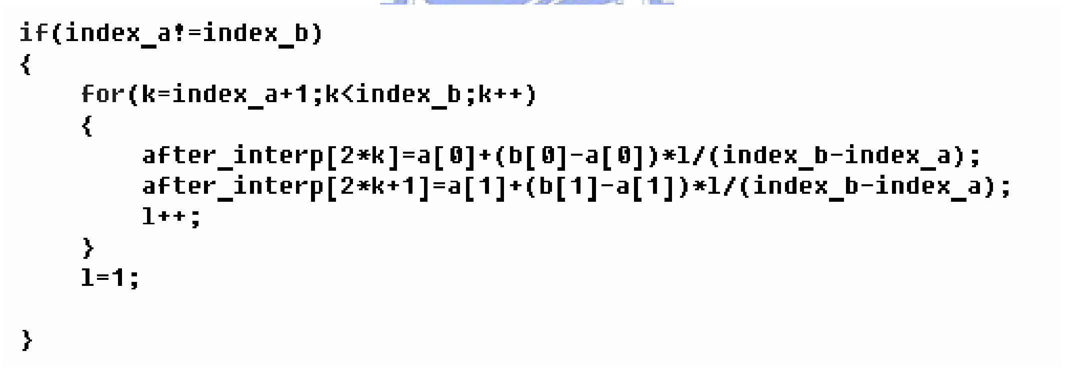
We can see from Tables 4.11 and 4.12 list the efficiency of *Complex_Mul* and *Complex_Div*. We get a good performance after all the code improvements done to 16-bits fixed-point operation. Fig. 4.44 is the software pipelining information for *Complex_Mul* with 16-bits fixed-point operation. The maximum trip count is 851 which is half the value of 32-bit fixed-point operation (see Fig. 4.30).

For *Complex_Div*, we can find that the efficiency is small and this is because there are a lot of load or store operations in this function (Fig. 4.39). Besides, we



```
void Linear_Interp(int size,FIXED *before_interp, FIXED *after_interp)
{
        int index_a,index_b=0,i,k,l=1;
        FIXED a[2],b[2]={0};
        for(i=0;i<size;i++)
        {
                if((before_interp[2*i]!=0)||(before_interp[2*i+1]!=0))
                {
                        after_interp[2*i]=before_interp[2*i];
                        after_interp[2*i+1]=before_interp[2*i+1];
                        index_a=index_b;
                        index_b=i;
                        a[0]=b[0];
                        a[1]=b[1];
                        b[0]=before_interp[2*i];
                        b[1]=before_interp[2*i+1];
                        if(index_a==index_b)
                        {
                        }
                        else
                        {
                                for(k=index_a+1;k<index_b;k++)
                                {
                                        after_interp[2*k]=a[0]+(b[0]-a[0])/(index_b-index_a)*l;
                                        after_interp[2*k+1]=a[1]+(b[1]-a[1])/(index_b-index_a)*l;
                                        l++;
                                }
                                l=1;
                        }
                }
        }
}
```

Figure 4.40: Original interpolation loop.

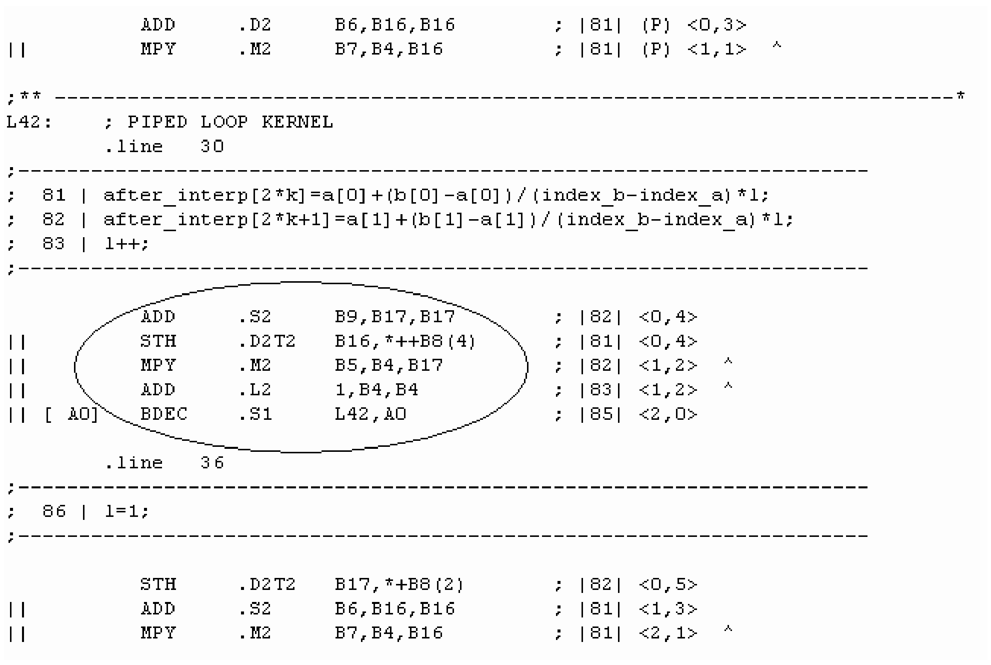


```
if(index_a!=index_b)
{
    for(k=index_a+1;k<index_b;k++)
    {
        after_interp[2*k]=a[0]+(b[0]-a[0])*l/(index_b-index_a);
        after_interp[2*k+1]=a[1]+(b[1]-a[1])*l/(index_b-index_a);
        l++;
    }
    l=1;

}
```

Figure 4.41: Final version of the interpolation loop.

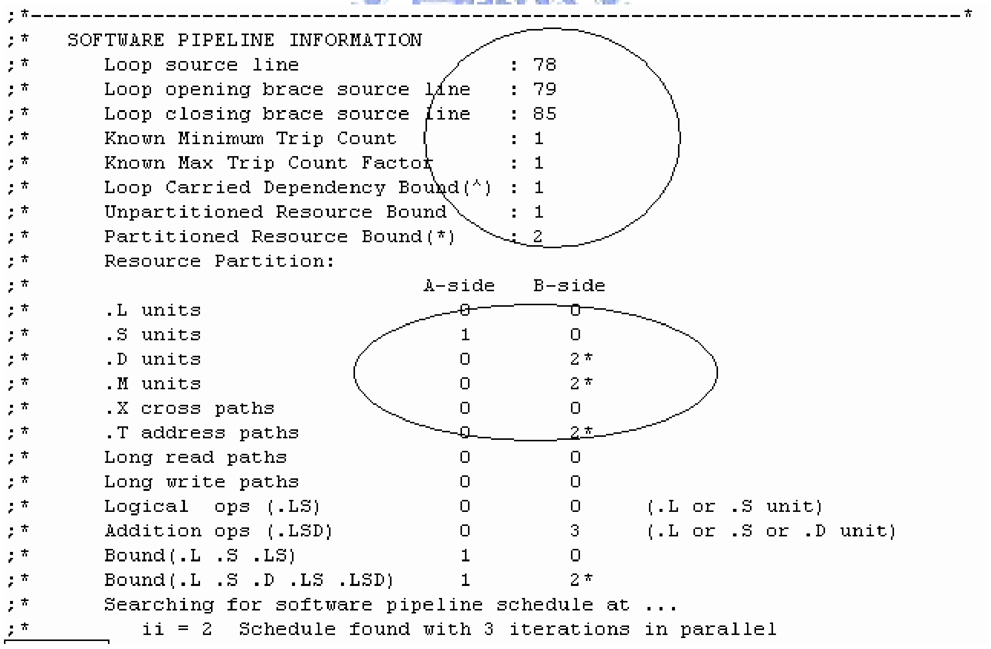
have if-else operations in this function for preventing the dividing by zero situation. Therefore, software pipelining cannot be done to accelerate the execution speed when codes contain a "control code." At least, adding intrinsic functions improves the performance compared with 32-bit fixed-point operation.

We use 567×4 divisions and 567×4 multiplications and 567×8 additions per symbol in *Linear_Interp*. Therefore, the minimum cycles are 26082 for 32-bit fixed-point



```
          ADD     .D2     B6,B16,B16        ; |81| (P) <0,3>
||        MPY     .M2     B7,B4,B16         ; |81| (P) <1,1>  ^

;** --------------------------------------------------------------------------*
L42:    ; PIPED LOOP KERNEL
        .line   30
;---------------------------------------------------------------------
;  81 | after_interp[2*k]=a[0]+(b[0]-a[0])/(index_b-index_a)*l;
;  82 | after_interp[2*k+1]=a[1]+(b[1]-a[1])/(index_b-index_a)*l;
;  83 | l++;
;---------------------------------------------------------------------

          ADD     .S2     B9,B17,B17        ; |82| <0,4>
||        STH     .D2T2   B16,*++B8(4)      ; |81| <0,4>
||        MPY     .M2     B5,B4,B17         ; |82| <1,2>  ^
||        ADD     .L2     1,B4,B4           ; |83| <1,2>  ^
|| [ A0]  BDEC    .S1     L42,A0            ; |85| <2,0>

        .line   36
;---------------------------------------------------------------------
;  86 | l=1;
;---------------------------------------------------------------------

          STH     .D2T2   B17,*+B8(2)       ; |82| <0,5>
||        ADD     .S2     B6,B16,B16        ; |81| <1,3>
||        MPY     .M2     B7,B4,B16         ; |81| <2,1>  ^
```

Figure 4.42: Loop kernel of modified assembly code in *Linear_Interp*.



```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*      Loop source line                 : 78
;*      Loop opening brace source line   : 79
;*      Loop closing brace source line   : 85
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 1
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                    0        0
;*      .S units                    1        0
;*      .D units                    0        2*
;*      .M units                    0        2*
;*      .X cross paths              0        0
;*      .T address paths            0        2*
;*      Long read paths             0        0
;*      Long write paths            0        0
;*      Logical  ops (.LS)          0        0     (.L or .S unit)
;*      Addition ops (.LSD)         0        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS)            1        0
;*      Bound(.L .S .D .LS .LSD)    1        2*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 3 iterations in parallel
```

Figure 4.43: Software pipelining information of the modified loop in *Linear_Interp*
.

operation and 24381 for 16-bit fixed-point operation. The efficiency of *Linear_Interp* is listed in Table 4.13. We only reach 36.01 % efficiency and this is because there is

Table 4.10: Fixed-Point 16-bit Operation with Coding Style Modified Profile of 802.16a DL Channel Estimation Function Blocks

Block	Code size (Bytes)	Max. Count (Cycles)	Min. Count (Cycles)	Avg. Count (Cycles)	Improvement Compared with Table4.7)	Multiples of Real Time
Modulation (QPSK)	124	27,775	27,775	27,775	2.69%	0.22
Modulation (16QAM)	296	70.637	69.429	73.802	13.73%	0.60
Modulation (64QAM)	396	101,585	101,082	101,308	3.02%	0.71
Complex_Mul	272	3,421	3,421	3,421	77.69%	0.02
Linear_Interp	332	81,713	47,689	67,705	84.66%	0.55
Complex_Div	428	163,108	162,793	162,960	70.57%	1.15
De_Modulation	1068	149,796	146,600	148,169	34.15%	1.05

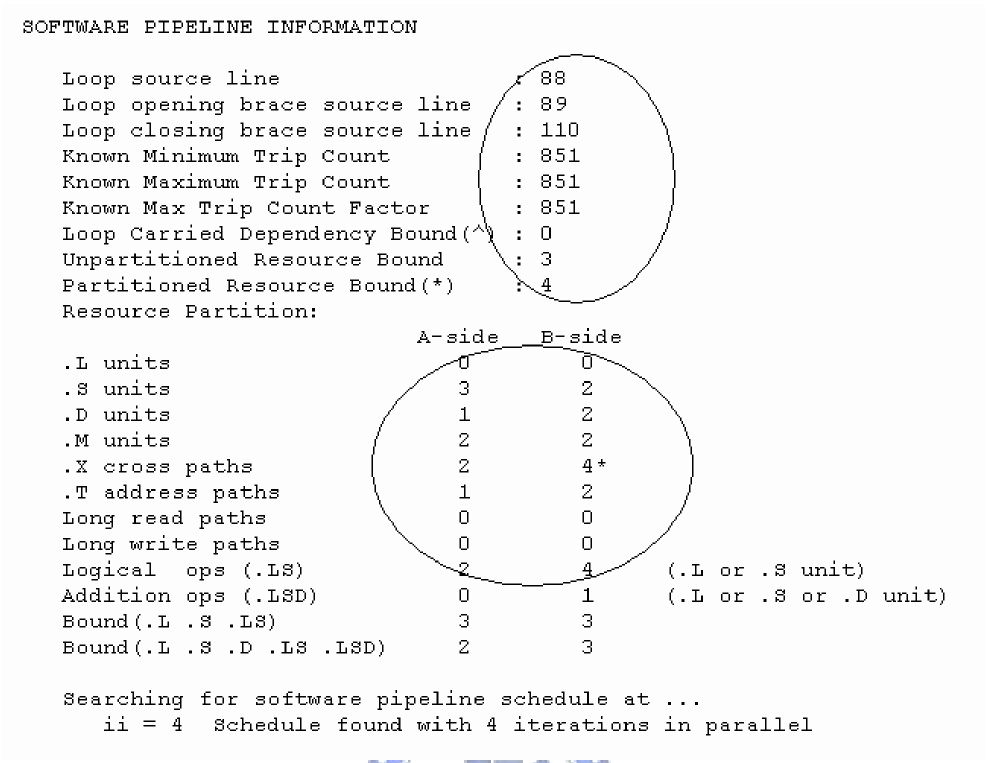
Table 4.11: Performance Comparison Between Different Data Types of *Complex_Mul*

Block	Execution Cycles per Symbol	Minimum Cycles Needed per Symbol	Efficiency
Complex_Mul (float)	899,231	3,404	0.37%
Complex_Mul (32-bits)	15,338	3,404	22.19%
Complex_Mul (16-bits)	3,421	1,702	49.75%

Table 4.12: Performance Comparison Between Different Data Types of *Complex_Div*

Block	Execution Cycles per Symbol	Minimum Cycles Needed per Symbol	Efficiency
Complex_Div (float)	1,900,051	44,252	4.1%
Complex_Div (32-bits)	688,850	44,252	6.42%
Complex_Div (16-bits)	162,960	39,146	24.02%

also control code in this function. Coding style improvement would be useful for this. Over all, it seems that the performance of our interpolation scheme is not efficient enough for there are many divisions. However, it reach the real time requirement.



```
SOFTWARE PIPELINE INFORMATION

   Loop source line                   : 88
   Loop opening brace source line     : 89
   Loop closing brace source line     : 110
   Known Minimum Trip Count           : 851
   Known Maximum Trip Count           : 851
   Known Max Trip Count Factor        : 851
   Loop Carried Dependency Bound(^)   : 0
   Unpartitioned Resource Bound       : 3
   Partitioned Resource Bound(*)      : 4
   Resource Partition:
                              A-side   B-side
   .L units                     0        0
   .S units                     3        2
   .D units                     1        2
   .M units                     2        2
   .X cross paths               2        4*
   .T address paths             1        2
   Long read paths              0        0
   Long write paths             0        0
   Logical  ops (.LS)           2        4     (.L or .S unit)
   Addition ops (.LSD)          0        1     (.L or .S or .D unit)
   Bound(.L .S .LS)             3        3
   Bound(.L .S .D .LS .LSD)     2        3

   Searching for software pipeline schedule at ...
      ii = 4  Schedule found with 4 iterations in parallel
```

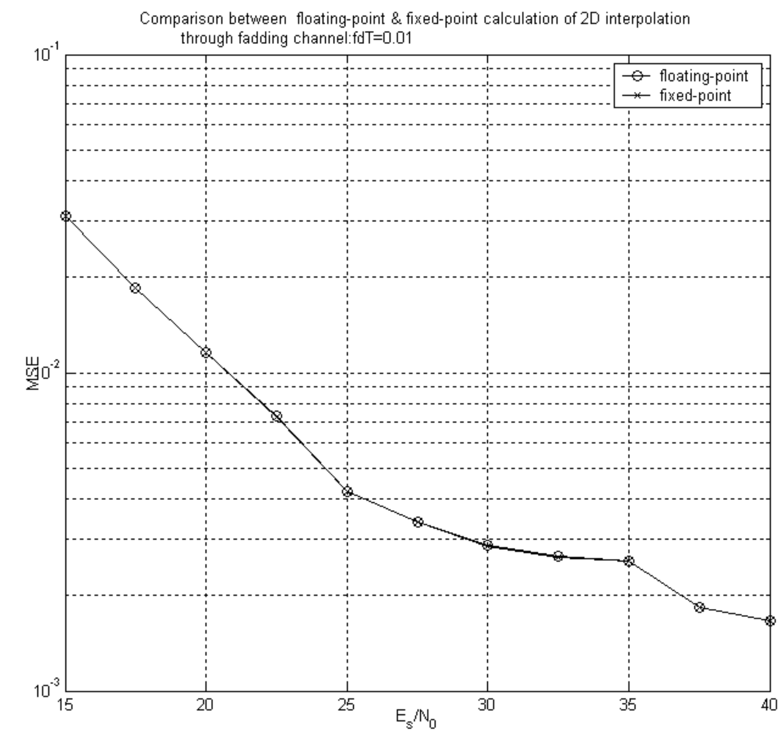
Figure 4.44: Software pipelining information of 16-bits fixed-point of *Complex_Mul*.

Table 4.13: Performance Comparison Between Different Data Types of *Linear_Interp*

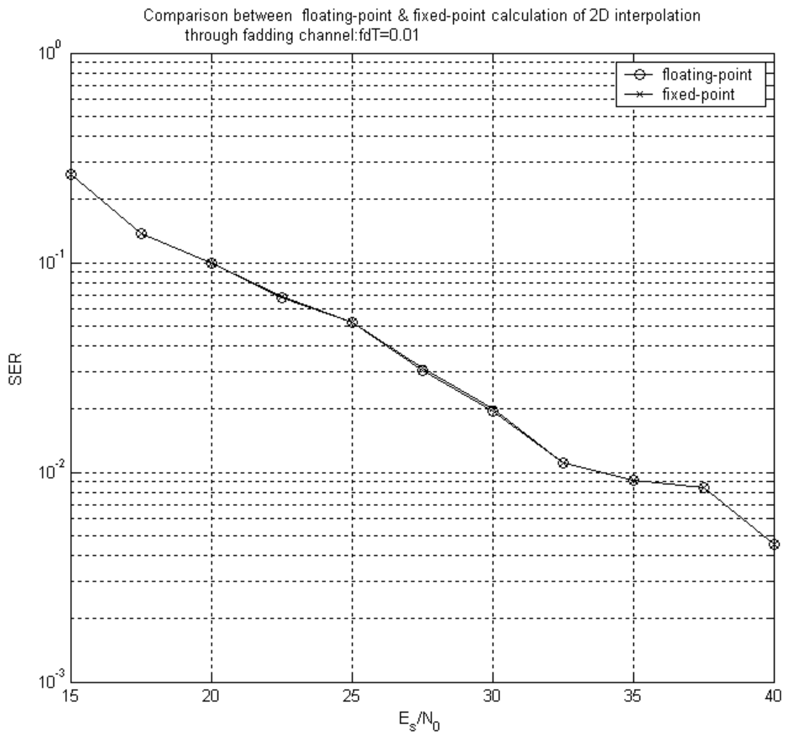
Block	Execution Cycles per Symbol	Minimum Cycles Needed per Symbol	Efficiency
Linear_Interp (float)	467,233	12,852	2.75%
Linear_Interp (32-bits)	441,423	26,082	5.9%
Linear_Interp (16-bits)	67,705	24,381	36.01%

4.2.6 Summary

We have done much work to accelerate the DSP execution speed such as changing data type, code style refinement, and using intrinsics. Compared with theoretical execution cycles, however, the performance is still not very good. The reason may be that there is still bad coding-style in our programs which leads to lower down the speed. At least, our interpolation scheme achieved the goal of 0.5 multiples of real time. It means we can finish interpolation of approximately two symbols in one symbol period. The accuracy of fixed-point 16-bit operation is almost the same with floating-point one although there is quantization error. Since we use Q1.14 format, the accuracy can reach $1/2^{14} = 6.1 \times 10^{-5}$. Therefore, there is only little difference between them. With correct analysis of dynamic range, we can avoid error caused by calculation overflow or underflow. Fig. 4.45 gives the execution accuracy comparison between floating-point and 16-bit fixed-point operations.



(a)



(b)

Figure 4.45: (a) MSE and (b) SER comparison between floating-point and 16-bit fixed-point operations with 2-D interpolation using formula 2 (4 sets) with linear interpolation in the frequency domain. $V = 27$ km/h, $fdT = 0.01$.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we developed channel estimation schemes for IEEE 802.16a OFDMA downlink transmission. We proposed two kinds of interpolation methods in frequency domain which were linear interpolation and 2nd-order interpolation. Alternately, we also applied 2-D interpolation and LMS adaptive algorithm in the time domain. The combination of 2-D interpolation and linear interpolation would work efficiently. Because the linear interpolation was of less complexity and its performance on the whole was almost the same with the 2nd-order interpolation. The 2-D interpolation was more excellent than LMS adaptive algorithm in the time domain. To such pilot allocation in the IEEE 802.16a OFDMA downlink system, 2-D interpolation would be a good choice.

As to DSP implementation, for the concern of fixed-point C64x DSP, we changed the original floating-point operation to fixed-point 32-bit one. Although this did accelerate a lot, there were still limitations in using intrinsics. Therefore, replacing 32-bit fixed-point operation with 16-bit fixed-point operation was a must. Thus, we only had half the original number of bits. To make work correct, we had to be careful with the calculation to prevent from data overflow or underflow. There were three ways to accelerate the DSP execution speed: changing data types, coding style optimization, and using intrinsics. The total execution cycles of the channel estimation scheme have been reduced from 425,630 cycles to 236,871 cycles during the optimization [24]. The realtime rate also raised from 28.46% to 51.14.%. The final result showed our fixed-point 16-bit version can work as well as the floating-point version. It means the decision the bit-field is right enough for our simulation environment. But with larger the dynamic range of data values, the bit-field must varies at the same time. Besides, most of the functions reach the real time requirements and the rest almost reach it. it shows that the whole system could finish the task in time.

5.2 Future Work

We mentioned the execution cycles of the whole channel estimation scheme have been reduced to 236,871 cycles. There is still distance from the real time. The critical path may be the function *Complex_Div* since executing divider is quite time-consuming. To solve this problem, we may map the received data Y(k) directly to de-64QAM. It means we need to combine the channel estimation output with the de-64QAM block. Meanwhile, the complexity of the de-mapping must be increasing. It is trade-off. However, it is supposed to be of less complexity than the original one since the added operation in the de-mapping block is multiply. Beside, there are other improvements could be done to accelerate the execution speed.

For the Rayleigh fading channel, we only simulated with fdT=0.01, 0.02. This because the dynamic range of the real channel response varies beyond what we set with the present situation when fdT is larger. Therefore, if we want to simulate with lager fdT, we have to change the bit-field setting in the channel estimation scheme at the same time such as Q6.9, Q7.8, etc.

Bibliography

[1] IEEE Std 802.16a-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.

[2] IEEE Std 802.16a-2003, *IEEE Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Fixed Broadband Wireless Access Systems — Amendment 2: Medium Access Control Modifications and Additional Physical Layer Specifications for 2–11GHz.* New York: IEEE, April 1, 2003.

[3] M.-T. Lin, "Fixed and mobile wireless communication based on IEEE 802.16a TDD OFDMA: transmission filtering and synchronization," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2003.

[4] O. Edfors, M. Sandell, J. J. van de Beek, D. Landstrom, and F. Sjoberg, "An introduction to orthogonal frequency-dicision multiplexing," http://courses.ece.uiuc.edu/ece459/spring02/ofdmtutorial.pdf.

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

[6] O. Edfors, M. Sandell, J. J. van de Beek, S.K. Wilson, and P.O. Börjesson, "OFDM channel estimation by singular value decomposition," in *IEEE 46th Vehicular Technology Conference*, Apr. 1996, pp. 923–927.

[7] C. K. Koc and G. Chen, "Authors' reply [Computational complexity of matrix inversion]," *IEEE Trans. Aerospace Electronic Systems,* vol. 30, no 4, p. 1115. Oct. 1994.

[8] 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.

[9] M.-H. Hsieh and C.-H Wei, "Channel estimation for OFDM systems based on comb-type pilot arrangement in frequency selective fading channels," *IEEE Trans. Consumer Electron.* vol. 44, no. 1, pp. 217–225, Feb. 1998.

[10] S. G. Kang, Y. M. Ha, and E. K. Joo, "A comparative investigation on channel estimation algorithms for OFDM in mobile communications," *IEEE Trans. Broadcasting,* vol. 49, no. 2, pp. 142–149, June 2003.

[11] I.-I. Chen, "Study and Techniques of IEEE 802.16a TDD OFDMA Downlink Channel Estimation," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2004.

[12] B. Farhang-Boroujeny, *Adaptive Filters: Theory and Applications.* Wiley, 1998, pp. 139–199.

[13] S. Haykin, *Adaptive Filter Theory.* Upper Saddle River, New Jersey: Prentice Hall, 2002.

[14] H.-M. Hang, *Adaptive Signal Processing.* Course notes, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., Spring 2004.

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

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

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

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

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

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

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

[22] Y. R. Zheng and C. Xiao, "Simulation models with correct statistical properties for Rayleigh fading channels," *IEEE Trans. Commun.,* vol. 51, no. 6, pp. 920–928, June 2003.

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

[24] Y.-S. Chen, "DSP software implementation and integration of IEEE 802.16a TDD OFDMA downlink transceiver system," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2005.

作者簡歷

陳汝苓，民國七十年十一月出生於桃園縣。民國九十二年六月畢業於國立交通大學電子工程學系，並於同年九月進入國立交通大學電子研究所就讀，從事通訊系統方面相關研究。民國九十四年六月取得碩士學位，碩士論文題目為『IEEE 802.16a 分時雙工正交分頻多重進接下行導引訊號輔助式通道估測之技術與數位訊號處理器軟體實現』。研究範圍與興趣包括：通訊系統、信號處理等。