國立交通大學

電機資訊學院 電子與光電學程

碩 士 論 文

IEEE 802.16a 分時雙工正交分頻多重進接之下行同步技術研討與在數位訊號處理器上的實現

Study and DSP Implementation of IEEE 802.16a TDD OFDM Downlink Synchronization

研 究 生：蔣宗書

指導教授：林大衛　博士

中 華 民 國 九 十 三 年 七 月

IEEE 802.16a 分時雙工正交分頻多重進接之下行同步技術研討與在數位訊號處理器上的實現

Study and DSP Implementation of IEEE 802.16a TDD OFDM Downlink Synchronization

研 究 生：蔣宗書　　　　Student：Tsung-Shu Chiang

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

國 立 交 通 大 學
電機資訊學院 電子與光電學程
碩 士 論 文

A Thesis
Submitted to Degree Program of Electrical Engineering Computer Science
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Electronics and Electro-Optical Engineering
July 2004
Hsinchu, Taiwan, Republic of China

中華民國九十三年七月

IEEE 802.16a 分時雙工正交分頻多重進接之下行同步技術研討與在數位訊號處理器上的實現

研究生：蔣宗書　　　　　　　　　指導教授：林大衛 博士

國立交通大學電機資訊學院 電子與光電學程（研究所）碩士班

摘要

在論文中我們介紹一種實現 IEEE 802.16a 分時雙工正交分頻多重進接之下行同步技術的方法。下行同步技術包含 OFDM 符元(symbol) 開始時間與分數頻率偏移之同步，整數頻率偏移之同步，以及傳送資料訊框(frame)的同步。我們將同步技術以軟體方便實現在 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。

為了能方便驗証同步技術，我們也同時實現了整個 802.16a 下行傳輸的系統。為了獲得較高的 DSP 運算效率，在此系統中所有的運算皆是以定點(fixed-point)的格式來進行。在同步技術中我們以 15 個位元(bits)代表小數 1 個位元代表正負號共 16 位元的定點格式作運算。我們使用了 TI 提供的程式庫裏以組合語言做過最佳化的 FFT 程式。我們藉著使用 C6416 本身具有的指令以及將無法做軟體程序規畫(software pipeline scheduling)的迴圈展開(unroll)以達到提高執行效率的目的。在同步技術的程式做過改善之後，其執行效率獲得大幅度的提高。

論文中並針對執行效率做了分析。以軟體實現的同步技術在一顆 DSP 上執並無法達到即時運算的要求。如果我們要使同步技術的執行可以達到即時運算的要求，我們必須將同步技術分割成數個部份。用更多顆的 DSP 來實現同步技術或將一部份用 FPGA 實現。

Study and DSP Implementation of IEEE 802.16a TDD OFDMA Downlink Synchronization

Student： Tsung-Shu Chiang Advisor：Dr. David W. Lin

Degree Program of Electrical Engineering Computer Science

National Chiao Tung University

Abstract

This thesis presents an implementation method of IEEE 802.16a TDD (time division duplex) OFDMA (frequency-division multiple access) downlink (DL) synchronization techniques. The DL synchronization includes symbol time synchronization, fractional frequency offset synchronization, integer frequency offset synchronization and frame synchronization. Our implementation is software-based, employing Texas Instruments' TMS320C6416 digital signal processor (DSP) housed on Innovative Integration's Quixote cPCI card.

We implement the complete 802.16a DL system to verify the accuracy of synchronization function. The computation on this system is fixed-point for obtaining a higher execution efficiency. The data format we use in synchronization is Q.15 which is a 16 bits fixed-point data format that consists of a sign bit nad 15 fractional bits. We use the assembly-optimized FFT which is supported by TI's DSP library to obtain the high execution efficiency. We increase the execution efficiency of synchronization by using intrinsics of C6416 DSP and unrolling the disqualified loops to make the software pipeline well scheduled. The efficiency is much increased after we refine the program.

The execution efficiency of synchronization is analyzed. We find that the real time operation requirement is over the synchronization execution time. If we want the synchronization function to achieve real-time speed, we must partition the synchronization function into sub-functions and implement these functions either on more DSPs or on FPGA.

致謝

誠摯地感謝指導老師林大衛博士二年來的指導。以在職生的身份要找指導老師一開始便是件辛苦的事。但林老師並不排斥我在職生的身份而將我收入門下，給予當時在尋找指導教授之途不甚順遂的我重新燃起對學業的熱情。林老師指導的二年中對我碩士論文的完整規畫，讓我有明確的目標可以努力，這對一個在職生的求學過程有相當大的幫助。而林老師高深的學術素養，對於我在通訊領域上專業知識的增進是難以用數字來衡量。我感到非常榮幸能成為林老師的學生。在此，我要向林老師及老師的家人表達由衷的謝意。

通訊電子與訊號處理實驗室設備完善，讓我在完成碩士論文的過程中有取用不盡的資源。我要感謝實驗室中和我一起 meeting 的團隊成員俊榮以及筱晴、明哲、子瀚和盈縈，因為有大家的幫助才能使我完成這篇論文。還要感謝郁男、崑健、明偉、建統、岳賢以及全體實驗室裏的同學們給予我各方面的幫助。由於這些同學，才使得我在本實驗室中充滿快樂的回憶。

我所服務的公司加爾發半導體，在我做論文的過程一直支持我、給我最大的方便。感謝黃董事長、廖總經理和我的直屬上司呂經理以及我部門的唐先生。有他們的支持才讓我無後顧之憂。

我要感謝我的父母及家人對我的支持。最後，特別要感謝我的妻子。只有她才知道我這一路走來的艱辛以及所承受的壓力。並且在這完成學業的過程中，她一直不斷給我鼓勵與支持。沒有她的支持，我不可能完成這一切。僅將這本論文獻給我親愛的妻子。

Contents

1 Introduction **1**

2 Techniques for Downlink Synchronization **3**

2.1 Introduction to the 802.16a TDD OFDMA System 4

2.1.1 Pilot and Data Carrier Allocatin 5

2.1.2 Data Modulation and Pilot Modulation [5] 9

2.1.3 Frame Structure 10

2.2 Downlink Synchronization Techniques 12

2.2.1 Initial Synchronization 13

2.2.2 Normal Synchronization 19

2.3 Summary of Downlink Synchronization Techniques 20

3 DSP Introduction **27**

3.1 DSP Board Introduction 27

3.2 Introduction to TMS320C6416 DSP [9] 29

3.2.1 TMS320C6416 Features 29

3.2.2 Central Processing Unit 30

3.2.3 Memory Architecture 36

3.3 TI's Code Development Environment [16], [17] 36

3.4 Code Development Flow to Increase Performance [10] 39

3.4.1 Compilier Optimization Options [10] 42

4 DSP Implementation **45**

4.1 Efficiency Enhancement of DL Synchronization Code 45

4.1.1 Performance of the Original Program 45

4.1.2 Fixed-Point Number System Consideration 47

4.1.3 Code Refinement 58

4.2 Performance Discussion 73

5 Conclusion and Future work **78**

5.1 Conclusion 78

5.2 Potential Future Work 79

List of Tables

2.1 Carrier Allocation in the OFDMA DL (from [5]) 8
2.2 Complexity of Symbol Time Synchronization 14
2.3 Possible Pilot Structures in Frame Synchronization 18
2.4 System Parameters Used in Our Study 20

3.1 Execution Stage Length Description for Each Instruction Type (from [9]) 34
3.2 Functional Units and Operations Performed (from [9]) 35

4.1 Floating-Point Profile of 802.16a DL Transmitter Function Blocks 47
4.2 Floating-Point Profile of 802.16a DL Receive Function Blocks 47
4.3 Characteristics of the ETSI “Vehicular A” Channel Environment 49
4.4 Relations Between Spreed and Maximum Doppler Shift at Carrier Frequency 6 GHz and Subcarrier Spacing 5.58 kHz 50
4.5 Performance Comparision of Frequency Lock Between Floating-Point and Fixed-Point Implementation 51
4.6 Performance Comparision of Frame Lock Between Floating-Point and Fixed-Point Implementation 51
4.7 Q16.15 Bit Fields 52
4.8 Q.15 Bit Fields 52
4.9 Comparisons of Computational Complexity for Different FFT Algorithms 54
4.10 Complexity and Performance of IFFT/FFT Implementation 54
4.11 Sine/Cosine Look-Up Table 57
4.12 Fixed-Point Profile of 802.16a DL Transmitter Function Blocks 58
4.13 Fixed-Point Profile of 802.16a DL Receiver Function Blocks 58
4.14 Comparison Between FFT and Recursive DFT 59
4.15 Efficiency of Recursive DFT Implementation 60
4.16 The Execution Cycles of Pilot Correlation Loop 62
4.17 Profile of the sync Function 68
4.18 Profile of CP Correlation Function Loop Using Different Buffer Types . . 68
4.19 Multiply-Add Efficiency of CP Correlation Functions 72
4.20 Profile of Refined Code of 802.16a DL Receiver Function Blocks 77
4.21 Performances Estimation in Separate Initial and Tracking Condition . . . 77

List of Figures

2.1 OFDMA symbol time structure (from [5]). 4
2.2 DL transmitter structure (from [1]). 5
2.3 DL receiver structure (from [1]). 5
2.4 Illustration of carrier usage in OFDMA DL (from [1]). 6
2.5 Pilot allocation in the OFDMA DL (from [5]). 7
2.6 QPSK, 16-QAM and 64-QAM constellations (from [5]). 9
2.7 Pseudo random binary sequence (PRBS) generator for pilot modulation (from [5]). 10
2.8 Frame structure of the TDD OFDMA system (from [5]). 11
2.9 The stucture of the symbol time and frequency estimator from [1]. 15
2.10 DL/UL symbols identification. 16
2.11 (a) Symbol location detected in stage I, where the gray region is the useful samples which are applied FFT. (b), (c) Leftmost and rightmost ranges of correlation, respectively. (from [1]) 19
2.12 DL transmitter structure (from [1]). The gray regions indicate the implemented function in our study. 21
2.13 DL receiver structure (from [1]). The gray regions indicate the implemented fuction in our study. 21
2.14 DL synchronization process block diagram. 22
2.15 Flow chart of symbol time and fractional frequency offset synchronization. 24
2.16 Flow chart of integer frequency offset synchronization. 25
2.17 The state machine of framing synchronization. 26

3.1 Block diagram of Quixote (from [15]). 28
3.2 Block diagram of TMS320C6416 DSP (from [9]). 31
3.3 Pipeline phases of TMS320C6416 DSP (from [9]). 33
3.4 TMS320C64x CPU data path. (from [9]). 37
3.5 Code development flow for TI C6000 DSP. 41

4.1 The bursts allocation in a frame. 49
4.2 A part of assembly code for DSP_fft32x32. 54
4.3 The fixed-point data formats at the TX side. 55
4.4 The fixed-point data formats at the RX side 56
4.5 C code of recursive DFT. 59
4.6 The software pipeline information of recursive DFT. 60
4.7 Assembly code of recursive DFT. 61

4.8 C code of revised pilot correlation loop. 63
4.9 Partial assembly code of original pilot correlation loop. 64
4.10 The software pipeline information of pilot correlaton loop 65
4.11 Partial assembly code of revised pilot correlation loop. 66
4.12 The abs() function is replaced by instrinsic _abs() in C code. 67
4.13 Shift-register buffer arrangement. 68
4.14 Code of CP correlation functions using shift-register buffer and circular buffer. 70
4.15 Software pipeline information of shift-register buffer type CP correlation loop. 71
4.16 Software pipline information of circular buffer type CP correlation loop. . 72
4.17 Hand-unrolled code of circular buffer type CP correlation. 73
4.18 Software pipline information of hand-unrolled circular buffer type CP correlation loop. 74
4.19 Execution cycles of synchronization functions. 75

Chapter 1

Introduction

The IEEE-SA (Institute of Electrical and Electronics Engineers Standards Association)'s 802.16 working group is concerned with the WirelessMAN air interface for wireless metropolitan area networks. The IEEE 802.16 Task Group a developed IEEE Standard 802.16a that amends IEEE Std 802.16-2001 by enhancing the medium access control layer and providing additional physical layer specifications in support of broadband wireless access at frequencies 2–11 GHz.

We consider the DSP implementation of a IEEE802.16a downlink synchronization method. The synchronization includes symbol time synchronization, frequency offset synchronization and frame synchronization. The synchronization techinques are from [1] with some modifications. Our implementation is software-based, employing Texas Instrument's TMS320C6416 digital signal processor (DSP) housed on Innovative Integration's Quixote cPCI card. The TMS320C6416 is a fixed-point DSP with 1.67 ns instruction cycle time. It adopts the advanced VelociTI Very Long InstructionWord (VLIW) architecture that enables sustained throughput of eight instructions in parallel.

The implemented code is modified from the simulation program from [1]. We rewrite the floating-point version to the 16-bit fixed-point version and refine the code to maximize the execution performance.

The thesis is organized as follows. In chapter 2, we introduce the 802.16a downlink synchronization techniques. Chapter 3 introduces the synchronization program executing

environment, including the Quixote card and the TMS320C6416 DSP chip. Chapter 4 describes the DSP implementation and its performance. Finally, chapter 5 containes the conclusion.

Chapter 2

Techniques for Downlink Synchronization

The IEEE standard 802.16a [5] specifies the WirelessMAN air interface for wireless metropolitan area networks. There are several system modes in 802.16a: SC (single carrier), 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.

Accurate demodulation and detection of an OFDM signal requires carrier orthogonality. Variations of the carrier oscillator, sampling clock or the symbol time affect the orthogonality of the system. In this thesis, the sample clocks of the users and the base station are assumed to be identical. Then, before an OFDM receiver can demodulate the carriers, it has to perform two synchronization tasks. First, timing synchronization is needed to detect the proper frame start time. Secondly, it has to estimate and correct the carrier frequency offset of the received signal.

Before a more detailed technical overview of the IEEE 802.16a standard, we introduce some frequently used terms below. The subscriber station (SS) is usually known as the mobil station or the user. The base station (BS) is a generalized equipment set providing connectivity, management, and control of the subscriber station. The direction of transmission from the BS to the SS is called downlink (DL), and the opposite direction is

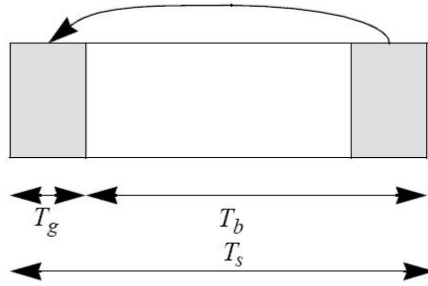


Fig. 2.1: OFDMA symbol time structure (from [5]).

uplink (UL). In this thesis, we only discuss the downlink synchronization techniques.

2.1 Introduction to the 802.16a TDD OFDMA System

The 802.16a WirelessMan-OFDMA system is based on OFDMA modulation. The inverse Fourier transform creates the OFDMA waveform. The time duration is referred to as the useful symbol time T_b. The cyclic prefix (CP) is a copy of the last T_g μs of the usful 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 shulod be supported includes 1/32, 1/16, 1/8 and 1/4. In this thesis, CP time to useful time ratio is set to 1/8. The time domain OFDMA symbol is as shown in Fig. 2.1.

In frequency domain, an OFDMA symbol is made up of carriers. There are several carrier types: data carriers, pilot carriers and null carriers. Data carriers are used for data transmission. Pilot carriers carry pilot data and are used for various estimation purposes. Null carriers do not transmission at all, they consist of the guard band and the DC carrier. The total carrier number in a DL OFDMA symbol is 2048. There are 166 pilot carriers, 1536 data carriers and 346 null carriers.

The DL system structures are shown in Figs. 2.2 and 2.3. This thesis focuses on synchronization techniques. The pilot and data carrier allocation, pilot and data modulation, and frame structure that impact the synchronization techniques are described in the

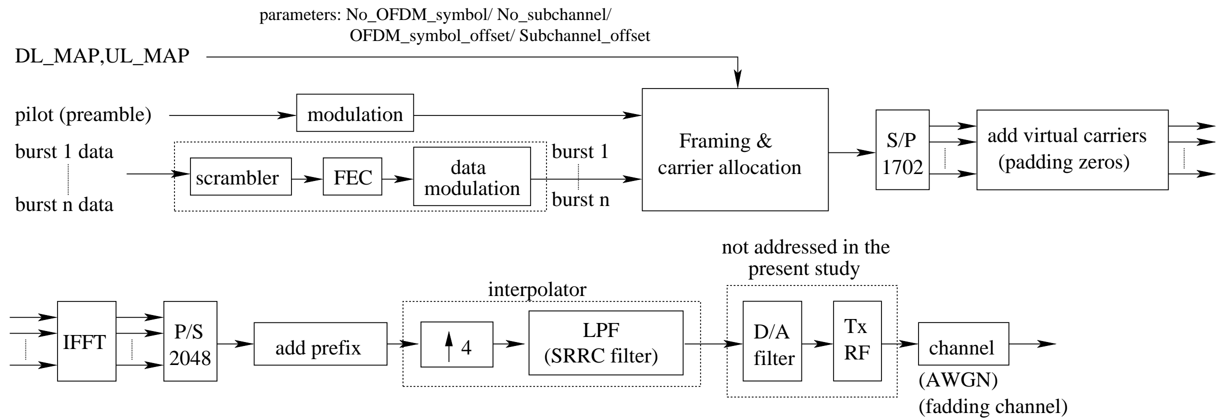


Fig. 2.2: DL transmitter structure (from [1]).

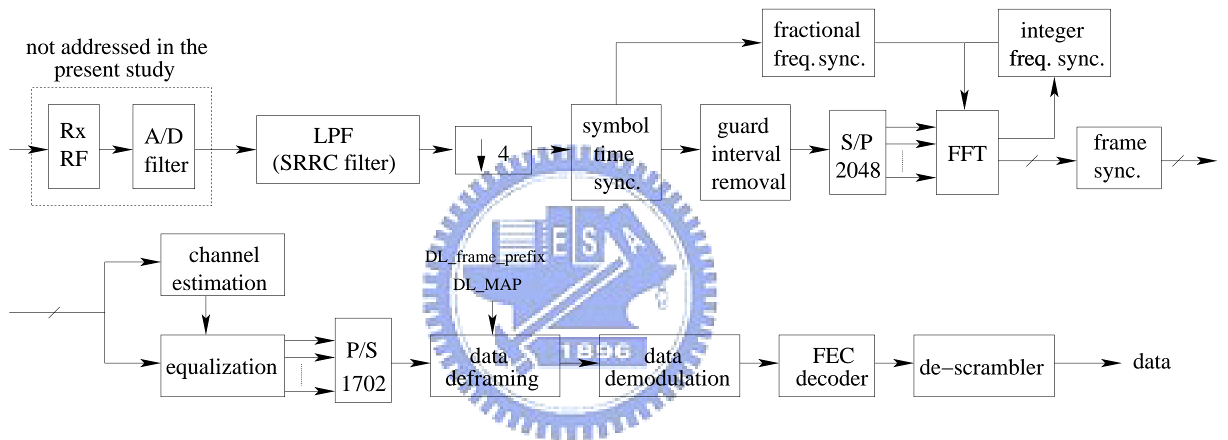


Fig. 2.3: DL receiver structure (from [1]).

following.

2.1.1 Pilot and Data Carrier Allocatin

2.1.1.1 Pilot Allocation

The carriers allocation in a DL OFDM symbol is shown in Fig. 2.4. Null carriers are allocated in the left side,the right side and the DC carrier. The pilot and data carriers are termed useful carriers for 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.

The pilot carriers include fixed-location pilots and varible-location pilots. The carrier

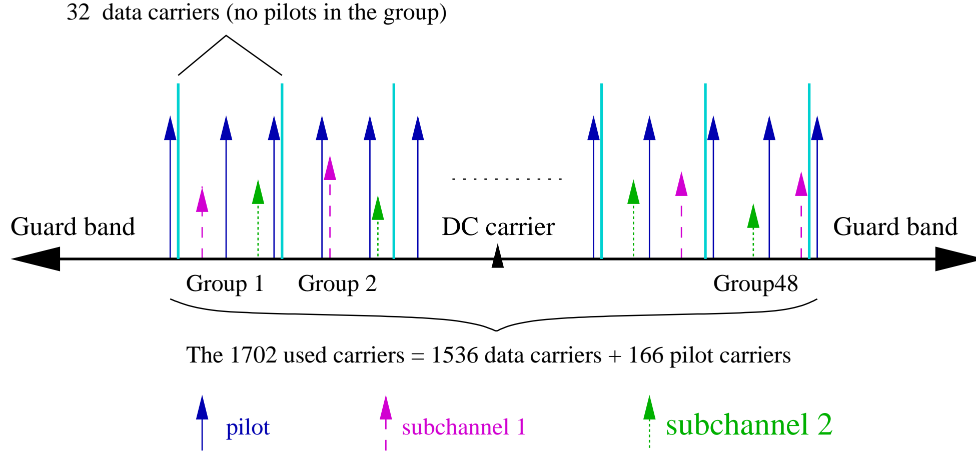


Fig. 2.4: Illustration of carrier usage in OFDMA DL (from [1]).

indices of fixed-location pilots never change. The carrier indices of the varible-location pilots vary according to the formula $varLocPilot_k = 3L + 12P_k$, where $varLocPilot_k$ is the carrier index of a varible-location pilot, L is the symbol index that cycles through the values 0,2,1,3,0,..., periodically every 4-symbol period, and $P_k = \{0, 1, 2, ..., 141\}$. The pilot carriers allocation map is shown in Fig. 2.5.

2.1.1.2 Carrier Allocation

After mapping the pilots, the remainder of the useful carriers from the data subchannels. To allocate data subchannels, partition the remaining carriers into groups of contiguous carriers. Each subchannel consists of one carrier from each of these groups. The number of the carriers in a subchannel is therefore equal to the number of groups, and it is denoted $N_{subcarriers}$. The number of carriers in a 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 patitioning into subchannels is according to the following equation called a permutation formula:

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

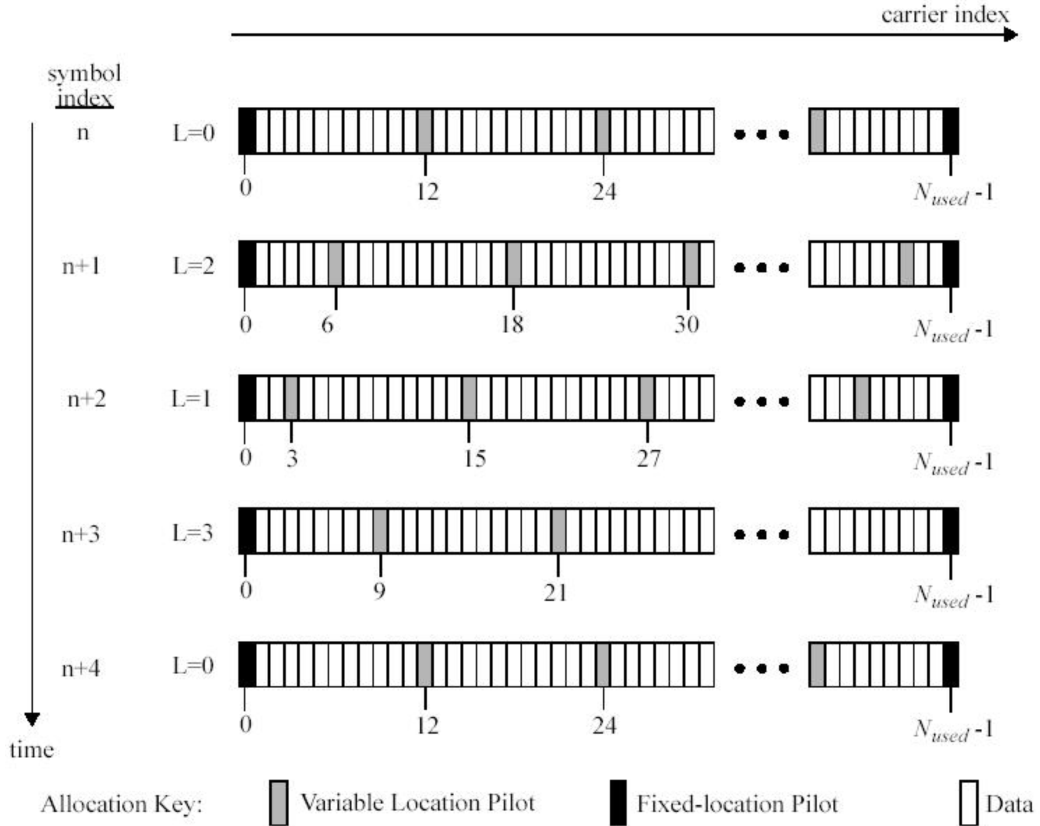


Fig. 2.5: Pilot allocation in the OFDMA DL (from [5]).

where $carrier(n, s)$ is the carrier index of carrier n in subchannel s, $s \in [0, N_subchannels-1]$ is the index of a subchannel, $n \in [0, N_subcarriers - 1]$ is the index of a subcarrier in the subchannel, $N_{subchannel}$ is the number of subchannels, $p_s[j]$ is the series obtained by rotating $\{PermutationBase_0\}$ cyclically to the left s times, $ceil[]$ is the function that rounds its argument up to the next integer, ID_{cell} is a positive integer assigned by the MAC (Medium Access Control) to identify this particular BS sector, and $X_{mod(k)}$ denotes the remainder of the quotient X/k (which is most $k - 1$). The numerical parameters are given in Table. 2.1.

Table 2.1: Carrier Allocation in the OFDMA DL (from [5])

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.

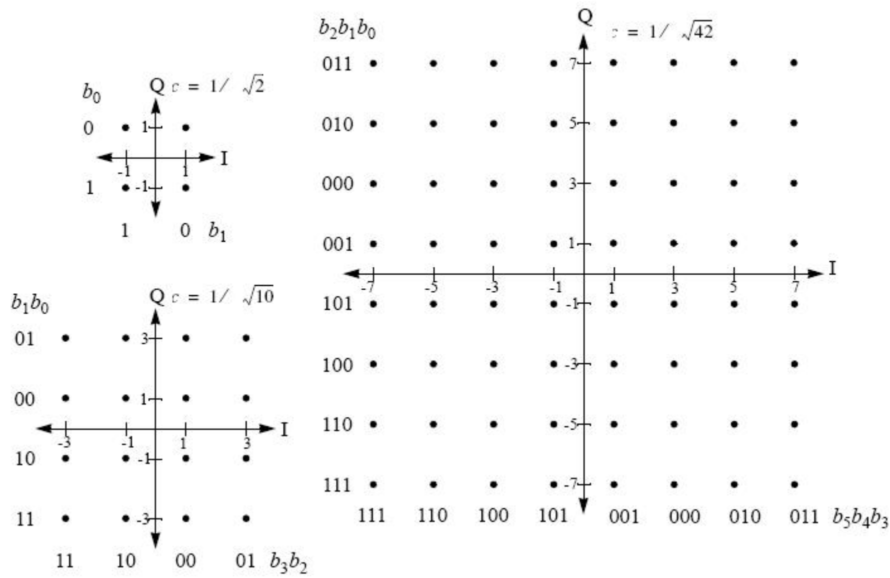


Fig. 2.6: QPSK, 16-QAM and 64-QAM constellations (from [5]).

2.1.2 Data Modulation and Pilot Modulation [5]

2.1.2.1 Data Modulation

The data modulation in 802.16a are shown in Fig. 2.6. 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.

2.1.2.2 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. The 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 Fig. 2.7 shows.

The symbols in an TDD OFDMA system DL transmission can be separated to two different types. The first three symbols are termed preamble symbols, and other symbols

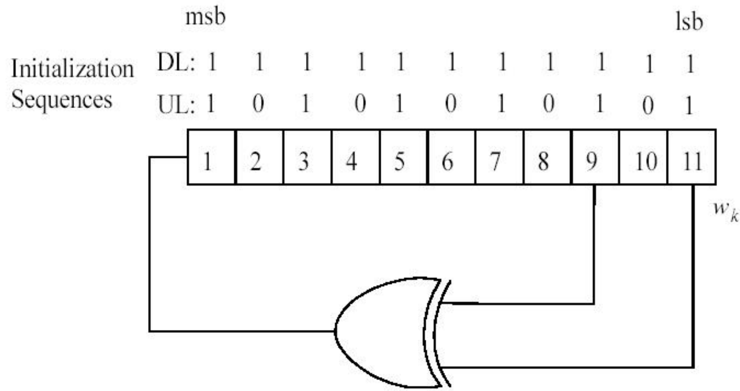


Fig. 2.7: Pseudo random binary sequence (PRBS) generator for pilot modulation (from [5]).

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

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

2.1.3 Frame Structure

The frame structure of TDD OFDMA is as shown in Fig. 2.8. The data are segmented into blocks from the view of coding, and each fit into one FEC (forward error correction) block. Each FEC block spans one OFDMA subchannel in the subchannel axis and three OFDM symbols in the time axis. A frame consists of one DL subframe and one UL subframe. The duration of a frame can be from 2 to 20 ms and is specified by the frame duration code. A subframe contains several transmission bursts, which are composed of multiples of FEC blocks. In each frame, the Tx/Rx transition gap (TTG) and Rx/Tx transition gap (RTG) shall be inserted between the downlink and uplink and at the end of each frame respectively to allow the BS and the SS to turn around. TTG and RTG shall

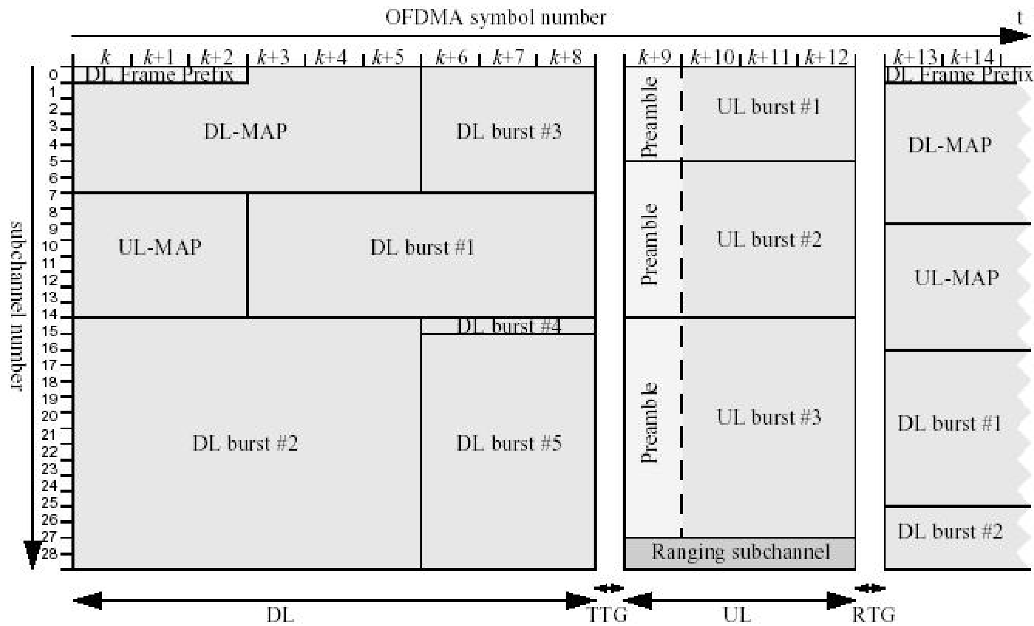


Fig. 2.8: Frame structure of the TDD OFDMA system (from [5]).

be at least 5μs and an integer multiple of four samples in duration [5].

For DL, the transmitted data from the BS should contain the control messages and the system parameters, so that the subscribers can know when and how to receive and transmit their data. The burst profile is used to define the parameters such as modulation type, FEC type, preamble length, guard times, etc. The first FEC block of each frame is the DL_Frame_Prefix that is always transmitted in the most robust burst profile QPSK-1/2. The DL_Frame_Prefix contains the parameters of the FCH (Frame Control Header) which includes the DL-MAPs, UL-MAPs and may additional DCD and UCD messages. The DL-MAP/UL-MAP messages define the access to the DL/UL information, including the burst profiles and the allocation in the subchannel and time axes of the bursts. The Downlink Channel Descriptor (DCD) and Uplink Channel Descriptor (UCD) shall be transmitted by the BS at a periodic interval to define the characteristics of downlink and uplink physical channels. The pilots of the first three OFDM symbols is the DL preamble in the sense that they indicate where the OFDMA frame starts. The number of OFDM symbols of the DL is $3N$, where N is positive integer.

2.2 Downlink Synchronization Techniques

A time offset gives rise to a phase rotation of the carriers. If the time offset is smaller than the length of the guard interval minus the length of the channel impulse response, then the orthogonality among carriers is maintained. In this case, the time offset will appear as a linear phase shift of the demodulated data symbols across the carriers but will not result in inter-symbol interference (ISI) and inter-carrier interference (ICI). For larger time offset, ISI and ICI occur. By increasing the length of the guard interval, the timing requirement can be loosened.

Frequency offset due to oscillator mismatch usually exists between the transmitter and the receiver. Each subcarrier can be assumed equally affected by a center carrier frequency spread, because the system bandwidth is small compared to the center carrier frequency. The frequency offset causes three effects : reducing the amplitude of FFT output, introducing ICI from other carriers, and introducing a common phase rotation of the subcarriers [3]. The frequency offset can be separated to an integer part and a fractional part. The former gives frequency offset in integer times carrier spacing, and the latter gives frequency offset in fractional number times carrier spacing. The integer frequency offset results in the entire spectrum of an OFDMA signal be cyclicly shifted, and no ICI [4].

There are two DL synchronization conditions: initial synchronization and normal synchronization. In the beginning when one subscriber wants to join the transmission network, it has no idea about the timing of the network and frequency offset with the base station. When the SS receives DL OFDMA symbol, the OFDMA symbol start time should be found, and the frequency offset between SS and BS should be estimated and compensated. According to 802.16a, the center frequency of the SS shall be synchronized to the BS with a tolerance of maximum 2% of the inter-carrier spacing. The frame start time should be found after symbol time and frequency offset synchronization are finished. After the frame synchronization, SS can get the frame information and use it to enter the

normal synchronization condition [1].

2.2.1 Initial Synchronization

The scheme that we use divides initial synchronization into four stages [1], which are symbol time synchronization, fractional frequency synchronization, integer frequency synchronization and frame synchronization.

2.2.1.1 Stage I: Symbol Time Synchronization

The research in [1] suggests estimating symbol time by using the cyclic prefix. Two algorithms are mentioned in that thesis: ML estimation and CP correlation. ML estimation algorithm is proposed in [2], using the maximum likelihood criterion to estimate time and frequency offsets. Under the assumption that received samples are jointly Gaussian, symbol time offset $\hat{\theta}$ is given by

$$\hat{\theta} = \arg\max \left\{ |\Gamma(\theta)| - \rho\Phi(\theta) \right\}, \tag{2.2}$$

where

$$\Gamma(\theta) = \sum_{k=\theta}^{\theta+L-1} r(k)r^*(k+N), \tag{2.3}$$

$$\Phi(\theta) = \frac{1}{2}\sum_{k=\theta}^{\theta+L-1} |r(k)|^2 + |r(k+N)|^2, \tag{2.4}$$

and $\rho = \frac{SNR}{SNR+1}$ with SNR being signal to noise ratio. It is a one-shot estimator in the sense that the estimates are based on the observation of one OFDM symbol. To roduce the complexity, CP correlation algorithm [1] suggests using only the correlation part to estimate the symbol time. As the samples of different OFDM symbols are uncorrelated, the peak of the sliding sum of $r(k)r^*(k+N)$ would occur when the samples $r(\theta), \cdots, r(\theta+N+L-1)$ are all within the same OFDM symbol. Then, the symbol time offset estimator becomes

$$\hat{\theta} = \arg\max \left| \sum_{k=\theta}^{\theta+L-1} r(k)r^*(k+N) \right|. \tag{2.5}$$

The complexities of ML estimation and CP correlation algorithm are shown in Table 2.2. Notes that after the CP correlation is computed at sample time θ by formula 2.3,

Table 2.2: Complexity of Symbol Time Synchronization

	Multipications(complex)	Additions(complex)	Other Functions
$\Gamma(\theta)$	4350	4349	
$\Phi(\theta)$	8700	8452	1 absolute value
ρ	6913	6909	1 division 1 square root 1 absolute value

the CP correlation at sample time θ+1 is simplified as

$$\begin{aligned}\Gamma(\theta+1) &= \sum_{k=\theta+1}^{\theta+L} r(k)r^*(k+N), \\ &= \Gamma(\theta) - r(k)r^*(k+N) + r(\theta+L)r^*(\theta+L+N). \end{aligned} \quad (2.6)$$

The CP correlation algorithm only calculates $\Gamma(\theta)$, and ML estimation algorithm calculates all the entries listed. The research in [1] shows that although the performance of ML estimator algorithm is better than that of CP correlation algorithm, neither algorithm can estimate the exact symbol time at 100% accuracy. To estimate the exact symbol time, both algorithms should be assisted by some other auxiliary operations. Here pilot correlation is used as the auxiliary operation to estimate the symbol time, which is performed in stage IV. The complexity of ML estimaiton is much more than CP correlation algorithm, but the benefit is not as much. We use the CP correlation to estimate the symbol time in this stage.

2.2.1.2 Stage II: Fractional Frequency Synchronization

In our algorithm, integer frequency offset is estimated in the post-FFT stages. Fractional frequency offset is estimated in this stage.

Based on the frequency part of the joint ML estimator in [2] and [8], the fractional frequency offset $\hat{\epsilon}$ is given by

$$\hat{\epsilon} = \frac{-1}{2\pi}\angle\Gamma(\hat{\theta}),$$

as shown in Fig. 2.9. It is easy to understand why ϵ can be estimated by this method. The frequency offset ϵ results in a sinusoidal wave in the time domain, and thus the received

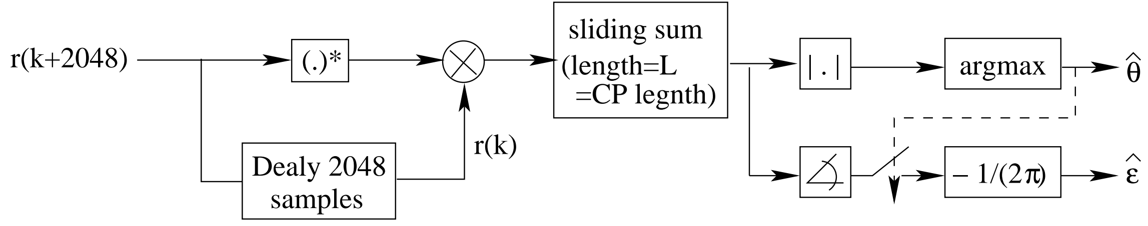


Fig. 2.9: The stucture of the symbol time and frequency estimator from [1].

samples are multiplied by $\left\{1, e^{j\frac{2\pi\epsilon}{N}}, e^{j\frac{2\pi\epsilon 2}{N}}, ...\right\}$. In AWGN channel, the received sample in the guard time is

$$r(k) = s(k)e^{j\frac{2\pi\epsilon k}{N}} + n(k),$$

and the sample in the last part of the useful time is

$$r(k+N) = s(k+N)e^{j\frac{2\pi\epsilon(k+N)}{N}} + n(k+N),$$

where $s(k)$ is the transmitted signal, N is the FFT size, and $n(k)$ is the noise. Then the multiplication of $r(k)$ and $r^*(k+N)$ becomes

$$r(k)r^*(k+N) = s(k)s^*(k+N)e^{-j2\pi\epsilon} + \text{noise}.$$

Note that $e^{-j2\pi\epsilon}$ is the common factor of all the sample pairs with $r(k)$ in the guard interval. It makes sense that the sum of these sample pairs would reduce the noise effect. The frequency offset ϵ can be given by the angle part of the sum of $r(k)r^*(k+N)$ taken at the symbol start position. Note that the phase rotation of integer frequency offset is integer times of 2π. Thus this estimator is merely able to detect the fractional frequency offset.

The structure of this estimator including stages I and II is shown in Fig. 2.9.

2.2.1.3 Stage III: Integer Frequency Synchronization

After the fractional frequency synchronization, we use the guard bands information to estimate integer frequency offset [1]. To begin, an SS shuld check whether the received OFDM symbol is from BS rather than from another SS. In 802.16a [5], the definition

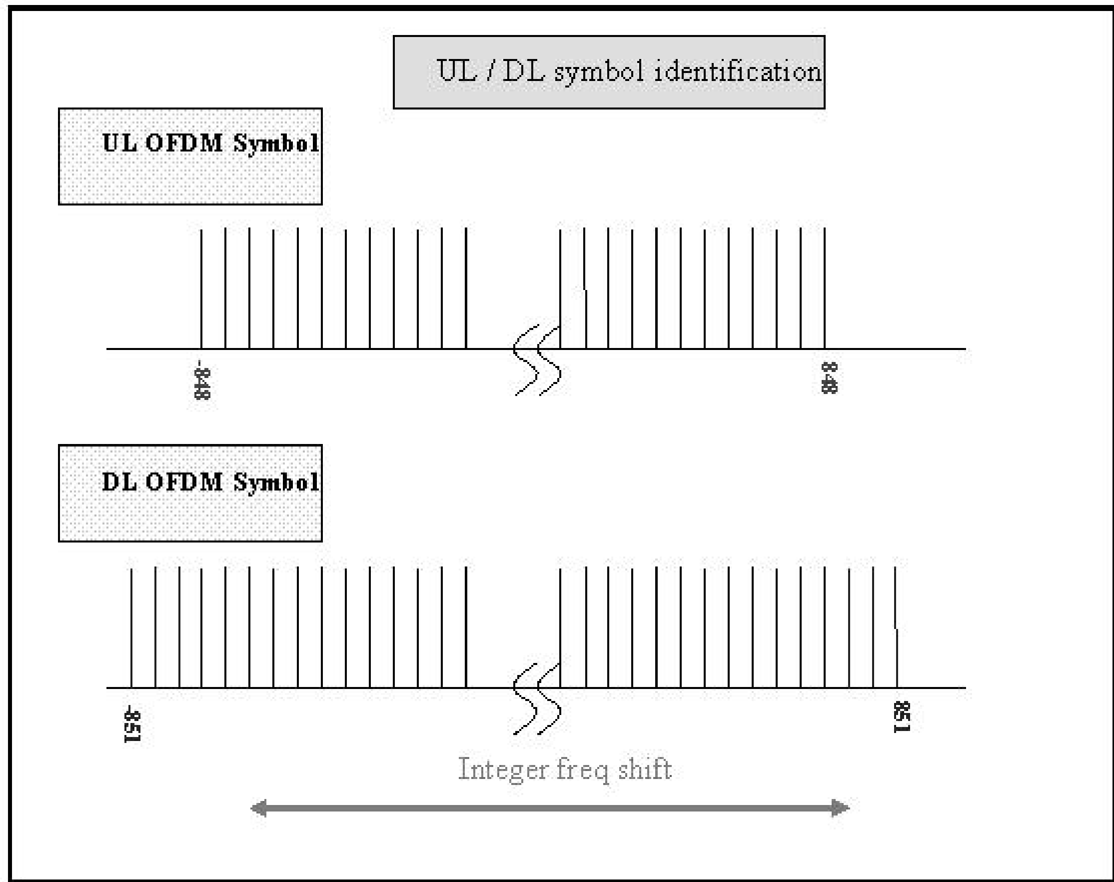


Fig. 2.10: DL/UL symbols identification.

of the guard bands and pilots are different for DL and UL. The indices of the DL guard carriers are from -1024 to -852 and from 852 to 1023, while the UL are from -1024 to -849 and from 849 to 1023. Because the symbol from another SS has the limitation that its frequency offset to the BS must not be over 2% carrier spacing, if the OFDMA symbol is from another SS, the magnitude in carrier indices $\{-851, -850, -849, 849, 850, 851\}$ must be small. A threshold can be set that if any of the carriers $\{-849, -850, -851, 849, 850, 851\}$ is larger than the threshold, the SS will regard the symbol as a DL symbol, as shown in Fig. 2.10.

For the DL, the standard defines the carriers -851 and 851 as fixed location pilots which are modulated to $\pm\frac{4}{3}$ in amplitude. If there is no integer frequency offset, the FFT outputs of all the guard carriers will be small. So, all the guard carriers are checked to

see if any of them exceeds the threshold. The checking direction is from 1023 to 852, and then from -1024 to -852. If carrier k is detected to be larger than the threshold in the checking procedure, the ±851st fixed pilots are supposed to shift $k-851$ carrier spacings due to the frequency offset. Thus the checking is stopped and the frequency is corrected by $k - 851$ carrier spacings. The checking and correction take turns until all the guard carriers are checked to be smaller than the threshold.

In fading channels, ICI may cause serious distortion. Thus, if the ±851st pilots are distorted to be less than the threshold, the frequency offset will not be detected by the previous method. An additional check is added to see whether both of the ±851st pilot carriers are larger than the threshold. After these three checks, the integer synchronization finishes.

2.2.1.4 Stage IV: Frame Synchronization

By stage I, the OFDMA symbol start time can be ruoghly estimated, but the SS has to know exactly where the frame starts. The frame start time estimation suggested in [1] uses the pilot correlation method. In the 802.16a standard [5], the varible location pilots change their location from symbol to symbol depending on symbol index L. The modulation of pilots is decided by the PRBS generator, and the initialization vector of the PRBS generator is different in preamble symbol generation from in non-preamble symbol generation. Therefore, there are 7 possible kinds of pilot sructures as shown in Table 2.3. If the received symbol has the same pilot locations and the same initial vector of modulation PRBS with the reference data, the correlation of them will be larger than the other 6 cases. A frame is determined to start if there are three successive DL symbols with the maximum correlation corresponding to the preamble.

The simulation result of [1] shows that the accuracy of symbol time estimation is not enough. There is a serious problem by using the post-FFT pilots or preamble if the symbol time synchronization in stage I does not detect the correct location of the symbol, for then there will be a time offset d. After FFT, the time offset causes phase shift across

Table 2.3: Possible Pilot Structures in Frame Synchronization

DL preamble	DL normal symbol
$L = 0$, $PRBS = 01010101010$	$L = 0$, $PRBS = 11111111111$
$L = 2$, $PRBS = 01010101010$	$L = 2$, $PRBS = 11111111111$
$L = 1$, $PRBS = 01010101010$	$L = 1$, $PRBS = 11111111111$
	$L = 3$, $PRBS = 11111111111$

the carriers by $e^{j\frac{2\pi kd}{N}}$, where k is the carrier index. This phase shift affects the correlation of the received pilots and the reference data. Moreover, if the detected symbol start time is later than the actual time, ISI and ICI may occur. Whether the maximum correlation of the 7 cases indicates the true frame start becomes doubtful.

To solve this problem, a more robust symbol time should be estimated. If there was a time offset, the useful time would be shifted and the pilots correlation would be smaller. The simulation of [1] shows that the symbol time estimation error in stage I has high probability to be smaller than 30 samples. Assume that the time offset may be from -32 to 32 sample times. Fig. 2.11(a) shows the symbol start location detected in stage I, where the gray region is the corresponding useful samples which are taken FFT. We apply the FFT to the gray region from -32 to 32 samples in offset, as shown in Fig. 2.11(b) and (c) [1]. After observing the correlation for 65 sample times, the location with peak correlation is assumed to be the real symbol start time. The maximum correlation of the 7 cases is then robust enough to be used. In order to reduce the complexity of FFT, the conventional FFT is only applied to location -32. When a new data value is received, the FFT may be computed successively as

$$X_n(k) = \left[X_{n-1}(k) - x_{n-N} + x_n\right] e^{j\frac{2\pi k}{N}} \tag{2.7}$$

where N is the FFT size, k is the carrier index, n is sample number, and x_n is the new incoming sample.

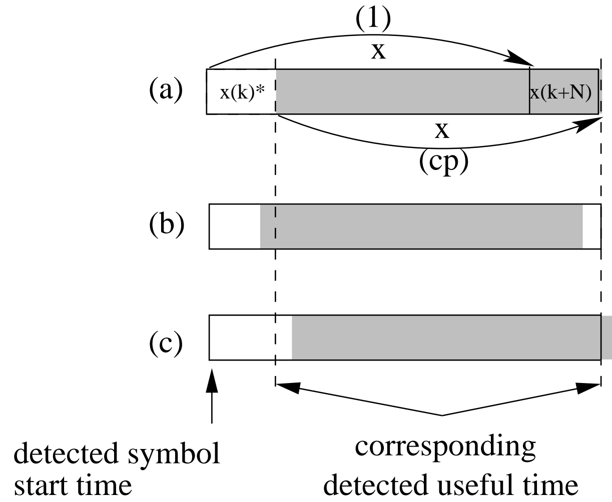


Fig. 2.11: (a) Symbol location detected in stage I, where the gray region is the useful samples which are applied FFT. (b), (c) Leftmost and rightmost ranges of correlation, respectively. (from [1])

2.2.2 Normal Synchronization

After finishing initial synchronization, the SS can find the frame duration from frame duration code in the MAPs. The timing synchronization stage should still be used to track the exact symbol time, because the received symbol time may shift with time due to channel variation. The CP correlation can estimate the rough symbol time. In normal synchronization condition, pilot correlation helps to find the robust symbol time. The simulation result in [1] shows that when the Doppler spread is small, the standard deviation of time synchronization error is about 3–4. If the channel is compensated, we can reduce the range of possible timing offset that estimated from CP correlation to simplify the complexity. The normal synchronization condition should be started after the channel is compensated. In our system, the channel estimator is performed after the synchronization. We assume that the channel is compensated before the frame is synchronized. In this case, the timing synchronization error in CP correlation stage is assumed to be less than 5 sample time. Just as the pilot correlation step in frame synchronization stage, we should take FFT in the range from 5 sample time before the estimated symbol time to 5 sample

Table 2.4: System Parameters Used in Our Study

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

time after the estimated symbol time. The FFT output is used to do the pilot correlation with 7 symbol types listed in Table 2.3. We can track the exact symbol time and check the symbol types. If the symbol type is not as expected, the initial synchronization should be re-done.

Besides, the frequency has been synchronized to the BS during normal operation. According to 802.16a, the SS shall track the frequency changes and shall defer any transmission if synchronization is lost. The small frequency changes can be tracked by the frequency part of the joint ML estimation (the same as stage II of initial synchronization). These changes are averaged for a period of time and then compensated, so the frequency offset under the tracking mode will be smaller than the initial frequency synchronization. If by any chance a larger frequency variation occurs, we may detect it by monitoring the received guard carriers and then try to correct it.

2.3 Summary of Downlink Synchronization Techniques

The system parameters employed in this study are shown in 2.4. Our goal in this thesis is to do software implementation of the synchronization techniques on DSPs. The implemented transmitter and receiver components are as indicated in Fig. 2.12 and 2.13. The gray regions are implemented blocks, and the others such as FEC, channel estimation and equalization are not implemented in this study.

Recall from 2.2 that the initial DL synchronization contains 4 stages, which are sym-

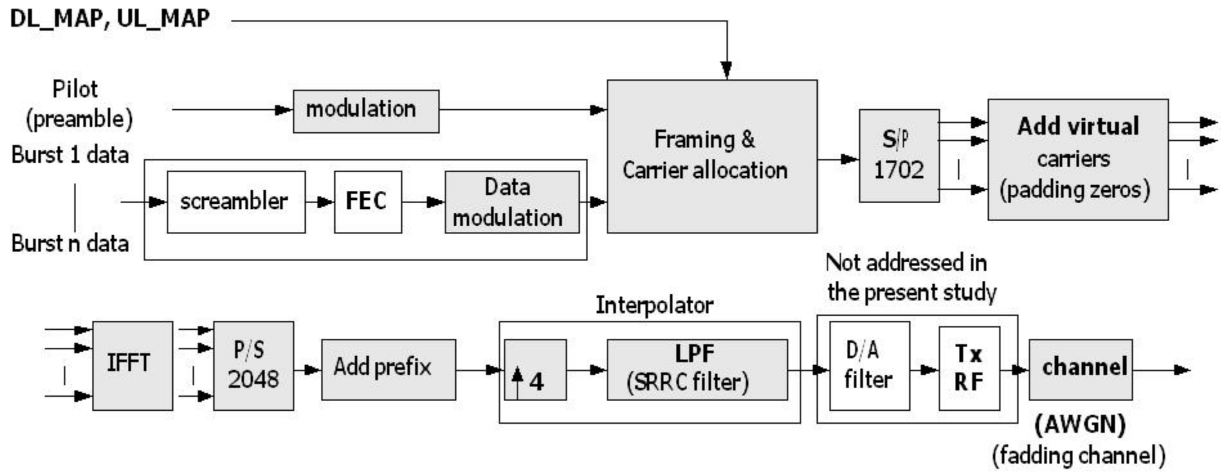


Fig. 2.12: DL transmitter structure (from [1]). The gray regions indicate the implemented function in our study.

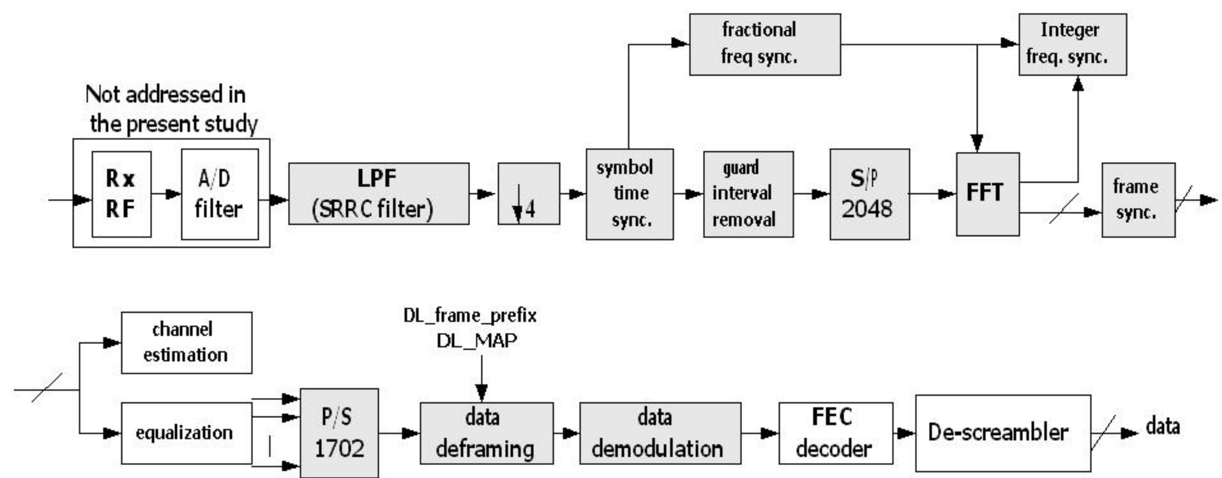


Fig. 2.13: DL receiver structure (from [1]). The gray regions indicate the implemented fuction in our study.

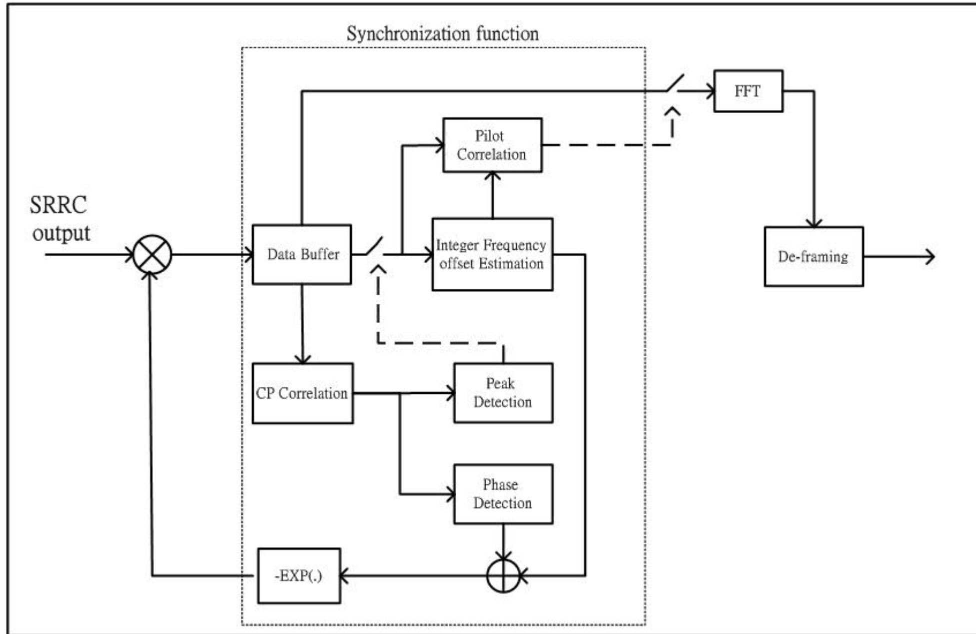


Fig. 2.14: DL synchronization process block diagram.

bol time synchronization, fractional frequency offset synchronization, integer frequency offset synchronization , and frame synchronization. At beginning, the CP correlator output detects an local peak value. The phase of correlator output peak is the fractional frequency offset. As shown in Fig. 2.14, use this peak location to perform the integer frequency estimaion. The integer frequency offset estimator estimates the integer frequency offset. Adding integer and fractional frequency offset and using this result to compensate the input data. After some iterations, the integer frequency offset will be fixed, than start to find the frame start by using pilot correlation.

The flow chart of symbol time and fractional frequency offset estimations are shown in Fig. 2.15. The CP_max records the maximum value of CP correlation, CP_corre_location records the start time of a symbol that estimated in CP correlation stage, Freq_Off records the estimated fractional frequency offset. A new correlation value is computed and then compared with CP_Max whenever a new sampled data is received and shifted into synchronization buffer. If the new correlation value is larger than CP_Max, we replace the value of CP_Max by the news correlation value, CP_corre_location by current location,

and Freq_Off by the phase of correlation value. If the correlation value is not larger than the maximum vlaue, we compute the next CP correlation value by receiving new sampled data without modify the content of these varibles that record the CP correlation information. If all the next 256 successive CP correlation values are not larger than CP_Max, the current CP_Corre_location is the estimated symbol time and the current Freq_Off is the estimated fractional frequency offset.

Integer frequency offset estimation is perfomed after FFT. The CP correlation peak location is used in this stage to be the symbol start time. The flow chart of integer frequency offset estimation are shown in Fig. 2.16. The lock condition is achieved after the spectrum offset of the received symbol is checked zero.

Frame synchronization is started after frequency offset is compensated. The type of every received symbol is identified by pilot correlation. In the beginning of frame synchronization, the preamble and $L = 0$ symbol is waited. This is the first symbol of a frame. The state machine is started when the first preamble symbol is received and goes to the next state when the predicted symbol is received. The normal synchronization codition is achieved when the third preamble symbol is received. If the received symobl is not the predicted symbol, the synchronization lost, and then the frame synchronization is re-started. Fig. 2.17 shows the state machine for frame synchronization.

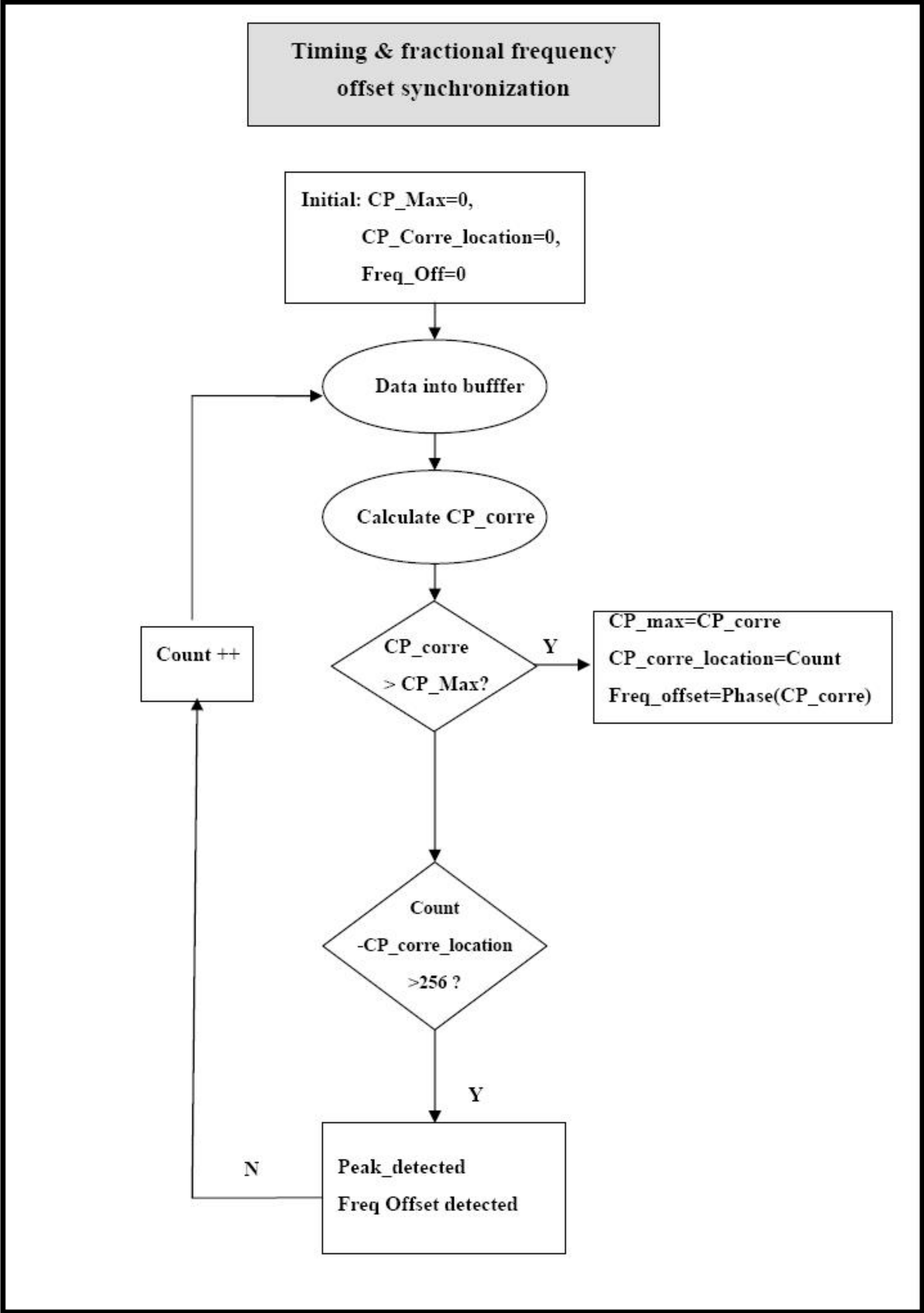


Fig. 2.15: Flow chart of symbol time and fractional frequency offset synchronization.

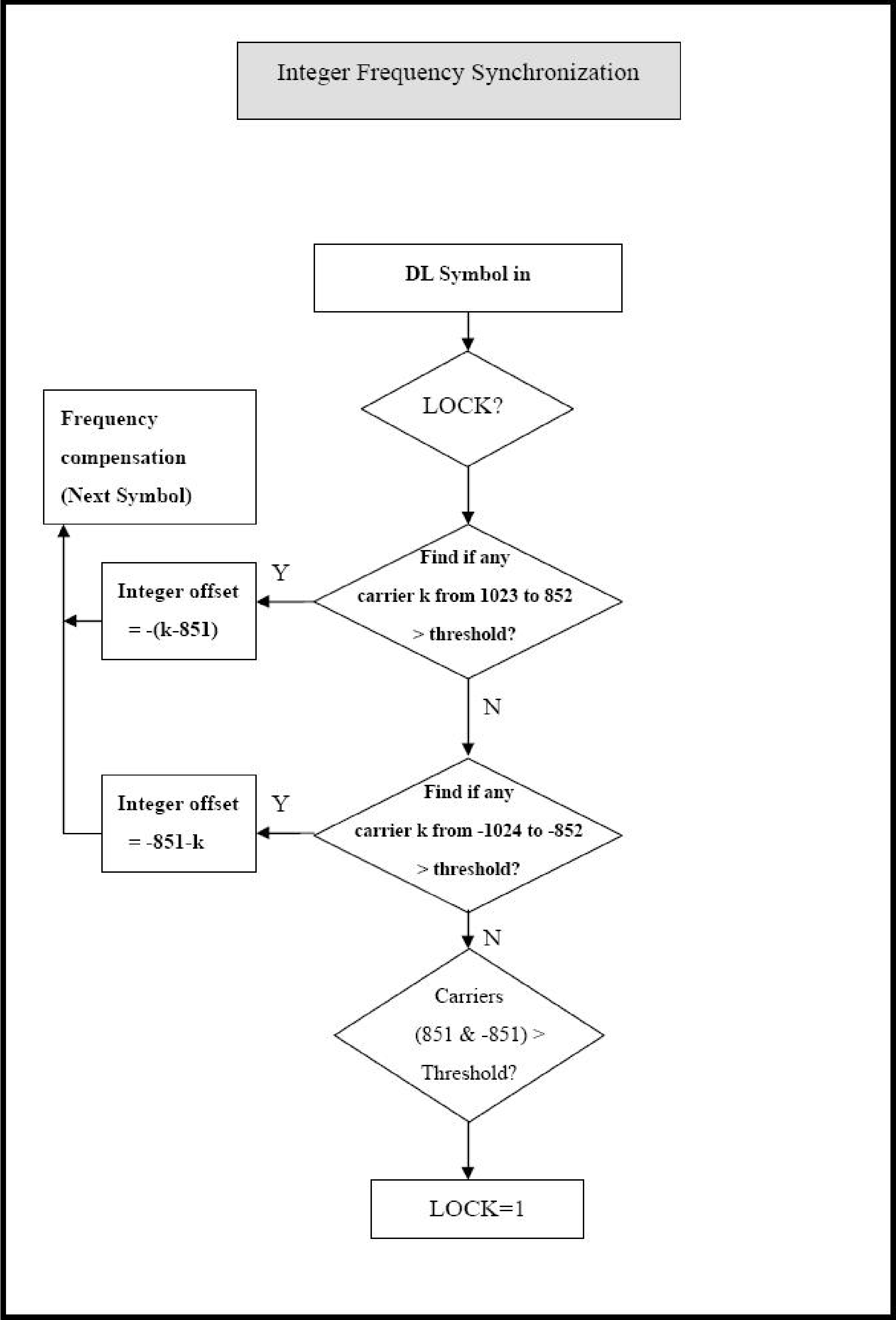


Fig. 2.16: Flow chart of integer frequency offset synchronization.

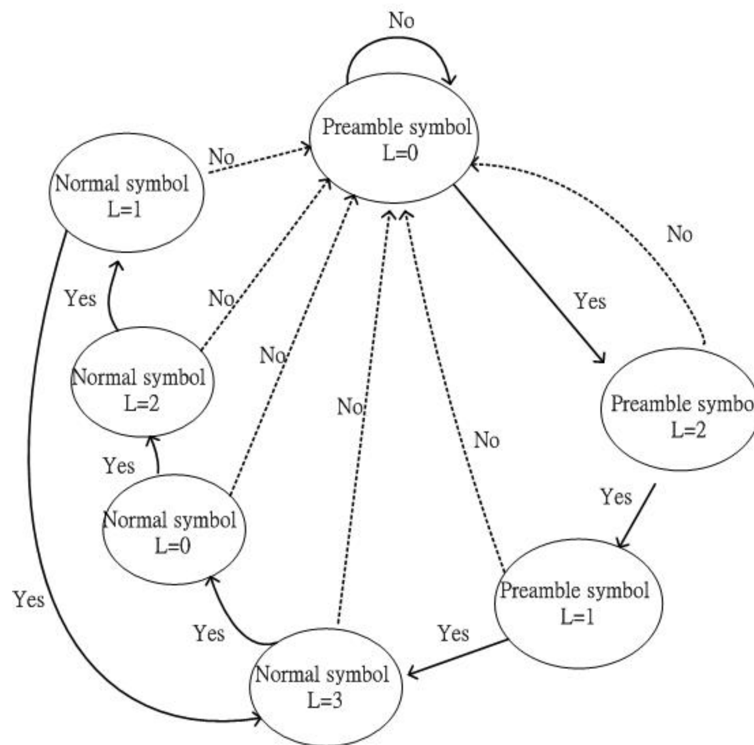


Fig. 2.17: The state machine of framing synchronization.

Chapter 3

DSP Introduction

The 802.16a DL synchronization techinques are implemented on DSP platform. The platform we use is a DSP card made by Innovative Integration, the Quixote. This chapter introduces the Quixote PC-plugin card and the DSP which is Texas Instruments' TMS320C6416 on this card. Our discussion will concentrate more on the DSP chip because of our implementation is pure software on the DSP.

3.1 DSP Board Introduction

Quixote is Innovative Integration's Velocia-family baseboard for various applications requiring high-speed computation. Fig. 3.1 shows a block diagram of the Quixote board. It combines a 600 MHz 32-bit fixed-point DSP, an FPGA (Virtex-II) analog acquisition, and system-level peripherals. The TI C6416 DSP operating at 600 MHz offers a processing power of 4800 MIPS.

The Virtex-II FPGA includes 18x18 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 process memory that are invaluable in efficient logic design.

The Quixote card has a 32MB 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 6416 cache controller is effective to over 85% of infinite on-chip memory performance for most DSP applications. A flash EEPROM allows configuration data to be saved and a 512 byte serial EEPROM memory allows storage of converter correction coefficients which is used by the embedded Viterbi and turbo decoder .

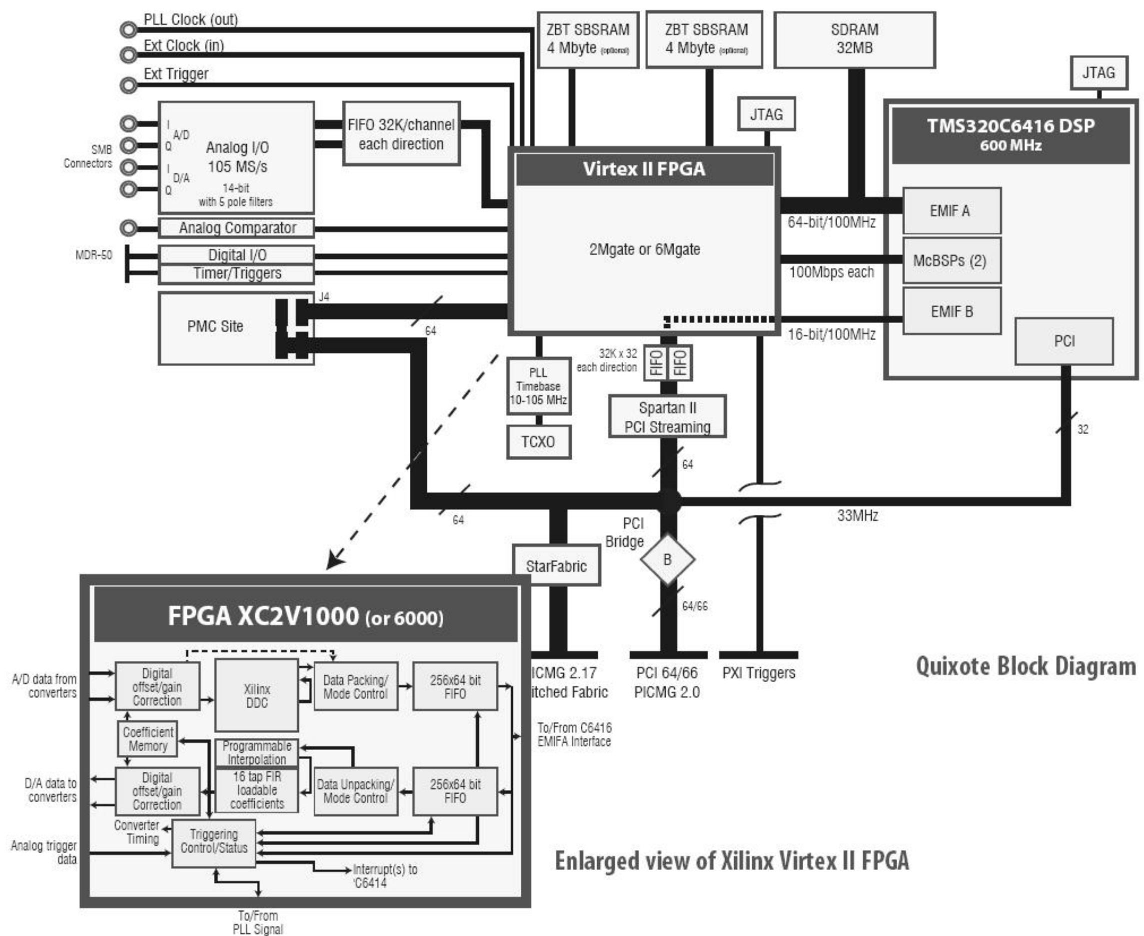


Fig. 3.1: Block diagram of Quixote (from [15]).

3.2 Introduction to TMS320C6416 DSP [9]

3.2.1 TMS320C6416 Features

The TMS320C64x DSPs are the highest-performance fixed-point DSP generation on the TMS320C6000 DSP platform. 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 co-processors, Viterbi Decoder Coprocessor (VCP) and Turbo Decoder Coprocessor (TCP) that significantly speed up channel-decoding operations on-chip.

The C64x core CPU consists of 64 general-purpose 32-bits registers and 8 function units. These 8 function units contain two multipliers and six ALUs. Features of C6000 device includes :

- 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 C64x additional features 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.

3.2.2 Central Processing Unit

The block diagram of C6416 DSP is shown in Fig. 3.2. The DSP contains:

- Program fetch unit.
- Instruction dispatch unit.
- Instruction decode unit.

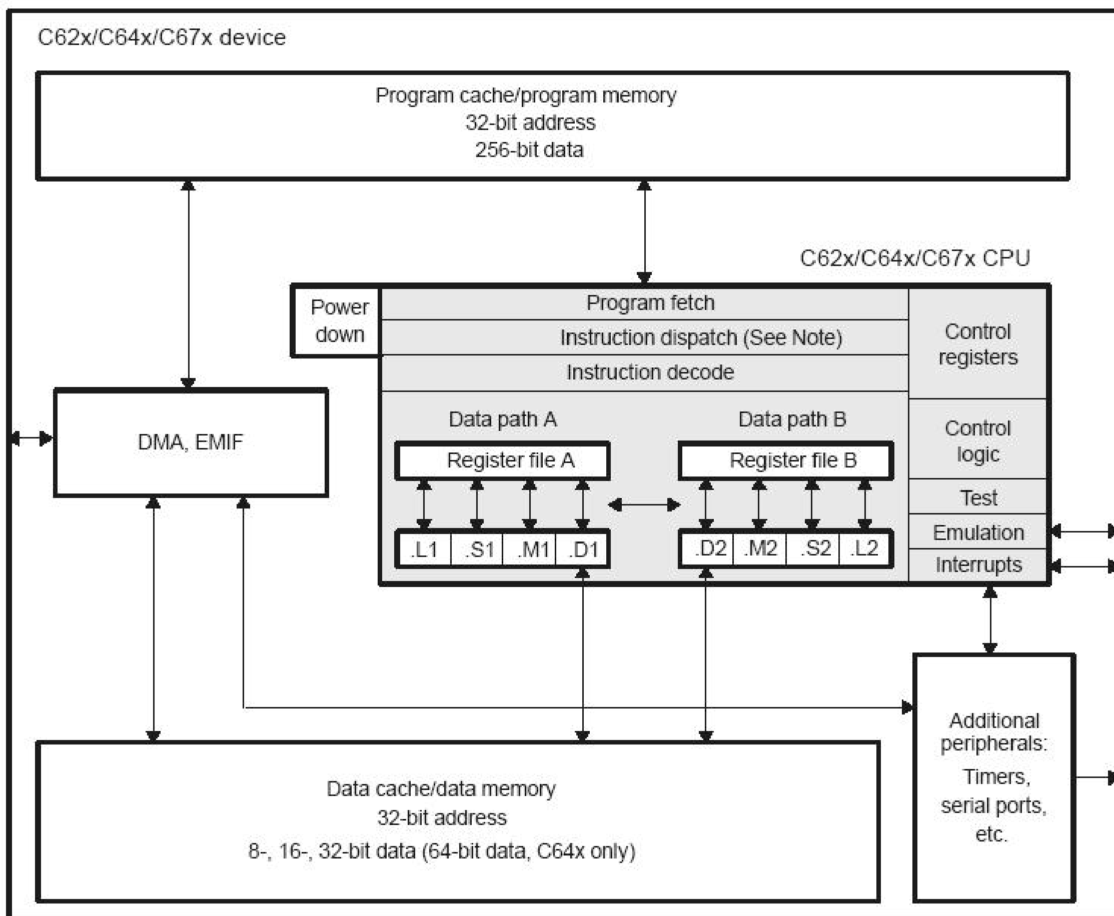


Fig. 3.2: Block diagram of TMS320C6416 DSP (from [9]).

- Two data paths, each with four functional units.
- 64 32-bit registers.
- Control registers.
- 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.

The pipeline phases are divided into three stages:

- Fetch.
- Decode.
- Execute.

All instructions in the C62x/C64x instruction set flow through the fetch, decode, and execute stages of the pipeline. The fetch stage of the pipeline has four phases for all instructions, and the decode stage has two phases for all instructions. The execute stage of the pipeline requires a varying number of phases, depending on the type of instruction. The stages of the C62x/C64x pipeline are shown in Fig. 3.3.

Reference [9] contains the detailed fetch and decode phases information. 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

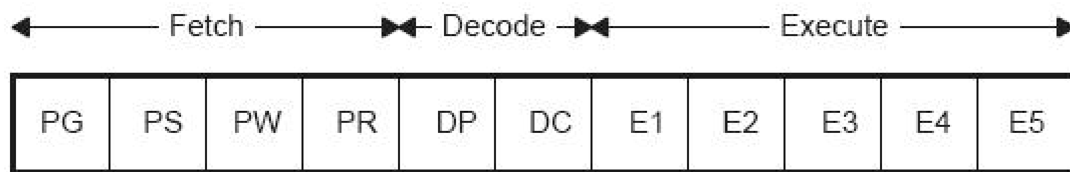


Fig. 3.3: Pipeline phases of TMS320C6416 DSP (from [9]).

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.

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.

Besided 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 8x8 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 (Refer to Fig. 3.4). All units ending in 1 (for example, .L1) write to register file A, and all units ending in 2 write to register file B. Each functional unit has two 32-bit read ports for source operands src1 and src2. Four units (.L1, .L2, .S1, and .S2) have an extra 8-bit-wide port for 40-bit long writes, as well as an 8-bit input for 40-bit long reads.

Table 3.1: Execution Stage Length Description for Each Instruction Type (from [9])

		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‡

Table 3.2: Functional Units and Operations Performed (from [9])

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

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.

3.2.3 Memory Architecture

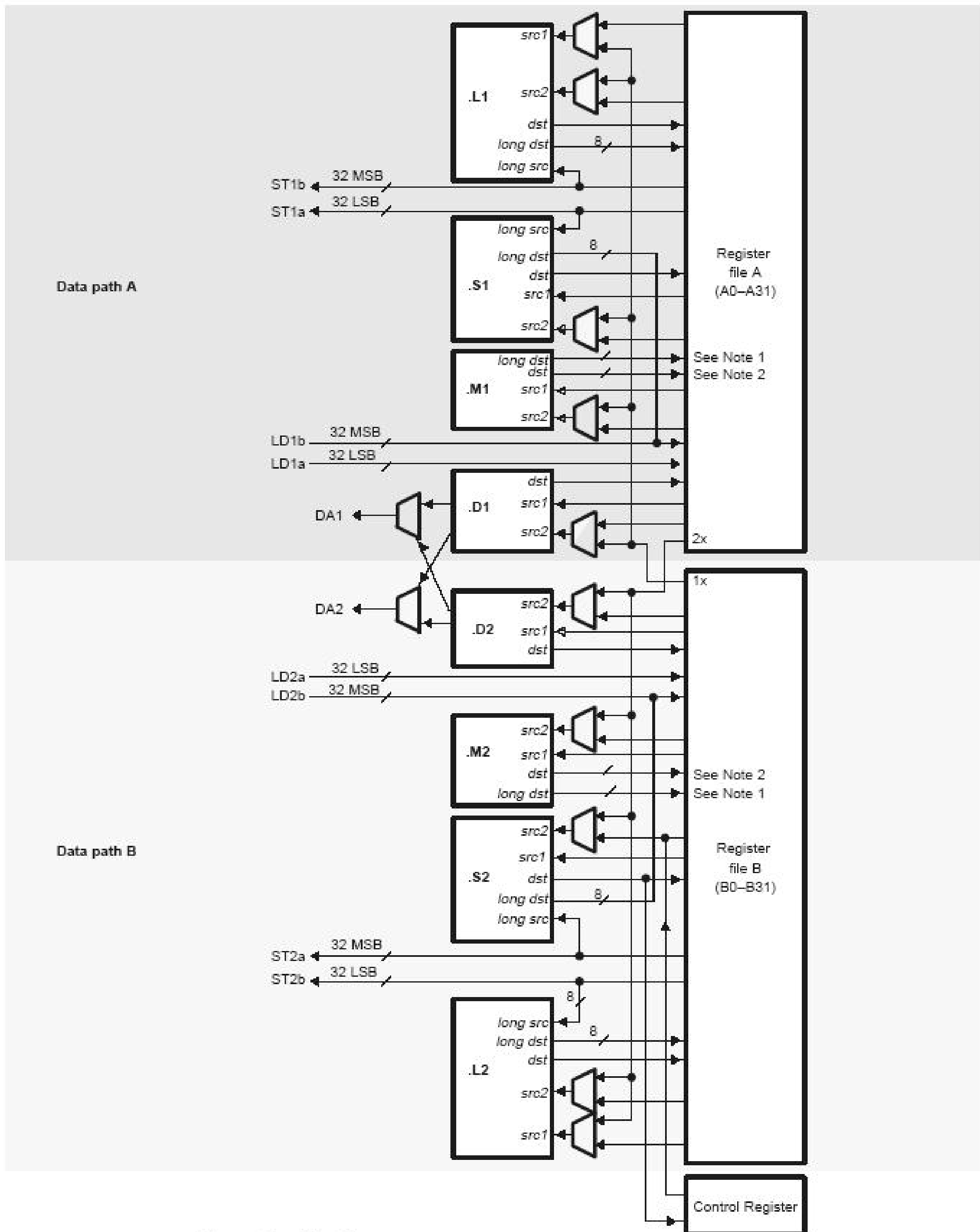
The C64x has a 32-bit, byte-addressable address space. Internal (on-chip) memory is organized in separate data and program spaces. When off-chip memory is used, these spaces are unified on most devices to a single memory space via the external memory interface (EMIF). The C62x/C67x have two 32-bit internal ports to access internal data memory. The C64x has two 64-bit internal ports to access internal data memory. The C62x/C64x/C67x have a single internal port to access internal program memory, with an instruction-fetch width of 256 bits.

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

- On-chip RAM, up to 7M bits.
- Program cache.
- Two-level caches.
- 32-bit external memory interface supports SDRAM, SBSRAM, SRAM, and other asynchronous memories. In our system, the external memory used by DSP is a 32MB SDRAM.

3.3 TI's Code Development Environment [16], [17]

TI supports a useful GUI development 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



Notes for .M unit:

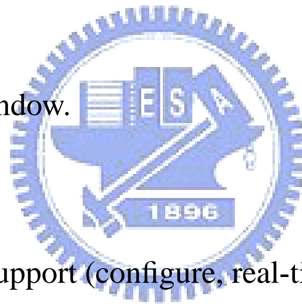
1. *long dst* is 32 MSB
2. *dst* is 32 LSB

Fig. 3.4: TMS320C64x CPU data path. (from [9]).

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 understand code performance.

CCS also delivers foundation software consisting of:

- DSP/BIOS kernel for the TMS320C6000 DSPs.
  - Pre-emptive multi-threading
  - Interthread communication
  - Interupt 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.

- DSP libraries for optimum DSP functionality. The DSP Library includes many C-callable, assembly-optimized, general-purpose signal-processing and image/video processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical.

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

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

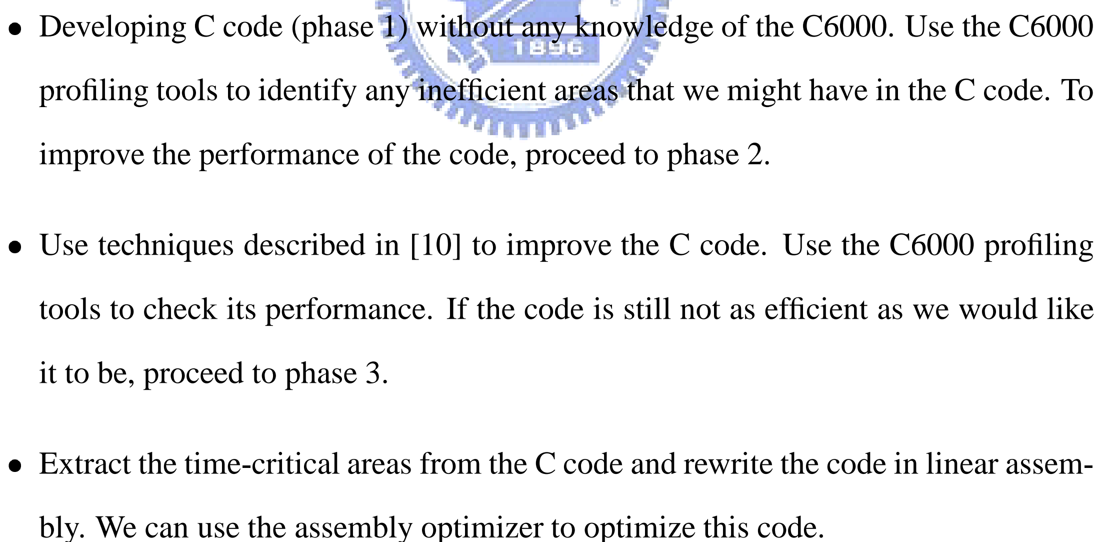
In our project, the FFT and IFFT functions are from this library.

3.4 Code Development Flow to Increase Performance [10]

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. 3.5. The tutorial section of the Programmers Guide focuses on phases 1 – 3. These phases will instruct the programmer when to go to 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 programmers high MIPS areas (loops). Based on this feedback, there are some very simple steps the programmer can take to pass complete and better information to the compiler allowing the programmer a quicker start in maximizing compiler performance. The following items list goal for each phase in the 3-step software development flow shown in Fig. 3.5.



- Developing C code (phase 1) without any knowledge of the C6000. Use the C6000 profiling tools to identify any inefficient areas that we might have in the C code. To improve the performance of the code, proceed to phase 2.

- Use techniques described in [10] to improve the C code. Use the C6000 profiling tools to check its performance. If the code is still not as efficient as we would like it to be, proceed to phase 3.

- Extract the time-critical areas from the C code and rewrite the code in linear assembly. We can use the assembly optimizer to optimize this code.

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

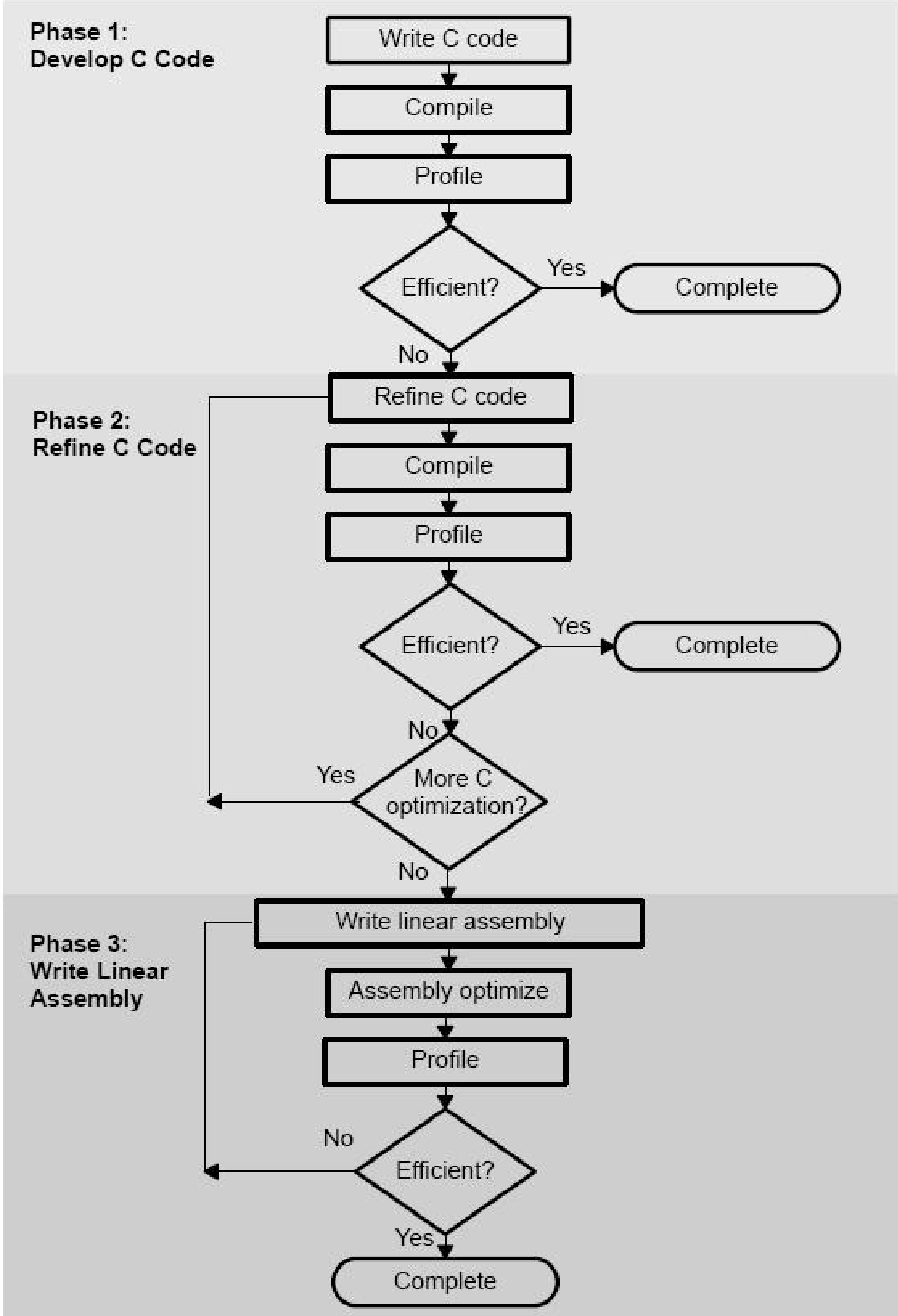


Fig. 3.5: Code development flow for TI C6000 DSP.

3.4.1 Compilier Optimization Options [10]

The compilier supports several options to optimize the code. The compilier options can be used to optimize code size or executing 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. Peforms all -o0 optimization, and:
  - Performs local copy/constant propagation.
  - Removes unused assignments.
  - Eliminates local common expressions.
- -o2. Performs all -o1 optimizations, and:
  - Performs software pipelining.
  - Performs loop optimizations.
  - Eliminates global common subexpressions.
  - Eliminates global unused assignments.

- Converts array references in loops to incremented pointer form.
- Performs loop unrolling.

- -o3. Performs all -o2 optimizations, and:
  - Removes all functions that are never called.
  - Simplifies functions with return values that are never used.
  - Inlines calls to small functions.
  - Reorders function declarations so that the attributes of called functions are known when the caller is optimized.
  - Propagates arguments into function bodies when all calls pass the same value in the same argument position.
  - Identifies file-level variable characteristics.

The -o2 is the defaule 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 to 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 hightest level optimization option. We use this option to optimization our code.

Chapter 4

DSP Implementation

Recall that 802.16a downlink synchronization process is as shown in Fig. 2.14. The process includes symbol timing synchronization, fractional frequency synchronization, integer frequency synchronization and frame synchronization. Our target is to implemente DL synchroization process on TI TMS32C6416 DSP.

Because of the memory on our platform is quite large, the most important issue to be optimized on our system is the execution efficiency. This chapter focuses on the performance improvement of the DL synchronization code. The DL synchronization programs developed in [1] employed floating-point computation. The code we implemente on DSP employs fixed-point computation. The precision of fixed-point numbers that we use is also discussed.

4.1 Efficiency Enhancement of DL Synchronization Code

The original DL synchronization program is written in C language. It is written without any knowledge of DSP at beginning. This section introduces the process of maximizing the performance.

4.1.1 Performance of the Original Program

The compile option that we use to optimize the original DL synchronization program is the program-level optimization. Tables 4.1 and 4.2 shows the the code size, maxi-

mum exection cycles, and minmmum exection cycles of individual function blocks for the transmitter and the receiver, respectively. Floating-point computation is used in the program. Because the C6416 is a fixed-point DSP, floating-point operations on it is time-consuming.

The transmitter consists of several function blocks that are listed in Table 4.1. Modulation performs the data modulation that IEEE 802.16a supports. The options of data modulation are QPSK, 16-QAM and 64-QAM. In our program, the modulation is fixed 64-QAM for all burst data. Framing performs the allocations of pilot carriers , guard carriers and burst data. Fft_float is the discrete fast fourier transfer from [1]. IFFT function includes the fft_float with some input data buffer arrangement of fft_float. Tx_mask_satisfaction performs the 4-times oversample and SRRC filter (from [1]).

The functions that executed in receiver are listed in Table 4.2. SRRC_downsample performs the 4-times downsample and SRRC filter (from [1]). CP_correlation, initial_freq_sync, integer_freq_sync, and pilot_corre functions perform the synchronization techniques that are CP correlation, fractional frequency synchronization, integer frequency synchronization and pilot correlation respectively. Fft_float in receiver is the same as that in transmitter with different input option. FFT consists of fft_float function and some input data buffer arrangement of fft_float. In de-framing function, data bursts are extracted from the received symbols. And finally, de-modulation of the burst data is performed in de-modulation function.

In our system, one symbol duration is 201.6 μs and there are 2304 samples in a symbol. The clock frequency of DSP is 600 MHz. The execution clock cycles is 120960 in a symbol duration and average 52.5 in a sample duration. The average counts of all transmitter functions are in a symbol duration. Their target counts are 120960 cycles for real time operation. In the receiver, the average count of fft_float, FFT, de_framing and de_modulation functions are in a symbol duration and their targets counts are 120960 for real time operations. For the other funcions in receiver, their average counts are in one

Table 4.1: Floating-Point Profile of 802.16a DL Transmitter Function Blocks

Block	Code size (Bytes)	Max. count (Cycles)	Min. count (Cycles)	Avg. count (Cycles)	Real time rate
Modulation	460	4294288	1441061	3058185	3.96%
Framing	2212	188125	188091	188110	64.30%
fft_float	1328	23487728	23476418	23481019	0.52%
IFFT	676	23491380	23480070	23484737	0.52%
Tx_mask_satisfaction	1852	46471084	46460414	46465084	0.26%

Table 4.2: Floating-Point Profile of 802.16a DL Receive Function Blocks

Block	Code Size (Bytes)	Max. count (Cycles)	Min. count (Cycles)	Avg. count (Cycles)	Real time rate
SRRC_downsample	608	23283	16387	21233	0.25%
CP_correlation	1188	185559	43	645	8.14%
initial_freq_sync	420	184	52	57	92.11%
integer_freq_sync	1228	23484952	40	2078	2.53%
pilot_corre	2972	24057628	48	167290	0.03%
sync	1132	47393690	56192	228702	0.02%
fft_float	1328	23258068	23250546	23254032	0.52%
FFT	420	23456722	23451576	23453957	0.52%
de_framing	948	1626187	1626187	1626187	7.44%
de_modulation	904	1883132	637124	1352664	8.94%

sample duration and their target counts are 52.5 cycles for real time operation. The real time rate listed in Table 4.1 and 4.2 show that the rate that average counts compared with real time requirement of individual function.

In this thesis, we will optimize the synchronization related functions. They are CP_correlation (CP correlation), initial_freq_sync (initial frequency synchronization) , integer_freq_sync (integer frequency synchronization) , pilot_corre (pilot correlation) and sync (synchronization). The sync function is the top-level function of synchronization.

4.1.2 Fixed-Point Number System Consideration

The C6416 is a fixed-point DSP. Floating-point operations on it are inefficient. We should realize the transmission system using fixed-point arithmetic to maximize the performance.

TI's programmer guide [10] recommands the user to use the short data type (16 bits) for fixed-point multiplication inputs whenever possible. Because this data type provides the most efficient use of the 16-bit multiplier in the C6416. Besides changing the data type, some sub-functions in this system such as FFT, IFFT, sine and cosine should be replaced by fixed-point version.

4.1.2.1 On the Precision of Fixed-Point Computation

The fixed-point number format that we use in the system to do arithmetic operations is Q.15. We choose the format because the most efficiency data format for the multiply operation is 16 bits, and the data used in synchronization process are less than 1 in their numerical values. Now, we evaluate whether the precision is enough for the synchronizationi work.

For this, we allocate 6 bursts (users) in the downlink part of one 802.16a frame. Source data are generated randomly, and are modulated to 64 QAM symbols. There are 12 OFDMA symbols in one DL frame and 4 OFDMA symbols in UL frames. The TTG and RTG are 136 samples. The frame structure and the bursts allocation are shown in Fig. 4.1. The frame is repeated several times in transmission.

In the simulation environment, we employ the multipath ETSI "Vehicular A" channel model [1]. The time-varying channel impulse response for these models can be described by

$$h(\tau, t) = \sum_i \alpha_i(t)\delta(\tau - \tau_i), \tag{4.1}$$

which defines the channel impulse response at time t as a function of the lag τ. The channel taps $\alpha_i(t)$ are independent complex stochastic variables, fading with Jakes' Doppler spectrum, with a maximum Doppler frequency of 240 Hz, reflecting a mobile speed of approximately 120 km/h (and scatterers uniformly distributed around the mobile). The real-valued τ_i and the variance of the complex-valued α_i are given in [13] and repeated in Table 4.3.

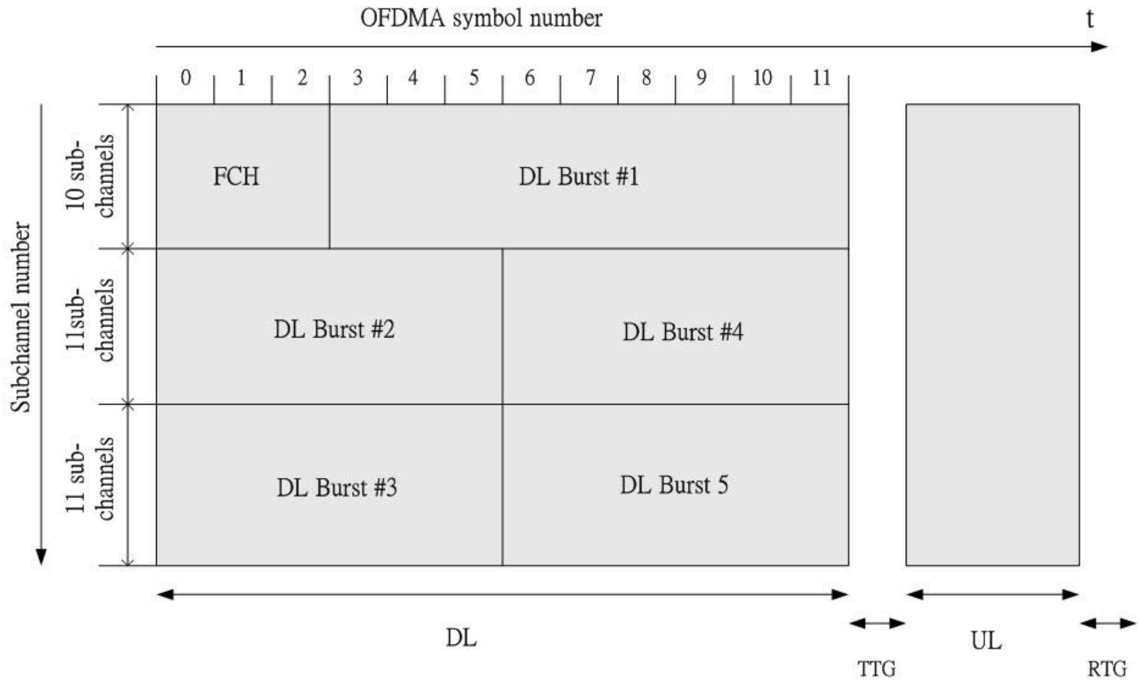


Fig. 4.1: The bursts allocation in a frame.

Table 4.3: Characteristics of the ETSI "Vehicular A" Channel Environment

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

Table 4.4: Relations Between Spreed and Maximum Doppler Shift at Carrier Frequency 6 GHz and Subcarrier Spacing 5.58 kHz

Speed (km/hr)	Doppler shift (Hz)	$f_d T_s$
0	0	0
20	111	0.0224
40	222	0.0448
60	333	0.0672
80	444	0.0896
100	556	0.112
120	557	0.134

The SNR is set to 10 dB in the fading chnanel. The receiver SNR specified in 802.16a test condition is from 9.4 to 24.4 dB, so 10 dB, which is almost the worst condition, is a reasonable value for simulation. The maximum Doppler shifts of our simulation are shown in Table 4.4 for the speed from 0 to 120 km/hr.

The goals of synchronization are to compensate the frequency offset and to find the frame start time. To evaluate the precision of fixed-point format, we compare the frequency lock and frame lock performance between floating-point system and fixed-point system. The frequency offset is estimated and compensated in the synchronization process. The frequency lock condition is achievd when the frequency offset is compensated. The frame lock condition is achieved when the three successive preamble symbols are identified. The simulation transmits 5 802.16a frames every time. If the frequency lock and frame lock are not obtained in these 5 frames, the synchronization is declared to fail. The current symbol number is recorded when the frequency is locked, and the current frame number is recorded when the frame is locked. The average symbol number of frequency lock and frequency lock fail rate is used to measure the performance of frequency lock, and the average frame number of frame lock and the frame lock fail rate is used to measure the performance of frame lock. Tables 4.5 and 4.6 show the simulation result.

The frequency offset is always locked in 5 frames duration. And it takes on average no more than 6 symbols to achieve the frequency lock. The performance is not clearly

Table 4.5: Performance Comparision of Frequency Lock Between Floating-Point and Fixed-Point Implementation

Doppler shift $f_d T_s$	Lock fail rate		Average lock symbol number	
	Floating-point	Fixed-point	Floating-point	Fixed-point
0	0	0	2.99	2.98
0.0224	0	0	2.66	2.69
0.0448	0	0	2.36	2.39
0.0672	0	0	2.30	2.32
0.0896	0	0	2.61	2.57
0.112	0	0	3.23	3.42
0.134	0	0	5.15	5.14

Table 4.6: Performance Comparision of Frame Lock Between Floating-Point and Fixed-Point Implementation

Doppler shift $f_d T_s$	Lock fail rate		Average lock frame number	
	Floating-point	Fixed-point	Floating-point	Fixed-point
0	0.001	0.001	1.00	1.00
0.0224	0.057	0.074	1.98	1.94
0.0448	0.008	0.100	1.26	1.24
0.0672	0.027	0.032	1.65	1.70
0.0896	0.136	0.140	2.59	2.59
0.112	0.107	0.135	2.14	2.19
0.134	0.063	0.069	1.50	1.47

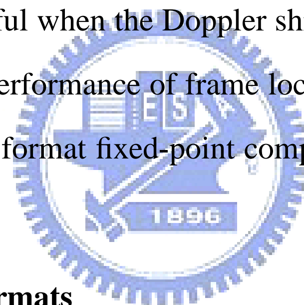
Table 4.7: Q16.15 Bit Fields

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

Table 4.8: Q.15 Bit Fields

Bits	15	14	13	...	1	0
Value	S	Q14	Q13	...	Q1	Q0

dependent on Doppler shifts. The performance of fixed-point system is very close to that of the floating-point system. The probability of frame locking in 5 received frames is not very high when the Doppler shift exists. But the frame can be locked quickly when Doppler shift is 0. This is because the IEEE 802.16a is designed for fixed environments. The useable information is useful when the Doppler shift is small. Comparing the simulation results, we see that the performance of frame lock is close in floating-point and in fixed-point systems. The Q.15 format fixed-point computation is precise enough for the synchronization process.



4.1.2.2 Fixed-Point Data Formats

In the transmitter (TX) side, as Fig. 2.2 shows, multiplication only exists in modulation, IFFT and the 4-times upsample SRRC filter. The data formats we set in the TX side are:

- The data format before IFFT is Q16.15.
- The data format after IFFT is Q.15.

Q16.15 format places the sign bit in the leftmost, followed by 16 integer bits and 15 bits fraction component (Table 4.7). Q.15 format places the sign bit in the leftmost, and the remainder 15 bits are fraction component (Table 4.8).

The range of data values before IFFT is $[-\frac{4}{3}, \frac{4}{3}]$, and the data after IFFT is less than 1 in their numerical values. The critical functions in TX are FFT and SRRC filter. We can get the FFT/IFFT code from TI TMS320C64x DSP library (DSPLIB). This library

supports two types of FFT/IFFT. They are 16 bits input/output data type and 32 bits input/output data type. The inputs of the FFT/IFFT must be scaled by the FFT length to prevent overflow. In our 802.16a system, the FFT/IFFT length is 2048. If we use the 16 bits type FFT/IFFT, the input data format of FFT/IFFT must be scaled by 2048. In this case, only 4 bits can be used to represent the fixed-point value. Intuitively, 4 bits is not enough in our system. For this reason, we choose the 32 bits FFT/IFFT DSP_fft32x32 and DSP_ifft32x32.

DSP_fft32x32 is the complex mixed radix 32-×32-bit FFT with rounding, while inverse FFT version of the same type is DSP_ifft32x32. It computes an extended precision complex forward mixed radix FFT with rounding and digital reversal. Input data $x[]$, output data $y[]$ and coefficients $w[]$ are 32-bit. The output is returned in the separate array y[ ] in normal order. The FFT coefficients (twiddle factors) are generated using the program "tw_fft32x32". No scaling is done with the routine; thus the input data must be scaled by $2^{\log_2 N}$ to completely prevent overflow. The routine uses $\log_4 N - 1$ stages of Cooley Tukey radix-4 DIF FFT and performs either a radix-2 or radix-4 DIF FFT on the last stage depending on N. If N is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform. In our work, we have 5 stages of radix-4 transform and 1 stage radix-2 transform.

Table 4.9 shows the comparisons of computational complexity for different FFT algorithm. The mixed radix FFT needs 19203 real multiplications and 64259 real additions theoretically in our application. Practically, the time DSP_fft32x32/DSP_ifft32x32 needed is 2811 clock cycles. The complexity and performance of IFFT/FFT are listed in Table 4.10. The efficiency of DSP_fft32x32 is quiet high because the code is assembly-optimized. The software pipeline is well scheduled as show in Fig. 4.2.

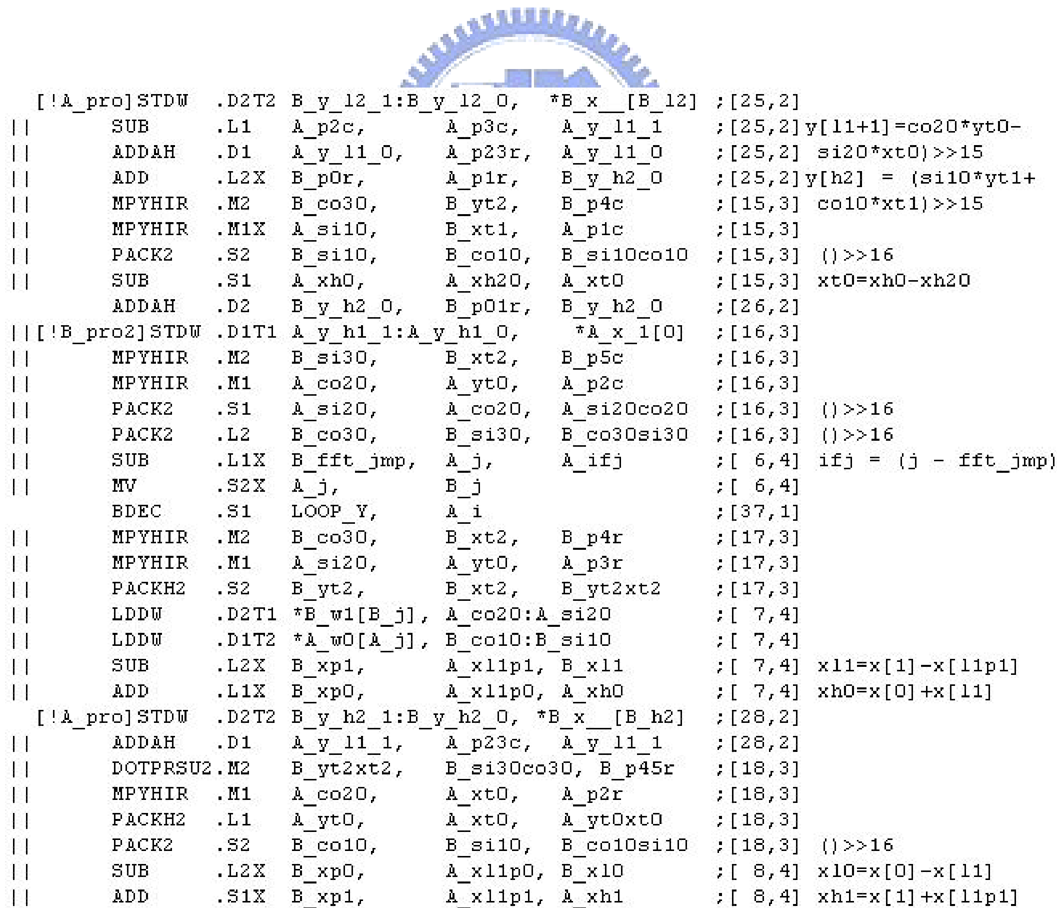
We set the data format before IFFT as Q16.15 rather arbitrarily. The most critical arithmetic operation in TX side is the SRRC filter. We set the data format after IFFT as Q.15 so that the inputs of multiplication in SRRC filter is 16 bits. This is the most efficient

Table 4.9: Comparisons of Computational Complexity for Different FFT Algorithms

Complexity	No. of Real Multiplications	No. of Real Additions
Radix-2 FFT	$\frac{2}{3}N\log_2 N - \frac{7}{2}N + 8$	$\frac{5}{2}N\log_2 N - \frac{7}{2}N + 8$
Radix-4 FFT	$\frac{9}{8}N\log_2 N - 3N + 3$	$\frac{25}{8}N\log_2 N - 3N + 3$
Radix-8 FFT	$\frac{25}{24}N(\log_2 N - 3) + 4$	$\frac{73}{24}N\log_2 N - \frac{25}{8}N + 4$
Split-radix-4/2 FFT	$N\log_2 N - 3N + 4$	$3N\log_2 N - 3N + 4$
Simplified FFT	$4N$	$6N$

Table 4.10: Complexity and Performance of IFFT/FFT Implementation

	Needed Number of Clock Cycles	Actual Number of Clock Cycles	Performance
IFFT/FFT	20311	28811	70.5%



```
   [!A_pro]STDW   .D2T2 B_y_l2_1:B_y_l2_0,   *B_x__[B_l2]  ;[25,2]
||         SUB    .L1   A_p2c,        A_p3c,    A_y_l1_1     ;[25,2]y[l1+1]=co20*yt0-
||         ADDAH  .D1   A_y_l1_0,     A_p23r,   A_y_l1_0     ;[25,2] si20*xt0)>>15
||         ADD    .L2X  B_p0r,        A_p1r,    B_y_h2_0     ;[25,2]y[h2] = (si10*yt1+
||         MPYHIR .M2   B_co30,       B_yt2,    B_p4c        ;[15,3] co10*xt1)>>15
||         MPYHIR .M1X  A_si10,       B_xt1,    A_p1c        ;[15,3]
||         PACK2  .S2   B_si10,       B_co10,   B_si10co10   ;[15,3] ()>>16
||         SUB    .S1   A_xh0,        A_xh20,   A_xt0        ;[15,3] xt0=xh0-xh20
           ADDAH  .D2   B_y_h2_0,     B_p01r,   B_y_h2_0     ;[26,2]
||[!B_pro2]STDW   .D1T1 A_y_h1_1:A_y_h1_0,     *A_x_1[0]     ;[16,3]
||         MPYHIR .M2   B_si30,       B_xt2,    B_p5c        ;[16,3]
||         MPYHIR .M1   A_co20,       A_yt0,    A_p2c        ;[16,3]
||         PACK2  .S1   A_si20,       A_co20,   A_si20co20   ;[16,3] ()>>16
||         PACK2  .L2   B_co30,       B_si30,   B_co30si30   ;[16,3] ()>>16
||         SUB    .L1X  B_fft_jmp,    A_j,      A_ifj        ;[ 6,4] ifj = (j - fft_jmp)
||         MV     .S2X  A_j,          B_j                    ;[ 6,4]
           BDEC   .S1   LOOP_Y,       A_i                    ;[37,1]
||         MPYHIR .M2   B_co30,       B_xt2,    B_p4r        ;[17,3]
||         MPYHIR .M1   A_si20,       A_yt0,    A_p3r        ;[17,3]
||         PACKH2 .S2   B_yt2,        B_xt2,    B_yt2xt2     ;[17,3]
||         LDDW   .D2T1 *B_w1[B_j], A_co20:A_si20            ;[ 7,4]
||         LDDW   .D1T2 *A_w0[A_j], B_co10:B_si10            ;[ 7,4]
||         SUB    .L2X  B_xp1,        A_xl1p1, B_xl1         ;[ 7,4] xl1=x[1]-x[l1p1]
||         ADD    .L1X  B_xp0,        A_xl1p0, A_xh0         ;[ 7,4] xh0=x[0]+x[l1]
   [!A_pro]STDW   .D2T2 B_y_h2_1:B_y_h2_0, *B_x__[B_h2]      ;[28,2]
||         ADDAH  .D1   A_y_l1_1,     A_p23c,   A_y_l1_1     ;[28,2]
||         DOTPRSU2.M2  B_yt2xt2,     B_si30co30, B_p45r     ;[18,3]
||         MPYHIR .M1   A_co20,       A_xt0,    A_p2r        ;[18,3]
||         PACKH2 .L1   A_yt0,        A_xt0,    A_yt0xt0     ;[18,3]
||         PACK2  .S2   B_co10,       B_si10,   B_co10si10   ;[18,3] ()>>16
||         SUB    .L2X  B_xp0,        A_xl1p0, B_xl0         ;[ 8,4] xl0=x[0]-x[l1]
||         ADD    .S1X  B_xp1,        A_xl1p1, A_xh1         ;[ 8,4] xh1=x[1]+x[l1p1]
```

Fig. 4.2: A part of assembly code for DSP_fft32x32.

use of the 16-bit multiplier in C6416. The output of SRRC is Q.15 for the arithmetic operations of RX side are 16 bits fixed-point. Fig. 4.3 shows the data format of TX side.

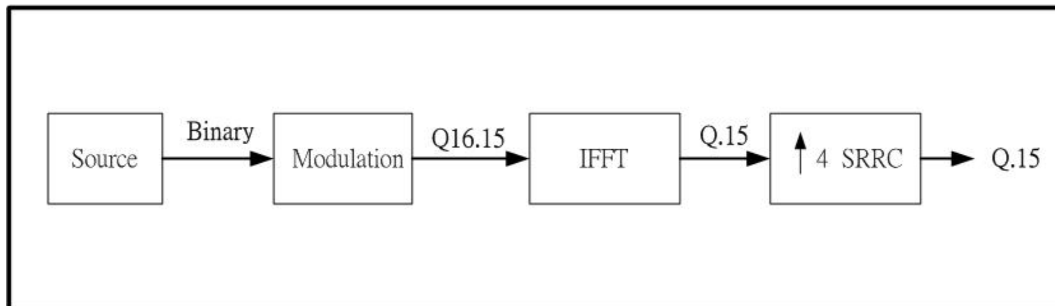


Fig. 4.3: The fixed-point data formats at the TX side.

In the RX side, the operation is much more complex than in TX side. The main consideration of setting fixed-point data format is that the multiplier operations are always 16×16. The data formats we set in the RX side are:

- The data format of SRRC filter input is Q.15.
- The data format after SRRC filter is Q.15.
- The data format after FFT is Q16.15.
- The data format of estimated frequency offset is Q16.15.

Fig. 4.4 shows the data formats at RX side. The stages after FFT are de-framing and de-modulation. The range of data values after FFT is $[-\frac{4}{3}, \frac{4}{3}]$ and the Q.15 format can not cover this range. The performance of these functions is not discussed in this thesis, and we set the fractional part of fixed-point data after FFT to be 15 bits for simplifying the data format transformation. For these reasons, the data format after FFT is set to be Q16.15. The fractional part of fixed-point number in this system in 15 bits. Hence the finest fractional resolution is $2^{-15} = 3.05 \times 10^{-5}$.

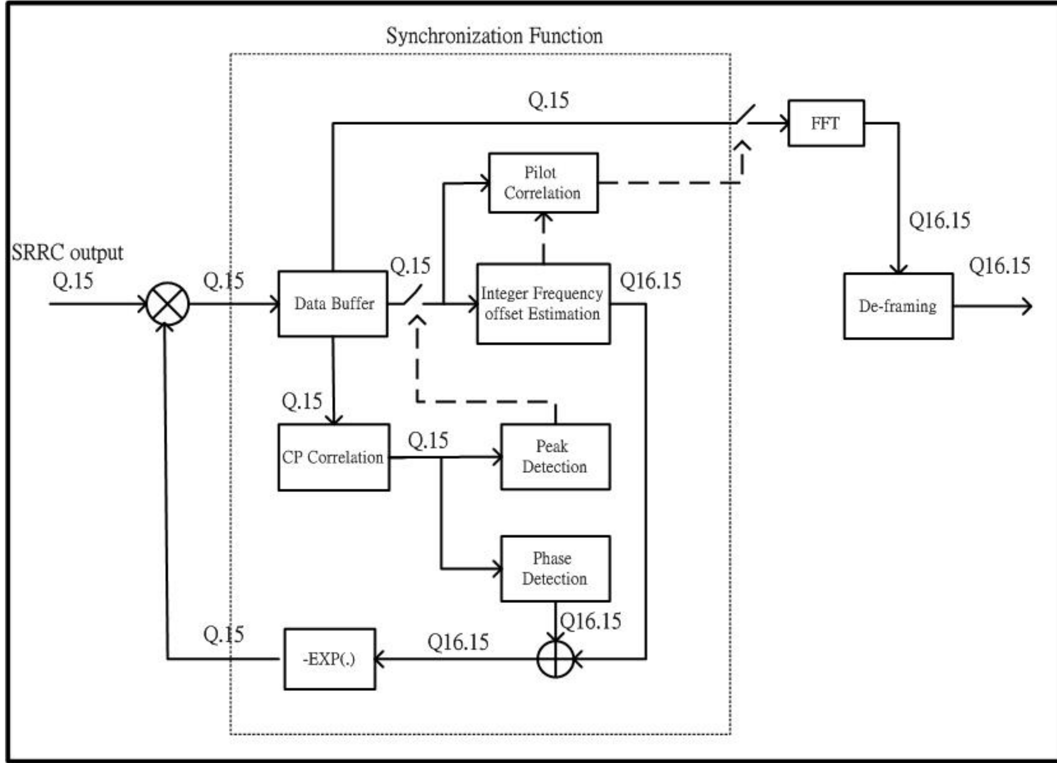


Fig. 4.4: The fixed-point data formats at the RX side

4.1.2.3 Fixed-Point Sine and Cosine Functions

The sine and cosine functions in RX side are used to compensate the frequency offset. The library of TI C compiler only supports the floating-point version. We have to replace these two functions by fixed-point version for efficiency. There are several methods that can be used to accomplish these two functions. The table look-up method is faster than the series expansion [12]. The former stores values of the function and the values of the slope used to interpolate between the table entries. If we let the constants in the table be represented by C_i, and the interpolation values (multipliers) by M_i, a function table might appear as shown in Table 4.11, and

$$\sin(x) = C_i + M_i x \tag{4.2}$$

where

$$C_i = \sin(\theta_i) - i \times (\sin(\theta_{i+1}) - \sin(\theta_i)), \tag{4.3}$$

Table 4.11: Sine/Cosine Look-Up Table

0	$\sin 0$	$\cos 0$
θ_1	$\sin\theta_1$	$\cos\theta_1$
θ_2	$\sin\theta_2$	$\cos\theta_2$
.	.	.
.	.	.
.	.	.
θ_n	$\sin\theta_n$	$\cos\theta_n$

$$M_i = \frac{\sin(\theta_{i+1}) - \sin(\theta_i)}{\theta_{i+1} - \theta_i}. \tag{4.4}$$

Once the program has been written to use the table look-up method, it can be used to generate any function required by changing the values in the table. So the equation for cosine is the same as that for sine.

In this thesis, the table length is 512 and its data type is Q.15. The $\theta_i = \frac{2\pi i}{512}$, and

$$(\theta_{i+1} - \theta_i) = \frac{2\pi}{512}.$$

The input data can be normized by a factor 2π for convenience. Then (4.4) can be modified to

$$M_i = (\sin(\theta_{i+1}) - \sin(\theta_i)) \times 512. \tag{4.5}$$

The error mean of the fixed-point sine/cosine function is 2.07×10^{-5} and the mean square error is 5.99×10^{-10}. The precision is close to the resolution of the system.

4.1.2.4 Performance of Fixed-Point System

After the data format is changed to fixed-point, the operation performance is shown in Tables 4.12 and 4.13. The performance is much better than the floating-point in the synchronization related functions which including CP_correlation, initial_freq_sync, pilot_corre and sync. Some other functions such as framing and de_framing are not much enhanced because the arthimetic operations are not the critical factors of their execution efficiency. The critical factors of these functions are the interface of input and output data.

Table 4.12: Fixed-Point Profile of 802.16a DL Transmitter Function Blocks

Block	Code Size (Bytes)	Max. count (Cycles)	Min. count (Cycles)	Avg. count (Cycles)	Improvement (compare with floating-point operations)	Real time rate
Modulation	616	2716875	906088	1932318	36.81%	6.26%
Framing	1624	191530	191496	191515	-1.81%	53.16%
IFFT	964	37528	37528	35728	99.85%	338.56%
Tx_mask_satisfaction	1624	6199459	6199459	6199459	86.55%	1.95%

Table 4.13: Fixed-Point Profile of 802.16a DL Receiver Function Blocks

	Code Size (Bytes)	Max. count (Cycles)	Min. count (Cycles)	Avg. count (Cycles)	Improvement (compare with floating-point operations)	Real time rate
SRRC_downsample	700	8942	1175	1301	93.87%	4.01%
CP_correlation	1040	376	37	80	87.60%	65.63%
initial_freq_sync	312	179	32	38	33.33%	138.16%
integer_freq_sync	1276	65128	39	42	97.98%	125%
pilot_corre	2400	638902	38	8462	94.94%	0.62%
sync	1132	713535	9759	18114	92.08%	0.29%
FFT	276	32259	32259	32259	99.86%	374.96%
de_framing	1036	1225985	1225985	1225985	24.61%	9.87%
de_modulation	460	755037	252196	537886	60.24%	22.49%

The performance of these functions should be fine-tuned but we have not worked on it in this thesis.

The synchronization related functions can be further improved by refining the program code.

4.1.3 Code Refinement

4.1.3.1 Recursive DFT in Pilot Correlation Function

In pilot correlation fucntion, the FFT is executed several times in one symbol duration. FFT should be done 64 times in initial condition and 10 times in normal condition during

```
1 void Recursive_DFT(FIXED_DOUBLE *fft_Out, FIXED x_old_real, FIXED x_old_imag,
2                   FIXED x_new_real,FIXED x_new_imag){
3         FIXED_DOUBLE temp_real,temp_imag;
4         int i;
5         for(i=0;i<2048;i++){
6                 temp_real=fft_Out[2*i]-x_old_real+x_new_real;    //Q16.15
7                 temp_imag=fft_Out[2*i+1]-x_old_imag+x_new_imag; //Q16.15
8                 fft_Out[2*i]=((temp_real)*rcos[i]-(temp_imag)*isin[i])>>15;
9                 fft_Out[2*i+1]=((temp_real)*isin[i]+(temp_imag)*rcos[i])>>15;
10        }
11 }
```

Fig. 4.5: C code of recursive DFT.

Table 4.14: Comparison Between FFT and Recursive DFT

	Code Size (Bytes)	Clock Cycles
DSP_fft32x32	932	28811
Recursive_DFT	652	6172

one symbol time. These FFT can be calculated recursively as discussed before as

$$X_n(k) = [X_{n-1}(k) - x_{n-N} + x_n]\, e^{j\frac{2\pi k}{N}}.$$

The input $\frac{2\pi k}{N}$ to sine and cosine in this equation is not a random number. We can store these sine and cosine values in a table to simplify the calculation. The resulting C code is shown in Fig. 4.5, and Table 4.14 shows the profile of the recursive DFT.

The recursive DFT calculates 2048 complex multiplications. One complex multiplication needs 4 real multiplications and 2 additions. The recursive DFT thus takes $2048 \times 4 = 8192$ multiplications and $2048 \times 2 = 4096$ additions. There are 2 multipliers and 6 ALUs in TI C6416 DSP, so the lower-bound execution time of recursive DFT is $\frac{8192}{2} + \frac{4096}{6} = 4779$ clock cycles. The efficiency of the recursive DFT implementation is 77.4% as Table 4.15 shows. The software pipeline information of the recursive DFT program is shown in Fig. 4.6. The resource is partitioned equally and software pipeline is well scheduled. Fig. 4.7 shows a part of the assembly code.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                : 23
;*      Loop opening brace source line : 23
;*      Loop closing brace source line : 31
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 1024
;*      Known Maximum Trip Count         : 1024
;*      Known Max Trip Count Factor      : 1024
;*      Loop Carried Dependency Bound(^) : 2
;*      Unpartitioned Resource Bound     : 6
;*      Partitioned Resource Bound(*)    : 6
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     5        4
;*      .D units                     6*       6*
;*      .M units                     4        4
;*      .X cross paths               3        3
;*      .T address paths             6*       6*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        1     (.L or .S unit)
;*      Addition ops (.LSD)          5        5     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             3        3
;*      Bound(.L .S .D .LS .LSD)     6*       6*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Schedule found with 5 iterations in parallel
;*      done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 3
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      For further improvement on this loop, try option -mh32
;*
;*      Minimum safe trip count     : 4 (after unrolling)
;*----------------------------------------------------------------------------*
```

Fig. 4.6: The software pipeline information of recursive DFT.

Table 4.15: Efficiency of Recursive DFT Implementation

	Lower-Bound of Execution Cycles	Actual Execution Cycles	Efficiency
Recursive_DFT	4778	6172	77.4%

```
13373 ;** -----------------------------------------------------------------------*
13374 L40:    ; PIPED LOOP KERNEL
13375
13376    [ A0]   BDEC    .S1     L40,A0          ;
13377 ||         SUB     .L2     B6,B4,B5        ; |28|
13378 ||         SHR     .S2     B23,13,B4       ; |29|
13379 ||         MPYLI   .M1     A21,A7,A7:A6    ; @|28|
13380 ||         SUB     .L1X    B22,A19,A22     ; @@|25|
13381 || [!B0]   LDH     .D2T2   *++B16(4),B9    ; @@|28|
13382 ||         LDW     .D1T1   *+A3(12),A9     ; @@@|25|
13383
13384            SUB     .S1     A16,A6,A4       ; |28|
13385 ||         ADD     .L1     A8,A4,A6        ; |29|
13386 ||         SHR     .S2     B5,13,B5        ; |28|
13387 ||         MPYLI   .M1X    A21,B21,A5:A4   ; @|29|
13388 ||         MPYLI   .M2X    B7,A20,B5:B4    ; @|29|
13389 ||         SUB     .L2     B6,B19,B22      ; @@|24|
13390 || [!B0]   LDH     .D2T1   *+B16(2),A21    ; @@|28|
13391 ||         LDW     .D1T2   *+A3(4),B22     ; @@@|25|
13392
13393            SHR     .S1     A6,13,A6        ; |29|
13394 ||         MPYLI   .M1     A9,A7,A9:A8     ; @|29|
13395 ||         MPYLI   .M2X    B9,A20,B5:B4    ; @|28|
13396 ||         ADD     .L1     A18,A22,A8      ; @@|25|
13397 || [!B0]   LDH     .D2T1   *+B17(2),A9     ; @@|28|
13398 ||         SHR     .S2     B8,2,B22        ; @@|28|
13399 ||         ADD     .L2     B18,B22,B7      ; @@|24|
13400 ||         LDW     .D1T2   *+A3(8),B6      ; @@@|24|
13401
13402            SHR     .S1     A4,13,A9        ; |28|
13403 || [!A1]   STW     .D1T1   A6,*-A3(36)     ; |29|
13404 || [!A1]   STW     .D2T2   B5,*++B20(16)   ; |28|
13405 ||         MPYLI   .M1X    A9,B21,A17:A16  ; @|28|
13406 ||         ADD     .L1     A18,A5,A4       ; @@|25|
13407 ||         SHR     .S2     B7,2,B21        ; @@|28|
13408
```

Fig. 4.7: Assembly code of recursive DFT.

Table 4.16: The Execution Cycles of Pilot Correlation Loop

	Original Code (Cycles)	Refined Code (Cycles)
Pilot Correlation Loop	76293	1013

4.1.3.2 Pilot Correlation Function Refinement

In pilot correlation function, the locations of pilots are found in the frequency domain and then they are used to be the reference of correlation. The pilot locations in OFDMA symbol has only 4 types depending on the symbol index L. We can store the pilot locations instead of calculating them time after time. Besides, the values of the pilots are either $\frac{4}{3}$ or $-\frac{4}{3}$. We do not need to do multiplications to find the maximum of pilot correlation. We can replace the multipliers by additions.

The revised code is shown in Fig. 4.8. The pilot locations are stored in var_pilot_loc array and no multiplication is needed. The most important enhancement of the performance is that the loop count is reduced from 1702 (the useful carriers number) to 144 (the variable carriers number), as shown in Table 4.16. Fig 4.9 shows the partial assembly code of original pilot correlation loop. It is a disqualified loop and the software pipeline is not scheduled. The software pipeline information of revised code is shown in Fig. 4.10, which shows that the software pipline is well scheduled. Fig 4.11 shows a part of the assembly code.

4.1.3.3 Using Intrinsics

The C6000 compiler provides intrinsics, which are special functions that map directly to inlined C62x/C64x/C67x instructions, to optimize the C/C++ code quickly. The intrinsic function we used in synchronization code is the integer absolute value instruction _abs(). The absoulate value function used originally was fabs(). In TI's CCS library, it takes 70 clock cycles to perform the absolute value computation through fabs(). Replacing the library function by the instrinsic enhances the execution performance. Fig. 4.12 shows

```
1 //==========================================================//
2 // The original pilot correlation code                      //
3 //==========================================================//
4 for(used_carrier=0;used_carrier<1702;used_carrier++)
5 {
6         if(((used_carrier-3*L+12)%12==0))
7         {
8
9
10                if(used_carrier<851)
11                        fft_carrier=used_carrier+1197;
12                else
13                        fft_carrier=used_carrier-850;
14                if( ((wk[used_carrier/8]<<(used_carrier%8))&(0x80))==0x80 )
15                        freq_corre[preamble][L]=freq_corre[preamble][L]-fft_Out[fft_carrier*2];
16                else
17                        freq_corre[preamble][L]=freq_corre[preamble][L]+fft_Out[fft_carrier*2];
18        }
19 }
20
21
22 //==========================================================//
23 // The refined pilot correlation code                       //
24 //==========================================================//
25 for(i=0;i<corr_pilot_cnt;i++){
26        used_carrier= var_pilot_loc[i][L];
27        if(used_carrier<851)
28                fft_carrier=used_carrier+1197;
29        else
30                fft_carrier=used_carrier-850;
31
32        if( ((wk[used_carrier/8]<<(used_carrier%8))&(0x80))==0x80 )
33                freq_corre[preamble][L]=freq_corre[preamble][L]-fft_Out[fft_carrier*2];
34        else
35                freq_corre[preamble][L]=freq_corre[preamble][L]+fft_Out[fft_carrier*2];
36 }
```

Fig. 4.8: C code of revised pilot correlation loop.

```
 1 ;----------------------------------------------------------------
 2 ; 531 | for(used_carrier=0;used_carrier<1702;used_carrier++)
 3 ;----------------------------------------------------------------
 4 ;*---------------------------------------------------------------
 5 ;*   SOFTWARE PIPELINE INFORMATION
 6 ;*      Disqualified loop: bad loop structure
 7 ;*---------------------------------------------------------------
 8 L49:
 9             B        .S1      __remi            ; |533|
10             SUB      .D1      A3,A8,A4          ; |533|
11             ADDKPC   .S2      RL146,B3,1        ; |533|
12             ADD      .D1      12,A4,A4
13             MVK      .D2      0xc,B4            ; |533|
14 RL146:      ; CALL OCCURS                       ; |533|
15             MV       .D1      A4,A0             ; |533|
16    [ A0]    BNOP     .S1      L50,4
17    [!A0]    MVK      .S1      0x80,A4
18             ; BRANCH OCCURS                     ; |533|
19
20             CMPLT    .L1      A3,A17,A0         ; |537|
21
22    [!A0]    MVK      .S2      850,B4            ; |537|
23 || [ A0]    MVK      .S1      1197,A5           ; |537|
24
25    [ A0]    ADD      .D1      A5,A3,A5          ; |537|
26
27    [ A0]    EXT      .S1      A5,16,16,A5       ; |537|
```

Fig. 4.9: Partial assembly code of original pilot correlation loop.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line               : 574
;*      Loop opening brace source line : 574
;*      Loop closing brace source line : 585
;*      Known Minimum Trip Count         : 142
;*      Known Maximum Trip Count         : 142
;*      Known Max Trip Count Factor      : 142
;*      Loop Carried Dependency Bound(^) : 6
;*      Unpartitioned Resource Bound     : 6
;*      Partitioned Resource Bound(*)    : 6
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        1
;*      .S units                     6*       5
;*      .D units                     2        2
;*      .M units                     0        0
;*      .X cross paths               2        5
;*      .T address paths             2        2
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        0     (.L or .S unit)
;*      Addition ops (.LSD)          8        6     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             4        3
;*      Bound(.L .S .D .LS .LSD)     6*       5
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 4 iterations in parallel
;*      done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      For further improvement on this loop, try option -mh48
;*
;*      Minimum safe trip count     : 4
;*----------------------------------------------------------------------------*
```

Fig. 4.10: The software pipeline information of pilot correlaton loop

```
14866 ;** ----------------------------------------------------------------------*
14867 L57:    ; PIPED LOOP KERNEL
14868
14869    [!A0]   LDW     .D2T2   *B19,B7             ;  ^ |696|
14870 ||         CMPLT   .L2X    A4,B5,B0            ; @|688|
14871 ||         SHRU    .S2     B17,29,B17          ; @|689|
14872 ||         LDBU    .D1T1   *+A18[A16],A22      ; @|689|
14873
14874    [ B1]   BDEC    .S2     L57,B1              ;
14875 || [ A0]   LDW     .D2T2   *B18,B16            ;  ^ |694|
14876 || [!B0]   EXT     .S1     A6,16,16,A21        ; @|688|
14877 ||         ADD     .L2X    B17,A19,B17         ; @|689|
14878
14879    [ B0]   EXT     .S1     A3,16,16,A21        ; @|688|
14880 ||         ANDN    .D2     B17,B6,B17          ; @|689|
14881 ||         ADD     .D1     A8,A4,A6            ; @@|688|
14882 ||         MV      .L1     A4,A18              ; @@|689|
14883
14884            ADD     .D1     A17,A4,A3           ; @@|688|
14885 ||         MV      .L1     A4,A19              ; @@|689|
14886 ||         SHR     .S1     A18,2,A16           ; @@|689|
14887
14888            SHL     .S2X    A21,3,B17           ; @|696|
14889 ||         SUB     .L1X    A9,B17,A16          ; @|689|
14890 ||         SHRU    .S1     A16,29,A9           ; @@|689|
14891 ||         LDH     .D1T1   *++A5(16),A4        ; @@@|687|
14892
14893    [!A0]   ADD     .L2     B7,B8,B8            ;  ^ |696|
14894 ||         SHL     .S2X    A21,3,B17           ; @|694|
14895 ||         ADD     .D2     B4,B17,B19          ; @|696|
14896 ||         SHL     .S1     A22,A16,A18         ; @|689|
14897 ||         MV      .D1     A19,A9              ; @@Inserted to split a long life
14898 ||         ADD     .L1     A9,A18,A16          ; @@|689|
```

Fig. 4.11: Partial assembly code of revised pilot correlation loop.

```
 1 //============================================================//
 2 // The original code, link to library function                //
 3 //============================================================//
 4
 5 if( (fabs(fft_Out[849*2])>check_null_threshold_2) || (fabs(fft_Out[850*2])>check_null_threshold_2) ||
 6         (fabs(fft_Out[851*2])>check_null_threshold_2) || (fabs(fft_Out[1197*2])>check_null_threshold_2) ||
 7         (fabs(fft_Out[1198*2])>check_null_threshold_2) || (fabs(fft_Out[1199*2])>check_null_threshold_2))
 8
 9                                 ...
10                                 ...
11 }
12
13
14
15 //============================================================//
16 // The refined code, using instrinsics                        //
17 //============================================================//
18
19 if( (_abs(fft_Out[849*2])>check_null_threshold_2) || (_abs(fft_Out[850*2])>check_null_threshold_2) ||
20         (_abs(fft_Out[851*2])>check_null_threshold_2) || (_abs(fft_Out[1197*2])>check_null_threshold_2) ||
21         (_abs(fft_Out[1198*2])>check_null_threshold_2) || (_abs(fft_Out[1199*2])>check_null_threshold_2))
22
23                                 ...
24                                 ...
25 }
```

Fig. 4.12: The abs() function is replaced by instrinsic _abs() in C code.

the change in program.

4.1.3.4 Synchronization Buffer Arrangement

Table 4.13 shows that the minimum clock cycle of sync function is 9759. The minimum condition should be much faster because the minimum condition of sync is almost idle. The inefficiency in sync code is caused by that the original code uses the shift-register buffer. Whenever a new data is received, the data in buffer are shifted left, and the new data is put in the rightmost position. If the shift-register buffer had been implemented in hardware, then the shift operation can be done in one clock cycle. But in the DSP software, shifting all the data in the buffer, as shown in Fig. 4.13, is time-consuming. After changing the buffer to a circular buffer, the new data input is kept in the buffer in circular order. So the code used to handle buffer like Fig. 4.13 is no more needed.

The minimum count of sync function should close to idle. Table 4.17 lists the execution profile of sync functions which use different buffer arrangements. It shows that the performance is much enhanced by using a circular buffer in place of a shift-register buffer.

```
temp_real=sync_buffer_1_real[buffer_max_num];
temp_imag=sync_buffer_1_imag[buffer_max_num];

for(i=buffer_max_num;i>0;i--)
{
        sync_buffer_1_real[i]=sync_buffer_1_real[i-1];
        sync_buffer_1_imag[i]=sync_buffer_1_imag[i-1];

        sync_buffer_2_real[i]=sync_buffer_2_real[i-1];
        sync_buffer_2_imag[i]=sync_buffer_2_imag[i-1];

}
sync_buffer_2_real[0]=temp_real;
sync_buffer_2_imag[0]=temp_imag;
```

Fig. 4.13: Shift-register buffer arrangement.

Table 4.17: Profile of the sync Function

	Shift buffer (Cycles)	Circular buffer (Cycles)
Minimum count of sync	9759	312

Table 4.18: Profile of CP Correlation Function Loop Using Different Buffer Types

	Code Size (Bytes)	Maximum count (Cycles)	Minimum Count (Cycles)	Average Count (Cycles)
CP_correlation (Shifted Buffer)	1040	376	37	80
CP_correlation (Circular Buffer)	844	1329	37	99

4.1.3.5 Loop Unrolling

The maximum count of CP correlation is increased when using circular buffer as shown in Table 4.18. The maximum count occurs when the CP correlation is caculated for the first time. The method of calculation is given in (2.3). The code of CP correlation functions using two different buffer types are shown in Fig. 4.14. When using shift-register buffer, the data used to calculate correlation are obtained from fixed memory addresses. It is much easier for compiler to optimize the code. When using the circular buffer, the data used to calculate correlation are from a different memory address each time. And the pointer of circular buffer must be checked whether or not it arrives the end of buffer. The conditional statement in loop is very difficult for compiler to do optimization.

The software pipeline scheduling of CP correlation function using shifted-register buffer is shown in Fig. 4.15, and that of using circular buffer is shown in Fig. 4.16. Resource partition for shift-register buffer type CP correlation is apparently better than for circular buffer type correlation. And the loop is unrolled automatically in the case of shift-register buffer type of CP correlation, but not in the case of circular buffer type. This is because the compiler does not know how to unroll the loop with conditional operation in it. We unrolled the loop 4 times by hand as shown in Fig. 4.17. The software pipeline information is as shown in Fig. 4.18. The resource partition is better than before and the software pipeline is well scheduled.

The C6416 DSP has 2 .M units, and each unit can execute dual 16×16 multiply operation. Because our data formats are 16 bits and C6416 has 6 ALUs, 4 multiply operations and 6 additions can be executed simultaneously. The CP correlation in the first step calculates a total of 256 samples times another 256 samples separated by 2048 samples. It is the maximum count condition. The CP correlation needs 256 complex number multiply operations, and 255 complex number add operations. The number of real multiply operations is $256 \times 4 = 1024$, and the number of real addition operations is $256 \times 2 + 255 \times 2 = 1022$. The minimum required number of execution cycles is

```
1 //=====================================================================================\\
2 // CP correlation code, shifted buffer                                                 \\
3 //=====================================================================================\\
4 for(i=0;i<CP_downsampling_samples;i++)   {
5         *CP_real=*CP_real+
6                 sync_buffer_1_real[i]*sync_buffer_1_real[i+2048]+
7                 sync_buffer_1_imag[i]*sync_buffer_1_imag[i+2048];
8         *CP_imag=*CP_imag+
9                 sync_buffer_1_real[i]*sync_buffer_1_imag[i+2048]-
10                sync_buffer_1_imag[i]*sync_buffer_1_real[i+2048];
11 }
12
13
14
15
16
17 //=====================================================================================\\
18 // CP correlation code, Circular buffer                                                \\
19 //=====================================================================================\\
20 for(i=0;i<CP_downsampling_samples;i++){
21        operand_location1=((cirbufptr-2048)>=0)?(cirbufptr-2048):((cirbufptr-2048)+2304*2);
22        *CP_real=*CP_real+
23                sync_buffer[cirbufptr*2]*sync_buffer[operand_location1*2]+
24                sync_buffer[cirbufptr*2+1]*sync_buffer[operand_location1*2+1];
25        *CP_imag=*CP_imag+
26                sync_buffer[cirbufptr*2]*sync_buffer[operand_location1*2+1]-
27                sync_buffer[cirbufptr*2+1]*sync_buffer[operand_location1*2];
28        cirbufptr--;
29        if(cirbufptr<0) cirbufptr=2304*2-1;     //point to the last of sync buffer;
30 }
```

Fig. 4.14: Code of CP correlation functions using shift-register buffer and circular buffer.

256. The performance of CP correlation functions are shown in Table 4.19, . After the first time calculation, the CP correlation is calculated by (2.6). The equation includes 2 complex number multiplications and 2 complex number additions. The number of real multiply operations is 8 and that of real addition operations is 8. In this stage, the absolute value should be calculated and then the maximum found. The absolute value calculation requires 2 real number multiplications and 1 real number addition. In total, the number of real multiply operations is 10 and the number of real addition operations is 5. This stage should be finishable in 3 cycles ideally.

The performance of CP correlation is shown in Table 4.19. After changing the buffer to circular buffer, the performance can be increased by unrolling the loop by hand. The efficiency of stage II is very low. In C6416 DSP, all instructions executing in parallel constitute an execute packet. An execute packet can contain up to eight instructions. Besides, the .M unit can executes dual 16×16 multiply operations simultaneously. In stage I, the software pipeline can be well scheduled, and an execute packet can contain several

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 174
;*      Loop opening brace source line : 175
;*      Loop closing brace source line : 182
;*      Loop Unroll Multiple               : 4x
;*      Known Minimum Trip Count           : 64
;*      Known Maximum Trip Count           : 64
;*      Known Max Trip Count Factor        : 64
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound      : 5
;*      Partitioned Resource Bound(*)     : 5
;*      Resource Partition:
;*                                  A-side   B-side
;*      .L units                       0        0
;*      .S units                       0        1
;*      .D units                       5*       3
;*      .M units                       4        4
;*      .X cross paths                 1        3
;*      .T address paths               5*       3
;*      Long read paths                0        0
;*      Long write paths               0        0
;*      Logical  ops (.LS)             0        0     (.L or .S unit)
;*      Addition ops (.LSD)            8        8     (.L or .S or .D unit)
;*      Bound(.L .S .LS)               0        1
;*      Bound(.L .S .D .LS .LSD)       5*       4
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 5  Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 1
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      For further improvement on this loop, try option -mh8
;*
;*      Minimum safe trip count     : 2 (after unrolling)
;*----------------------------------------------------------------------------*
```

Fig. 4.15: Software pipeline information of shift-register buffer type CP correlation loop.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line               : 201
;*      Loop opening brace source line : 201
;*      Loop closing brace source line : 209
;*      Known Minimum Trip Count         : 256
;*      Known Maximum Trip Count         : 256
;*      Known Max Trip Count Factor      : 256
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 4
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        2
;*      .S units                     1        0
;*      .D units                     2        3
;*      .M units                     1        3
;*      .X cross paths               3        1
;*      .T address paths             1        2
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          3        7     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        1
;*      Bound(.L .S .D .LS .LSD)     2        4*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 4  Did not find schedule
;*         ii = 5  Schedule found with 4 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 2
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      Minimum safe trip count     : 2
;*----------------------------------------------------------------------------*
```

Fig. 4.16: Software pipline information of circular buffer type CP correlation loop.

Table 4.19: Multiply-Add Efficiency of CP Correlation Functions

	Stage I		Stage II	
	Cycles	Multiply-Add Efficiency	Cycles	Multiply-Add efficiency
Shift-register buffer type	372	68.8%	159	1.89%
Circular buffer type without loop unrolling	1329	19.3%	174	1.72%
Circular buffer type with loop unrolling	395	64.8%	177	1.69%

```
1 //=====================================================================================//
2 // CP correlation code, circular buffer, loop unrolling 4 times                        //
3 //=====================================================================================//
4 for(i=0;i<CP_downsampling_samples/4;i++){
5         operand_location1=((cirbufptr-2048)>=0)?(cirbufptr-2048):((cirbufptr-2048)+2304*2);
6         *CP_real=*CP_real+
7                   sync_buffer[cirbufptr*2]*sync_buffer[operand_location1*2]+
8                   sync_buffer[cirbufptr*2+1]*sync_buffer[operand_location1*2+1];
9         *CP_imag=*CP_imag+
10                  sync_buffer[cirbufptr*2]*sync_buffer[operand_location1*2+1]-
11                  sync_buffer[cirbufptr*2+1]*sync_buffer[operand_location1*2];
12        cirbufptr--;
13        if(cirbufptr<0) cirbufptr=2304*2-1;     //point to the last of sync buffer;
14        i++;
15
16        .......
17        .......                                 // Code rewrite 3 times
18        .......
19 }
```

Fig. 4.17: Hand-unrolled code of circular buffer type CP correlation.

instructions. But in stage II, multiplications and additions that we used to measure the computing power of DSP are minor portion of CP correlation operations. The most computing power in stage II is contributed to handle that the operations of program branches into CP correlation function. Hence the code efficiency that obtained by caculating multiplications and additions is unreasonable in this stage. This is why the efficiency shown in stage II is much lower than in stage I, as Table 4.19 shows.

4.2 Performance Discussion

Fig. 4.19 shows that the performance of synchronization function is much better after the code is refined. The sub-functions of framing, de_framing, modulation, and de_modulation are not optimized in this thesis. The critical factor that affects the performance of these functions is not complexity of arithmetics. In this system, the input data of these functions are read from files, and the results are output to files. We use the library functions $fread$ and $fwrite$ functions to access the data. This is not good in terms of speed because the $fread$ and $fwrite$ function calls are very slow on the DSP card. If we want improve the system efficiency, then the data interface should be replaced by a more efficient method.

The profile of refined code is shown in Table 4.20. The performance of pilot correlation function is much enhanced after we refined the code. The performance of initial

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line               : 202
;*      Loop opening brace source line : 202
;*      Loop closing brace source line : 238
;*      Known Minimum Trip Count         : 16
;*      Known Maximum Trip Count         : 16
;*      Known Max Trip Count Factor      : 16
;*      Loop Carried Dependency Bound(^) : 17
;*      Unpartitioned Resource Bound     : 14
;*      Partitioned Resource Bound(*)    : 15
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        8
;*      .S units                     1        0
;*      .D units                    15*      12
;*      .M units                     8        8
;*      .X cross paths               9        3
;*      .T address paths             8        7
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)         22       24     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        4
;*      Bound(.L .S .D .LS .LSD)    13       15*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 17 Did not find schedule
;*         ii = 18 Register is live too long
;*         ii = 18 Did not find schedule
;*         ii = 19 Register is live too long
;*         ii = 19 Did not find schedule
;*         ii = 20 Register is live too long
;*         ii = 20 Schedule found with 2 iterations in parallel
;*      done
;*
;*      Collapsed epilog stages     : 1
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      Minimum safe trip count     : 1
;*----------------------------------------------------------------------------*
```

Fig. 4.18: Software pipline information of hand-unrolled circular buffer type CP correlation loop.

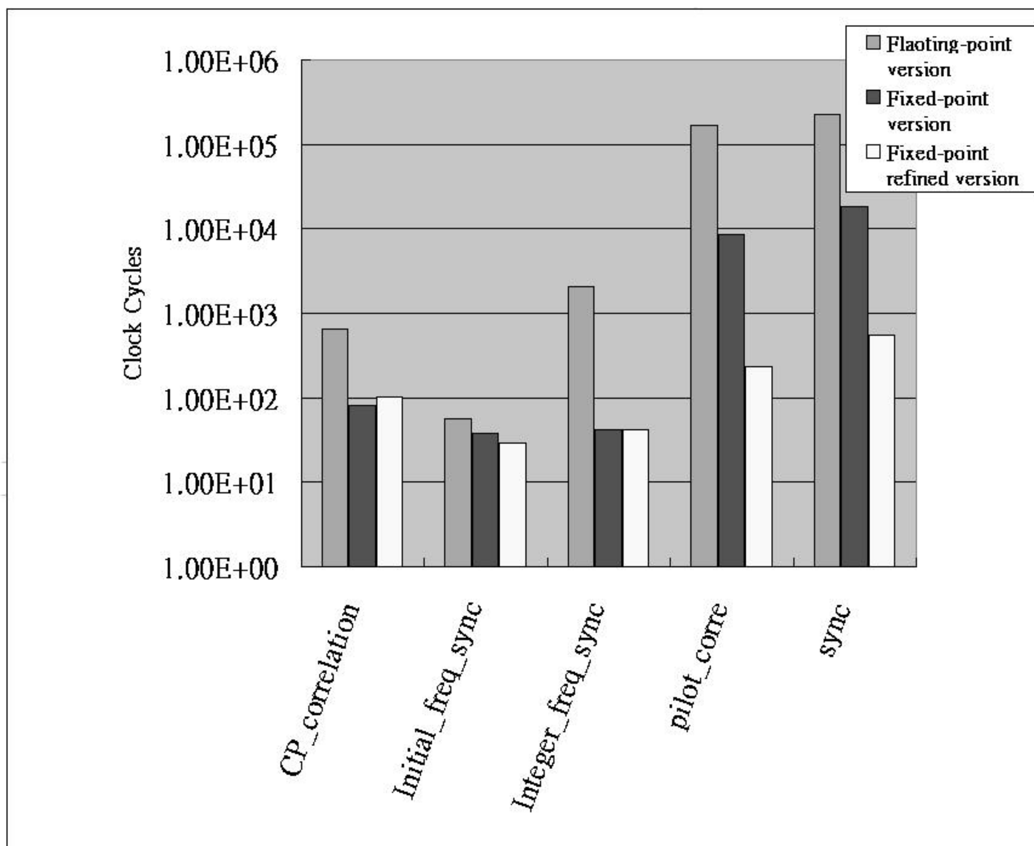


Fig. 4.19: Execution cycles of synchronization functions.

frequency synchronization is little enhanced by using intrinsics. The efficiency of CP correlation is decreased after refining the code because of the conditional operations are added in the correlation loop after the buffer type is modified to circular buffer. We have maximized the efficiency of CP correlation by unrolled the loop by hand. In spite of the efficiency of CP correlation is decreased, the efficiency of synchronization top-level function sync is much increased after we refine the code. That is, the overall efficiency is much increased. Recall that the clock cycle counts of real time requirement is 120960 in one symbol duration and 52.5 in on sample duration. The overall synchronization function does not meet the requirement when executing on one DSP chip. Table 4.21 shows the estimated performance of bottleneck functions of synchronization. The initial condition of CP correlation is the first time that we find the symbol start time. Because the next symbol start time is about 2048 away from the current symbol start time, we do not need to do CP correlation immediately. We can start the CP correlation at location that 256 samples before the possible start time of the next symbol. Hence performance of CP correlation in tracking stage meets the real time requirement. The reason why complexity of pilot correlation in initial condition is more complex than in tracking condition is given before. The performance of overall synchronization reachs real time requirement of 8.96% in initial condition and 24% in tracking condition. Hence the real time requirement can not be fulfilled in both condition. To meet the real time requirement, we can partition the synchronization function into sub-functions that are either executed on more DSPs or implemented on FPGA.

Table 4.20: Profile of Refined Code of 802.16a DL Receiver Function Blocks

	Code Size (Bytes)	Avg. Count (Cycles)	Improvement (compare with floating-point operations	Improvement (compare with fixed-point operations before refinement)	Real time rate
SRRC_downsample	700	1301	93.87%	0%	4.04%
CP_correlation	1320	101	84.34%	-26.25%	51.98%
initial_freq_sync	300	29	49.12%	23.68%	181.03%
integer_freq_sync	932	42	97.98%	0%	125.00%
pilot_corre	2824	234	98.56%	97.23%	22.44%
sync	784	560	99.76%	96.91%	9.38%
FFT	276	32259	99.86%	0%	374.97%
de_framing	988	1225991	24.61%	0%	9.87%
de_modulation	460	528672	60.92%	1.71%	22.88%

1896

Table 4.21: Performances Estimation in Separate Initial and Tracking Condition

	Initial Avg.		Tracking Avg.	
	Clock Cnt.	Real time rate	Clock Cnt.	Real time rate
CP_correlatior	177	29.66%	39	133.15%
pilot_corre	544	9.60%	103	50.97%
sync	585	8.96%	215	24.42%

Chapter 5

Conclusion and Future work

5.1 Conclusion

We considered implementation of 802.16a TDD OFDMA downlink synchronization techniques on TI's C6416 digital signal processor. The complete TDD OFDMA DL system is also implemented for verifying the accuracy. The implementation is based on the simulation program from [1], which is floating-point version. We modified the original program to fixed-point version and increased the efficiency of synchronization code.

The scheme that we used divides DL synchronization into four stages [1], which are symbol time synchronization, fractional frequency synchronization, integer frequency synchronization and frame synchronization. The recommended data type for fixed-point multiplication on TI's C6000 DSP is 16-bit [10] because this data type provides the most efficient use of 16-bit multiplication in the C6000. Hence the data format that we used in DL synchronization is Q.15 which is 16-bit fixed-point data format. The precision of using Q.15 data format in DL synchronizaiton was proved enough in this thesis. To increase the efficiency of synchronization code, we used intrinsics to replace the inefficient function calls, modified the shift-register buffers to circular buffers for simplifying data buffer operations, stored the pilot location in memory to simplify the pilot correlation loop, and unrolled the loop in CP correlation function to make the software pipeline be well scheduled. We also used the FFT/IFFT functions which had been optimized by TI from TI's DSP library to increase the execution efficiency.

After the optimization, the efficiency of CP correlation was increased 84.34%, fractional synchronization was increased 49.12%, integer synchronization was increase 97.98%, pilot correlation (frame synchronization) was increase 98.56%, and the overall synchronization function was increased 99.76%. The individual function such as fractional frequency synchronization, integer frequency synchronization, and FFT has met the real time operations requirement. The execution speed of total synchronization function is 9.38% of real time requirement. We estimated the performances in initial condition and tracking condition. The performance of overall synchronization reaches real time requirement of 8.96% in initial condition and 24% in tracking condition. The execution speed in tracking condition is better than in initial condition, but the real time requirement is not fulfilled in tracking condition.

5.2 Potential Future Work

We have been optimized the efficiency of arithmetic functions of synchronization. The real time requirement can not be fulfilled after the optimization. To fulfilled the real time requirement, we can make more effort to improve the synchronization program. One way we can consider is that skipping a function call when it is idle operation. In most time, program enters and then exits synchronization function without doing anything. In tracking condition, the idle time of synchronization operation can be predited. We can let the program skip the idle synchronization function to save the time of branching. The other way is increasing the efficiency of doing a function call. There are quite a few of delivered parameters in synchronization function and its sub-functions. The more parameters delivered by a function the more stack operations in the branching process, and the less efficiency of a function call. If the real time requirement can not be fulfilled after all the optimization methods are applied, We must partition the synchronization function into sub-functions that are either executed on more DSPs or implemented on FPGA to meet the real time requirement.

Bibliography

[1] 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.

[2] J. J. van de Beek *et al.*, "ML estimation of time and frequency offset in OFDM systems," *IEEE Trans. Signal Processing*, vol. 45, no. 7, pp. 1800–1805, July 1997.

[3] P. H. Moose, "A technique for orthogonal frequency-division multiplexing frequency offset correction," *IEEE Trans. Commun.*, vol. 42, no. 10, pp. 2908–2914, Oct. 1994.

[4] Y.-L. Huang, C.-R. Sheu, and C.-C. Huang, "Joint synchronization in Eureka 147 DAB system based on abrupt phase change detection," IEEE *J. Select. Areas commun.*, vol.17, no.10,Oct 1999.

[5] 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.

[6] Texas Instruments, *TMS320C64x Technical Overview*. Literature number SPRU395B, Jan. 2001.

[7] Texas Instruments, *TMS320C6000 DSP Peripherals Overviews Reference Guide*. Literature number SPRU190F, Apr. 2004.

[8] J. J. van de Beek, P. O. Borjesson, M. L. Boucheret, D. Landstrom, J. M. Arenas, P. Odling, C. Ostberg, M. Wahlqvist, and S. K. Wilson, "A time and frequency synchronization scheme for multiuser OFDM," *IEEE J. Select. Areas Commun.*, vol. 17, pp. 1900–1914, Nov. 1999.

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

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

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

[12] M. E. Frerking, *Digital Signal Processing in Communication Systems*, Van Nostrand Reinhold, 1994.

[13] ETSI SMG, "Overall requirements on the radio interface(s) of the UMTS," Technical Report ETR/SMG-21.02, v.3.0.0., ETSI, Valbonne, France, 1997.

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

[15] Innovative Integration, *Quixote Data Sheet*, http://www.innovative-dsp.com/support/datasheets/quixote.pdf.

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

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