國 立 交 通 大 學

電子工程學系　　電子研究所碩士班

碩士論文

IEEE 802.16a 分時雙工正交分頻多重進接下行傳收系統之數位訊號處理器軟體實現與整合

DSP Software Implementation and Integration of IEEE 802.16a TDD OFDMA Downlink Transceiver System

研究生:　陳昱昇

指導教授:　林大衛 博士

中華民國九十四年六月

IEEE 802.16a 分時雙工正交分頻多重進接下行傳收系統之數位訊號處理器軟體實現與整合

DSP Software Implementation and Integration of IEEE 802.16a TDD OFDMA Downlink Transceiver System

研究生：陳昱昇　　Student: Yu-Sheng Chen

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

國 立 交 通 大 學

電子工程學系　電子研究所碩士班

碩士論文

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

中華民國九十四年六月

IEEE 802.16a 分時雙工正交分頻多重進接下行傳收系統之數位訊號處理器軟體實現與整合

研究生: 陳昱昇　　指導教授:林大衛 博士

國立交通大學電子工程學系　　電子研究所碩士班

摘要

我們在此論文中介紹 IEEE 802.16a 分時雙工正交分頻多重進接之下行傳收系統。 傳收系統包含了在數位訊號處理器上實現發射端、同步裝置、通道狀態估測器和其他接收端功能，以及在電腦主機上實作通道模擬器來模擬多路徑衰減、外加白色高斯雜訊以及頻率偏移等通道效應。下行同步技術包含了符元(symbol)開始時間、頻率偏移和資料訊框(frame)之估測。我們使用德州儀器(TI)所製造的數位訊號處理器。此處理器的操作平台為 Innovative Integration 公司製名為 Quixote 的 cPCI 卡。

程式主要都是用 16 位元(bit)的定點(fixed point)格式來完成。我們藉著改變程式編碼的風格(coding style)以及 C6416 本身具有的指令來改進程式執行的效能，並把執行效能拿來跟能否達到即時運算的要求做比較以及分析。此外，我們還在電腦主機上做了一個用來在螢幕上監控同步裝置以及通道狀態估測器的圖形介面。 我們發現若要整個系統都達到即時運算的要求就需要把各個功能都分割到多顆數位訊號處理器上來實現。

DSP Software Implementation and Integration of IEEE 802.16a TDD OFDMA Downlink Transceiver System

Student: Yu-Sheng Chen Advisor: Dr. David W. Lin

Department of Electronics Engineering
Institute of Electronics
National Chiao Tung University

Abstract

This thesis presents an implementation of IEEE 802.16a TDD OFDMA DL transceiver system, which includes the implementation of transmitter, synchronizer, channel estimator, and other receiver functions on the DSP baseboard and channel simulator, which simulates multipath fading, AWGN and frequency offset, on host PC. The DL synchronization includes the estimations of symbol timing, frequency offset, and frame lock status. The implementation employs Texas Instruments' TMS320C6416 DSP chip housed on Innovative Integration's Quixote cPCI card.

The program is mainly implemented by 16-bit fixed point data format. Performances of the programs are analyzed and improved by changing the coding style and applying intrinsic function of C6416 DSP. The execution performances are compared to the real-time requirement. Besides, we also implement a host graphical interface which can monitor the synchronization and channel estimation results on the screen. We find that we may need to separate the functions into multi-DSPs to achieve the real-time of the overall system.

誌謝

誠摯的感謝我的指導老師林大衛博士這兩年多來的指導，老師對我的指導不僅僅只是在學識的指導，在研究方法以及學習態度上，給我的獲益更是難以估計。在通訊領域知識的學習上，林老師給我的只是個開頭，讓我知道還有許許多多的方向值得去研究。我感到非常榮幸可以成為林老師的學生誠摯的感謝我的指導老師林大衛博士，由衷的感謝老師的指導。

另外，我還要感謝這個像個大家庭似的實驗室，實驗室豐富的資源讓我們有最佳的學習環境，感謝博士班學長崑健、俊榮在學習過程中給予的許多建議以及幫助，感謝景中、汝芩、志凱、鎮宇等同學彼此間的砥礪以及幫助，有大家一起努力才有這篇論文。

最後，我要感謝我最愛的家人，有你們長久來一直對我的支持是我學習、成長最大的動力，有你們一路陪伴和幫助讓我在求學過程沒有後顧之憂。

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years there has been increasing interest in wireless technologies for subscriber access. For some years much interest has been devoted to fixed wireless access. To provide a standardized approach, the IEEE 802 committee set up the 802.16 working group in 1999 to develop broadband wireless access standards [24].

The IEEE 802.16 standards are concerned with the air interface between a subscriber's transceiver station and a base transceiver station. One IEEE 802.16 Task Group [24] developed the IEEE Standard 802.16a that amends IEEE Std 802.16-2001 by enhancing the medium access control (MAC) layer and providing additional physical layer specifications in support of broadband wireless access at frequencies 2–11 GHz. After 802.16-2001, a new IEEE Std 802.16-2004 (also called 802.16) has been published and the IEEE 802.16e is near completion. In the physical layer of the 802.16, the main differences between 802.16 and 802.16a are as follows:

- The preamble allocation of the TDD (time division duplexing) frame structure.
- The usage of subchannels in the symbol structure.
- Forward error correction code.

Details can be found in [3] and [4]. The IEEE 802.16e adds mobile extension to the 802.16 standard.

In this thesis, we consider the DSP software implementation of IEEE 802.16a downlink system. The reason that we consider the now defunct IEEE 802.16a rather than the current IEEE 802.16-2004 is because this project was started three years ago. We will consider newer 802.16 standards in the future. The synchronization techniques are modified from [2]. The implementation employs Texas Instrument's TMS320C6416 digital signal processor (DSP) housed on Innovative integration's Quixote cPCI card.

This thesis is organized as follows. In chapter 2, we introduce the 802.16a downlink system specification and the synchronization techniques. Chapter 3 introduces the Quixote baseboard and the TMS320C6416 DSP chip, as well as the program development environment and the host-target communication mechanism. In chapter 4, we describe the DSP implementation and examine the program efficiency. We also introduce the user interface to control program execution and display numerical results results. Finally, chapter 5 gives the conclusions and points out some potential future work.

Chapter 2

IEEE 802.16a Transmission Techniques

The IEEE 802.16a specification enhances the medium access control layer of the IEEE 802.16-2001 standard and its operating frequencies are between 2 to 11 GHz. There are three physical layer modes in 802.16a: SCa (single carrier a), OFDM (orthogonal frequency-division multiplexing), and OFDMA (orthogonal frequency-division multiple access). We consider OFDMA, as it is a technology of considerable research potential.

In this chapter, we first introduce the OFDMA specifications in 802.16a and then explain the approaches we take to implement the transceiver system. Finally , we introduce the sparse DFT algorithms and discuss the reason that we do not adopt the transform decomposition method.

2.1 Overview of the IEEE 802.16a TDD OFDMA Downlink System [3]

Before a detailed introduction to IEEE 802.16a standard, we explain some frequently used terms first. The direction of transmission from the base station (BS) the subscriber station (SS) is called downlink (DL), and the opposite direction from SS to

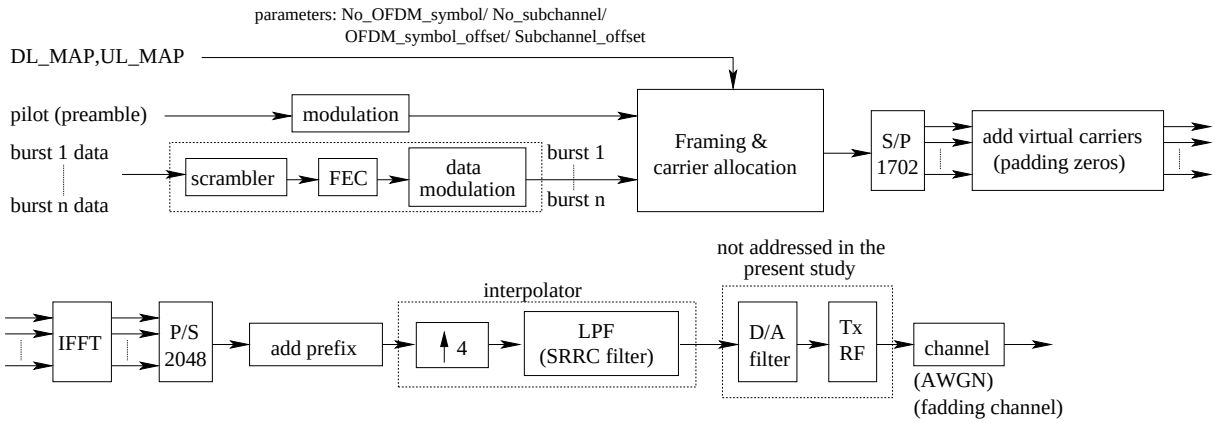

Fig. 2.1: DL transmitter structure (from [1]).

BS is called uplink (UL). The medium access control layer is used to provide the system grant/request access and the link of data between the upper layer and the lower layer (i.e., physical layer). The physical layer (PHY) handles the data transmission and may include use of multiple transmission technologies, each appropriate to a particular frequency range and application.

2.1.1 Transceiver System Structure [2]

The structure of the DL transmitter is shown in Fig. 2.1. The data bursts are fed into the FEC (forward error correction) encoder. Then we apply modulation and framing. Gray-mapped QPSK and 16-QAM are required to be supported in modulation, whereas the support of 64-QAM is optional. The framing is used to arrange the coded data, MAPs, pilots and preamble according to the specified frame structure and carrier allocation. After framing, the data are fed into IFFT with some null carriers (guard band) to obtain the time domain signal through IFFT. The result from IFFT is output sequentially to the pulse shaping filter. As the ideal lowpass interpolation filter cannot be implemented exactly, the square root raised cosine filter is used instead. The impulse response of the filter is given by

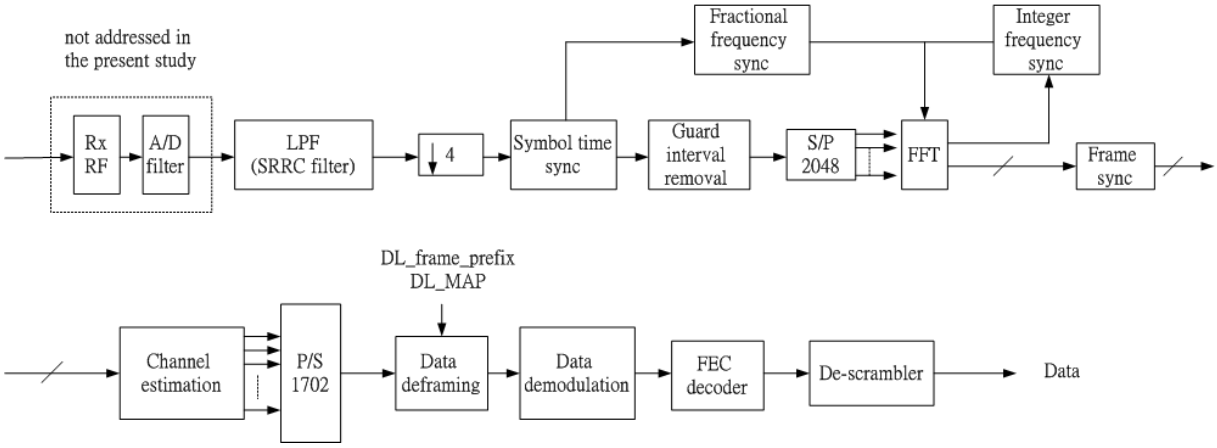

Fig. 2.2: DL receiver structure (modified from [1]).

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

where α is the roll-off factor. The D/A and RF parts are not addressed in the present study.

Fig. 2.2 shows the downlink receiver structure. The receiver is in some sense the reverse of the transmitter, except for the synchronizer and the channel estimator. The synchronizer is a major focus in this thesis, and it will be discussed in more detail later.

2.1.2 Downlink Carrier Allocation [3]

In the 802.16a OFDMA system, there are 2048 carriers per symbol. The carriers are divided into three groups: pilot carriers for synchronization and channel estimation purposes, data carriers for data transmission, and null carriers that are used for guard band and the DC carrier which transmits nothing at all. And the system parameters employed in this study are shown in Table 2.1.

As we can see in Fig. 2.3, there are 1702 used subcarriers, composed of 1536 data carriers and 166 pilot carriers. The remaining subcarriers are unused subcarriers as

Table 2.1: 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)

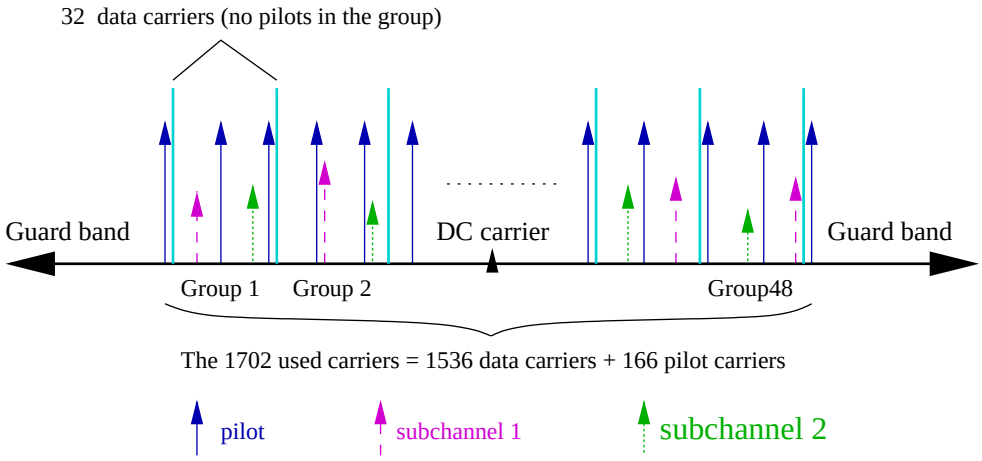

Fig. 2.3: Illustration of carrier usage in OFDMA DL (from [1]).

Table 2.2: OFDMA Carrier Allocation

Parameter	DL 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	166
Number of data carriers	1536
$N_{subchannels}$	32
$N_{subcarriers}$ per subchannel	48
Number of data carriers per subchannel	48

guard bands distributed on the edge of the symbol, and one DC carrier right in the middle of the OFDMA symbol. In the downlink, the pilot subcarriers are allocated first, and then the remainders of the used carriers are divided into 32 subchannels, each subchannel consisting of 48 data carriers. The pilot locations change with time according to some permutation formula which will be described below. Table 2.2 shows the OFDMA downlink carrier allocation.

There are variable location pilot carriers and fixed-location pilot carriers. The carrier indices of the fixed-location pilots never change. The variable location pilots shift their locations every symbol periodically every 4 symbols, according to the formula $varLocPilot_k = 3L + 12P_k$, where $varLocPilot_k$ is the carrier index of a variable location pilot, L periodically takes the values 0,2,1,3, cyclically over the symbols, and $P_k = \{0, 1, 2, 3, \ldots, 141\}$. The detailed illustration is given in Fig. 2.4.

After mapping the pilot carriers, we should also map the data carriers to the correct positions. Note that since the variable location pilots change their locations

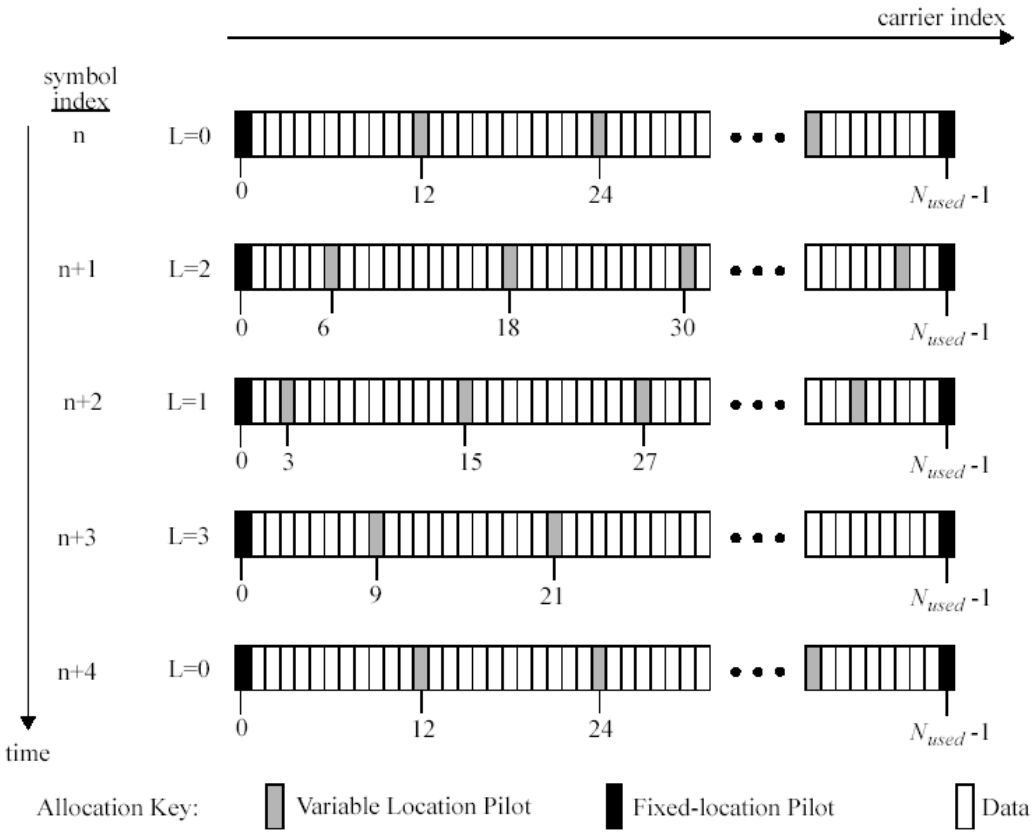

Fig. 2.4: Pilot allocation in the OFDMA DL (from [3]).

with symbols, the locations of the data carriers change also.

The exact partitioning into subchannels is done according to the formula below, called a permutation formula:

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

where

- $carrier(n,s)$ is the carrier index of carrier n in subchannels,
- s is the index number of a subchannel, from the set $[0, 1, ..., N_{subchannels} - 1]$,
- n is the carrier-in-subchannel index from the set $[0, 1, ...N_{subchannels} - 1]$,
- $N_{subchannels}$ is the number of the sunchannels,
- $p_s[j]$ is the series obtained by rotating PermutationBase 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 MAC to identify this particular BS, and
- $X_{mod(k)}$ is the remainder of quotient X/k.

The following text in this section is mainly taken from [3], [2] and [1].

2.1.3 OFDMA TDD Frame Structure [3]

According to IEEE 802.16a, the duplexing method in the 2–11 GHz band shall be either FDD (frequency division duplexing) or TDD (time division duplexing) in licensed band and TDD in license-exempt bands. We consider the TDD mode in this thesis. The advantage of using TDD is that we have flexibility to control the DL and UL traffic ratio.

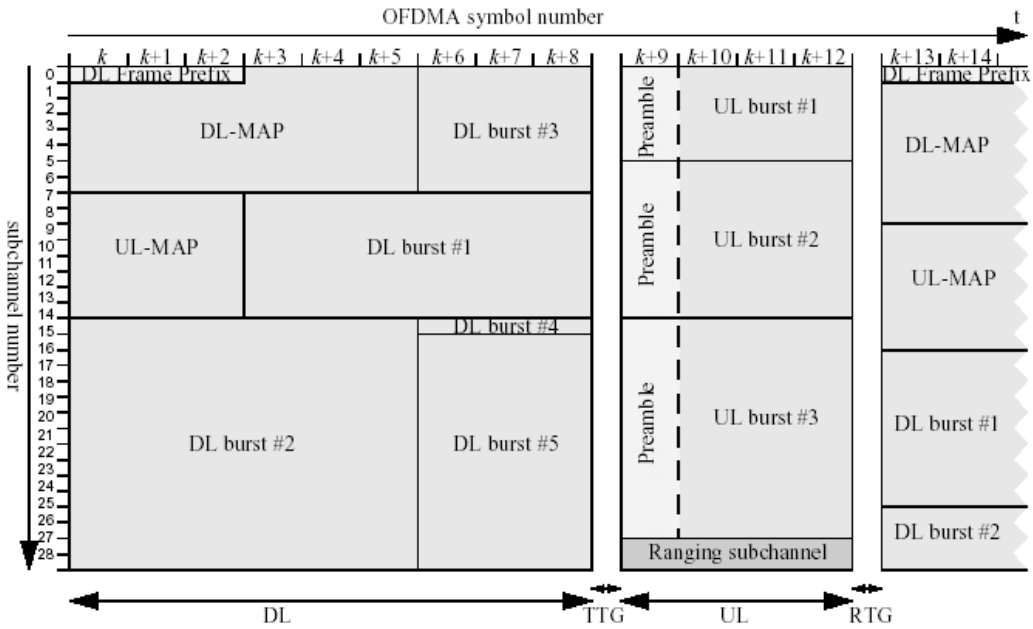

Fig. 2.5: Frame structure of the TDD OFDMA system (from [3]).

The frame structure of TDD OFDMA is as shown in Fig. 2.5. The data are segmented into blocks for FEC (forward error correction) coding. Each FEC block spans one OFDMA subchannel in the subchannel axis and three OFDMA symbols in the time axis. A frame consists of one DL subframe and one UL subframe. The duration of a frame can run from 2 to 20 ms and is specified by the frame duration code. A subframe contains several transmission bursts, which are composed of multiple FEC blocks. In each subframe, the TTG (Tx/Rx transition gap) and RTG (Rx/Tx transition gap) are inserted between the downlink and uplink transmissions at the end of each frame respectively to allow the BS and SS to turn around. TTG and RTG shall be at least 5 μs and an integer multiple of four samples in duration. For the DL, the transmitted data from the BS should contain the control message and 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, forward error correction 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 (Downlink Channel Descriptor) and UCD (Uplink Channel Descriptor) messages. The DL-MAP/UL-MAP messages define the access to the DL/UL information, including the burst profiles and the distributions of the subchannels and time axes of the bursts. The DCD and UCD shall be transmitted by the BS at a periodic interval to define the characteristics of DL and UL physical channels. The pilots of the first OFDM symbols is the DL preamble in the sense that they indicate where the OFDMA frame starts. Note that the DL preamble is not composed of an all-pilot symbol, so no additional OFDM symbol is transmitted. As a result, the number of OFDM symbols of the DL is $3N$, where N is a positive integer. And the number of UL OFDM symbols is $3N + 1$, including one preamble and subsequent data symbols.

2.1.4 Modulation [3]

There are three types of information to be modulated: data, pilot, and preamble. The modulation of pilot and preamble will be explained in detail for they are useful in synchronization.

Data Modulation

The data modulation in 802.16a is 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.

Pilot Modulation

Pilot carriers shall be inserted into each data burst in order to constitute the symbol and they shall be modulated according to their carrier locations within the OFDMA

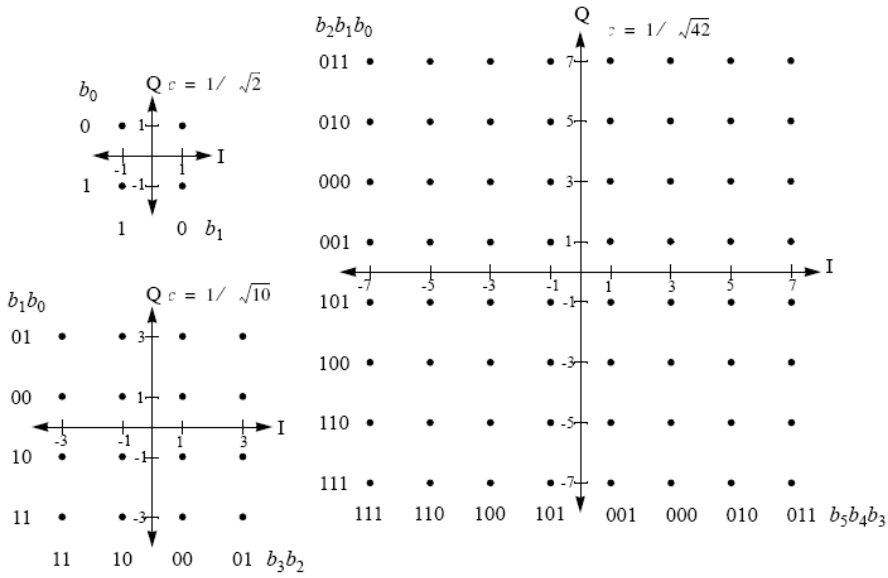

Fig. 2.6: QPSK, 16-QAM and 64-QAM constellations (from [3]).

symbol. The PRBS 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 initialization vector of the PRBS in the DL transmission is [11111111111] except for the OFDMA DL PHY preamble. For the UL, the initialization vector of the PRBS is [10101010101]. 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 boost of 2.5 dB over the average power of each data tone. The pilot carriers shall be modulated according to the following formulas:

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

Preamble Modulation

The first three symbols of a frame serve as the OFDMA DL preamble. For the DL preamble, the initialization vector of the pilot modulation PRBS is [01010101010].

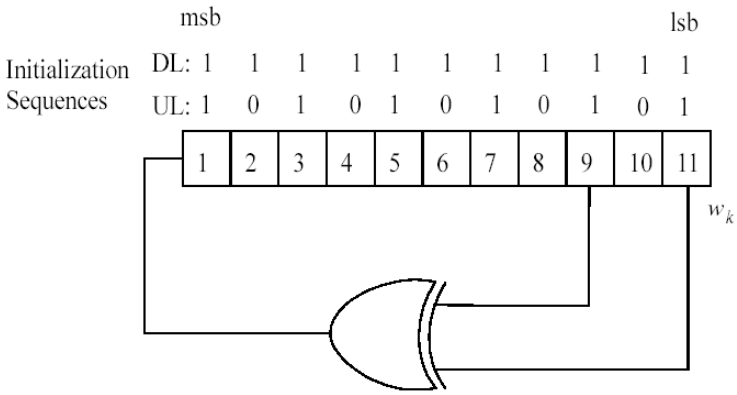

Fig. 2.7: Pseudo Random Binary Sequence (PRBS) generator for pilot modualtion (from [3]).

Hence, the preamble and other symbols may have the same pilot locations, but they can be recognized by different modulation values. The pilots shall be boosted and shall be modulated according to the following formulas:

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

For the UL preamble, all the used carriers are pilots. The initial vector of the PRBS is the same as the normal UL pilot modulation. The pilots shall not be boosted and is modulated as

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

2.2 Approach to Downlink Synchronization

Synchronization errors in OFDM can cause intersymbol and intercarrier interference. Accurate demodulation and detection of an OFDM signal requires carrier orthogonality. One way to suppress these interferences in OFDM systems is to track the carrier frequency of the received signal and the start time of each OFDM symbol. A blind joint maximum likelihood estimator of symbol time and carrier frequency offset for OFDM symbols using cyclic prefix is presented in [7]. The estimator exploits

the redundancy introduced by the prefix and is independent of how the subscribers are modulated. Therefore, it does not require extra pilot information to complete the timing and fractional frequency synchronization.

Variations of carrier oscillator, sample clocks or the symbol time affect the orthogonality of the OFDM system. In this thesis, we do not consider sample clock synchronization. The sample clocks of the users and the base station are assumed to be fully synchronized. The timing requirement is relaxed by using cyclic prefix (CP). 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 phase shift of the demodulated data symbols across the carriers but will not result in intersymbol interference (ISI) or intercarrier interference (ICI).

In practical OFDM systems, frequency offsets due to oscillator mismatch usually exist between transmitters and receivers. Each subcarriers can be assumed equally affected by a center carrier frequency shift, because the system bandwidth is small compared to the center carrier frequency. The frequency offset has three effects: reducing the amplitude of the FFT output, introducing ICI from other carriers, and introducing a common phase rotation of the subcarriers [9].

2.2.1 Downlink Synchronization Requirements

The DL synchronization can be divided into two conditions. One is for the establishment of the initial connection, called the initial synchronization. The other is the tracking of the synchronization, called the normal synchronization. The main reason to have a different normal synchronization than initial synchronization is to reduce the computational complexity in normal operation. In fact, we use a simplified version of the initial synchronization procedure for normal synchronization (tracking) purpose.

If a subscriber wants to join the transmission network for the first time, it has no idea about the timing of the network and the frequency offset with the BS. In this case, after detecting the symbol start time, frequency estimation and correction is needed. 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. Then, the SS has to check that the received OFDM symbol is from the BS or from other SSs. If the symbol is from the BS, further check is required to know whether this symbol is the start of a frame. After initial synchronization, the subscriber is able to extract the transmission parameters from the DL_MAPs and UL_MAPs. With these parameters, the SS can roughly predict the next symbol and frame start times, so normal timing synchronization can be simplified. The frequency offset is tracked during normal operation. If the OFDM symbol start time is out of the predicted range, re-initial synchronization is needed.

There are three kinds of useable information for synchronization: guard interval, pilot carriers (including preamble), and the guard bands. We employ the method proposed in [1] and divide the initial DL synchronization into 4 stages. In the first two stages, the OFDM symbol start time and the fractional frequency offset are detected using the guard interval. The third stage exploits the guard bands to correct integer frequency offset. Then, the final stage checks the pilot and preamble information to determine when a frame starts. For normal synchronization, only two stages are needed, where stage I is the same as that in initial DL synchronization and stage II is used to track the frequency. More detailed description of the synchronization technique is given below.

2.2.2 Procedure of Initial Downlink Synchronization

2.2.2.1 Stage I: Symbol Timing Synchronization

In [1], two methods of symbol timing estimation have been considered, both using the cyclic prefix: ML estimation and CP correlation. The method of ML estimation is proposed in [7], which uses the maximum likelihood criterion to estimate time and frequency offsets. Under the assumption that the received samples are jointly Gaussian, the estimated symbol time offset $\hat{\theta}$ is given by

$$\hat{\theta} = \arg\max \left\{ |\Gamma(\theta)| - \rho\Phi(\theta) \right\}, \tag{2.2.1}$$

where

$$\Gamma(\theta) = \sum_{k=\theta}^{\theta+L-1} r(k)r^*(k+N), \tag{2.2.2}$$

$$\Phi(\theta) = \frac{1}{2} \sum_{k=\theta}^{\theta+L-1} |r(k)|^2 + |r(k+N)|^2, \tag{2.2.3}$$

and $\rho = \frac{SNR}{SNR+1}$ with SNR being the 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, the CP correlation method uses only the correlation part to estimate the symbol time, ignoring the part that compensates for the difference in energy in the correlated samples. 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

$$\hat{\theta} = \arg\max \left| \sum_{k=\theta}^{\theta+L-1} r(k)r^*(k+N) \right|. \tag{2.2.4}$$

A comparison of the complexity difference between the two methods is given in [2]. For further reduction of the CP correlation complexity, we can compute the CP correlation at sample time θ by (2.2.2), then the CP correlation at sample time θ+1

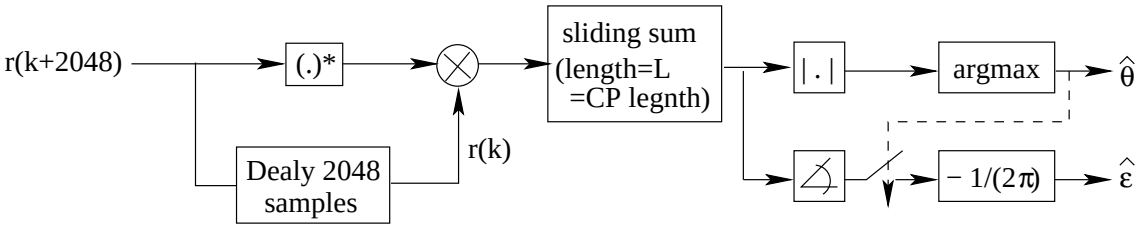

Fig. 2.8: Structure of the symbol time and frequency estimator (from [1]).

is given by

$$\begin{aligned}\Gamma(\theta+1) &= \sum_{k=\theta+1}^{\theta+L} r(k)r^*(k+N) \\ &= \Gamma(\theta) - r(k)r^*(k+N) + r(\theta+L)r^*(\theta+L+N). \qquad (2.2.5)\end{aligned}$$

Reference [1] shows that although the performance of ML estimator algorithm is better than that of CP correlation algorithm in AWGN channels, neither algorithm can estimate the exact symbol time at 100% accuracy. In addition, for fading multipath channels the CP correlation algorithm can outperform the ML estimator algorithm. To estimate the exact symbol time, both algorithms should be assisted by other means to find the symbol time more accurately. Here pilot correlation is used as the auxiliary operation, which is combined in stage IV with frame synchronization. Since the complexity of ML estimation is much higher than that of CP correlation, but the benefit is questionable [1], [2], we use CP correlation to estimate the symbol time in stage I. The algorithm structure is as shown in Fig. 2.8.

2.2.2.2 Stage II: Fractional Frequency Synchronization

The ML estimator of the fractional frequency offset $\hat{\epsilon}$ is given by [7], [8]

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

whose structure is already shown in Fig. 2.8. It is easy to understand why ϵ can be estimated by this method. The frequency offset ϵ results in an exponential

modulation in the time domain, in that the received 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 $r^*(k+N)$ yields

$$r(k)r^*(k+N) = s(k)s^*(k+N)e^{-j\frac{2\pi(\epsilon+N)}{N}} + \text{noise}.$$

Note that $e^{-j\frac{2\pi(\epsilon+N)}{N}}$ is the common factor of all the pairwise sample products for $r(k)$ in the guard interval. Hence the sum of these products should reduce the noise effect. The frequency offset ϵ can be estimated by the phase of the sum of $r(k)r^*(k+N)$ taken at the symbol start position. Note that the phase contribution of any integer frequency offset is an integer times 2π. Thus this estimator is merely able to detect fractional frequency offset.

2.2.2.3 Stage III: Integer Frequency Synchronization

The integer frequency synchronization stage is performed after FFT by utilizing the guard band and two fixed pilot carriers which are at the edge of the used carriers to correct the frequency offset. There are two reasons to using the guard band to do integer frequency synchronization. First, guard carriers suffer less degradation from by ICI than pilot carriers. Secondly, the complexity of using the guard carriers is much less than that using the pilot carriers as no multiplication is required.

The first step in integer frequency offset estimation is for SS to check whether the received OFDM symbol is from the BS rather than another SS. In 802.16a [3], the definition of the guard bands and pilots are different for DL and UL. The indices

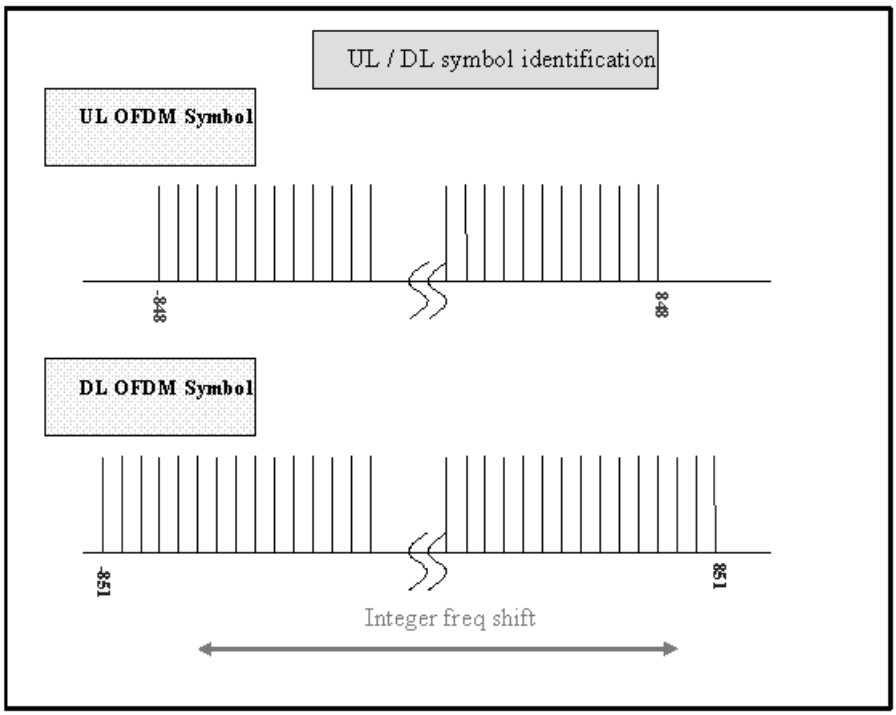

Fig. 2.9: DL/UL symbol identification (from [2]).

of the DL guard carriers are from -1024 to -852 and from 852 to 1023, while for UL they are from -1024 to -849 and from 849 to 1023. A threshold can be set and 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. 2.9.

For the DL, the standard defines that carriers -851 and 851 are 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 direction of checking is from 1023 to 852, and then from -1024 to -852. If a carrier k is detected to be larger than the threshold, the $\pm$851st fixed pilots are assumed to have shifted $k - 851$ carrier spacings due to the frequency offset. Thus the checking is stopped and the frequency is corrected by $k - 851$ carrier spacings.

In a fading channel, ICI may cause serious distortion. Thus, if the $\pm$851st pilots

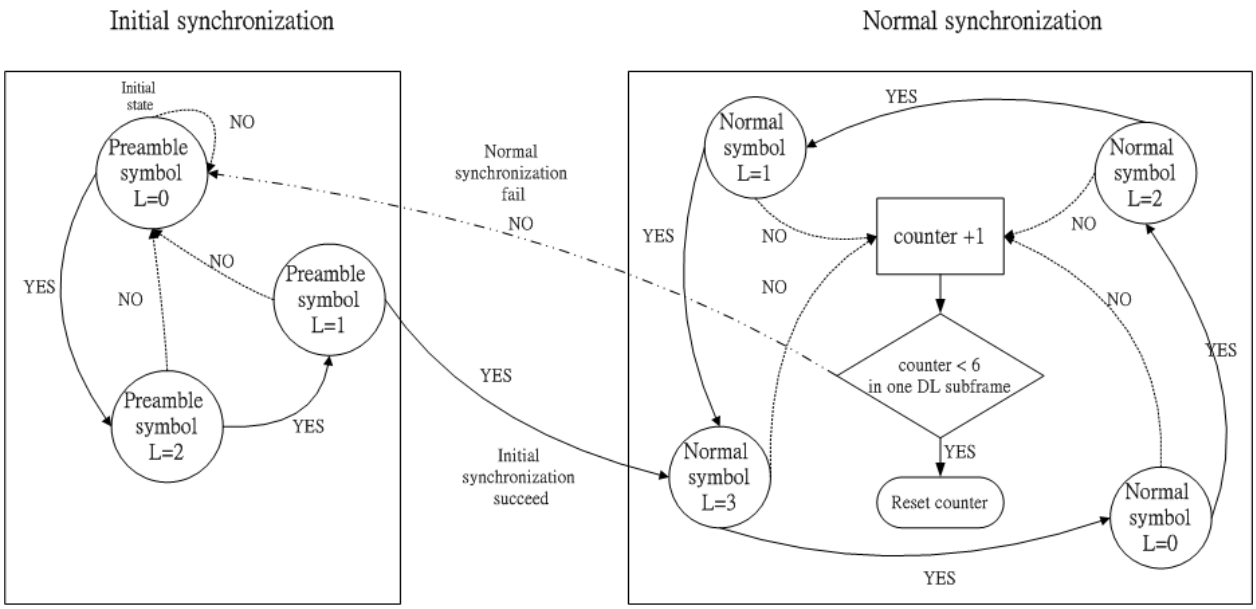

Fig. 2.10: State diagram of the frame synchronizer.

are distorted to be less than the threshold, the frequency offset will not be detected by the 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. The threshold is chosen to be 0.55 in our simulation. This value is derived from the simulation results in [1].

2.2.2.4 Stage IV: Frame Synchronization

In stage I, the OFDMA symbol start time have been roughly estimated, but the SS has to know exactly where the frame starts. The frame start time estimation proposed in [1] uses the pilot correlation method. In the 802.16a standard [3], the varible location pilots change their locations from symbol to symbol depending on the symbol index L. The modulation of pilots is decided by the PRBS generator, and the initialization vector of the PRBS generator is different in the preamble

symbol than in a non-preamble symbol. Therefore, there are 7 possible kinds of pilot structure 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.

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 proposed frame synchronization algorithm is illustrated in Fig. 2.10. In order to build connection, we have to find the starting point of a frame in initial synchronization. After finding the third preamble symbol, we can turn the operation to normal synchronization as shown in Fig. 2.10. The method presented in [2] declares frame synchronization failure when there is one unexpected symbol in pilot correlation. But we find that one unexpected symbol does not mean that it cannot find correct pilot correlation in the next symbol. So we modify the method to declaring frame synchronization failure with the detection of 6 unexpected symbols within one DL subframe.

From [2], because of the use of pilot correlation, we may need to do FFT at each sample location for a range of 65 samples (from -32 to $+32$, as shown in Fig. 2.11(b) and (c) [1]) in order not to miss the true symbol start time. In order to reduce the computational complexity, the conventional FFT is only applied at location -32. At the subsequent sample locations, the FFT may be computed recursively as

$$X_n(k) = [X_{n-1}(k) - x_{n-N} + x_n]\, e^{j\frac{2\pi k}{N}} \tag{2.2.6}$$

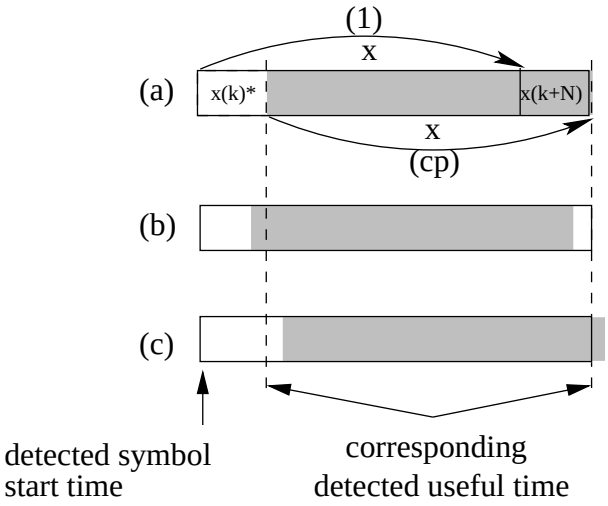

Fig. 2.11: Multiple FFTs are needed for a consecutive range of sample locations to ensure finding the true symbol start time. (a) Symbol location detected in stage I, where the gray region is the useful samples which are applied FFT. (b), (c) Leftmost and rightmost ranges of correlation, respectively. (From [1].)

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

2.2.3 Normal Synchronization

After initial synchronization, the SS can find the frame duration from the frame duration code in the MAPs. Thus the next frame start time can be predicted and there is no need to do complicated initial synchronization again. 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 and sampling clock offset. The CP correlation can estimate the rough symbol time. In normal synchronization, pilot correlation can still help to find a new accurate symbol time.

As shown in Fig. 2.12, we track the symbol timing and frequency offset in stages I and II respectively. And we use pilot correlation to search for a more accurate symbol time and frame start time with a smaller search range. The simulation in [1] sets the search range in initial synchronization to ±32 samples around the estimated

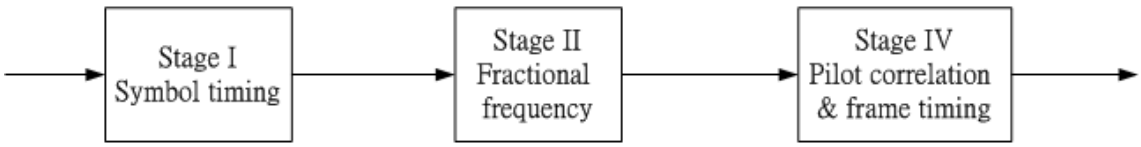

Fig. 2.12: Normal synchronization operations.

symbol time from CP correlation. For normal synchronization, the range is reduced to within ±5 samples. In this thesis, we set the normal synchronization's pilot search range to ±16 samples to get more reliable symbol timing estimates.

Concerning carrier frequency synchronization, according to 802.16a, the SS shall track the frequency changes and shall defer any transmission if synchronization is lost. Small frequency changes can be tracked by the fractional frequency part (stage II) of initial or normal 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 Sparse DFT

In some multiple access communications systems, transmitter and receivers may have different cost and capacity requirements. For instance, in a downlink scenario, one transmitter sends the same composite signal to multiple receivers. Each receiver may only be interested in a small fraction of the transmitted data. The transmitter may have high cost, provided the receivers have low cost.

Partial transforms offer the possibility of cost reductions in OFDM systems. In this section, we will introduce two kinds of methods. One is called the pruning

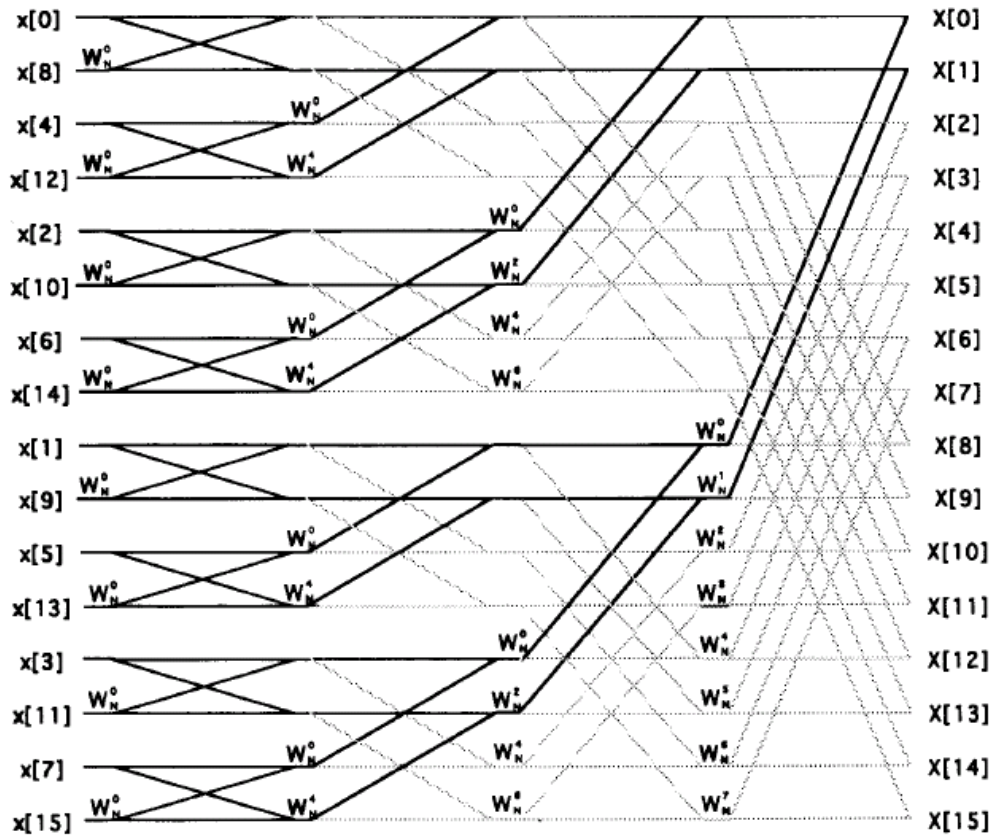

Fig. 2.13: Length 16 pruned FFT for a subset of output points (from [11]).

algorithm and the other is called the transform decomposition [11] algorithm. The following introduction is mainly taken from [11].

2.3.1 Pruning Algorithm

The pruning method is first devised by Markel [12]. Pruning is a modification of the standard one-butterfly radix-2 FFT. Fig. 2.13 shows how this pruning scheme works. Assuming that $X(0)$ and $X(1)$ are of interest, only the solid edges in the flow graph need to be computed, while the grey edges can be "pruned" away. By also shift twiddle factor in the program it is possible to get a band that does not start at $X(0)$, but can start anywhere. Multiplying all the twiddle factors by W_N^J, the L output values will be $X(J), X(J+1), \ldots, X(J+L-1)$, instead of $X(0), X(1), \ldots, X(L-1)$.

To compute L out of N DFT points, the regular pruning program requires

$$\#MUL_{PRUNE} = 2N\lfloor \log_2 L \rfloor + 2N - 4L + \frac{2NL}{2^{\lfloor \log_2 L \rfloor}}$$

real multiplications and

$$\#ADD_{PRUNE} = 3N\lfloor \log_2 L \rfloor + 3N - 6L + \frac{3NL}{\lfloor \log_2 L \rfloor}$$

real additions. More discussion about pruning algorithm can be found in [11].

The pruning algorithm can only compute consecutive output points. It cannot compute the output points with random indices. For this reason, the pruning algorithm is not suitable for 802.16a implementation.

2.3.2 Transform Decomposition [11]

A method, transform decomposition, for computing only a subset of output points will now be introduced. It is shown to be more efficient and more flexible than the pruning algorithm. We know that the DFT is designed as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \tag{2.3.1}$$

where $k = 0, 1, \ldots, N-1$. Assume that only L output points are needed and that there exists a P such that P divides N and define $Q = N/P$. Using the variable substitution

$$n = Qn_1 + n_2 \tag{2.3.2}$$

where $n_1 = 0, 1, \ldots, P-1$, and $n_2 = 0, 1, \ldots, Q-1$. We can rewrite the DFT as follows:

$$X(k) = \sum_{n_2=0}^{Q-1} \sum_{n_1=0}^{P-1} x(n_1 Q + n_2) W_N^{(n_1 Q + n_2)k} \tag{2.3.3}$$

$$= \sum_{n_2=0}^{Q-1} [\sum_{n_1=0}^{P-1} x(n_1 Q + n_2) W^{n_1 <k>_P}] W_N^{n_2 k} \tag{2.3.4}$$

where $<>_P$ denotes reduction modulo P, and k takes on any L consecutive values between 0 and $N-1$. Breaking this up into two equations

$$X(k) = \sum_{n_2=0}^{Q-1} X_{n_2}(< k >_P) W_N^{n_2 k} \tag{2.3.5}$$

where

$$X_{n_2}(j) = \sum_{n_1=0}^{P-1} x(n_1 Q + n_2) W_P^{n_1 j} \tag{2.3.6}$$

$$= \sum_{n_1=0}^{P-1} x_{n_2}(n_1) W_P^{n_1 j} \tag{2.3.7}$$

where $j = 0, 1, \ldots, P-1$. and $x_{n2} = x(n_1 Q + n_2)$. The sum in (2.3.7) can be recognized as a length P DFT, and it can be computed efficiently using any FFT algorithm. This is a great advantage of the transform decomposition method.

Inspecting (2.3.7), it can be seen that the sequence over which the DFT has to be computed is two dimensional and hence depends on n_2. Thus a DFT has to be computed for each different value of n_2, and hence there are Q such length P DFTs. The output of the DFTs are recombined using (2.3.5) which can be computed directly using Q complex multiplications and $Q-1$ complex additions per output point or a total of QL complex multiplications, each requiring 4 real multiplications and 2 real additions, and $L(Q-1)$ complex additions, each requiring 2 real additions. The advantage of the transform decomposition is that we can compute any output point with index k in (2.3.5), which can prove that the transform decomposition algorithm is more flexible than pruning algorithm. Fig.2.14 shows how this method works to compute the first L out of N DFT points.

2.3.3 Transform Decomposition with Filtering Approach [11]

It is possible to lower the number of operations required to compute (2.3.5) even further using a technique similar to Goertzel algorithm [13]. To see this, rewrite

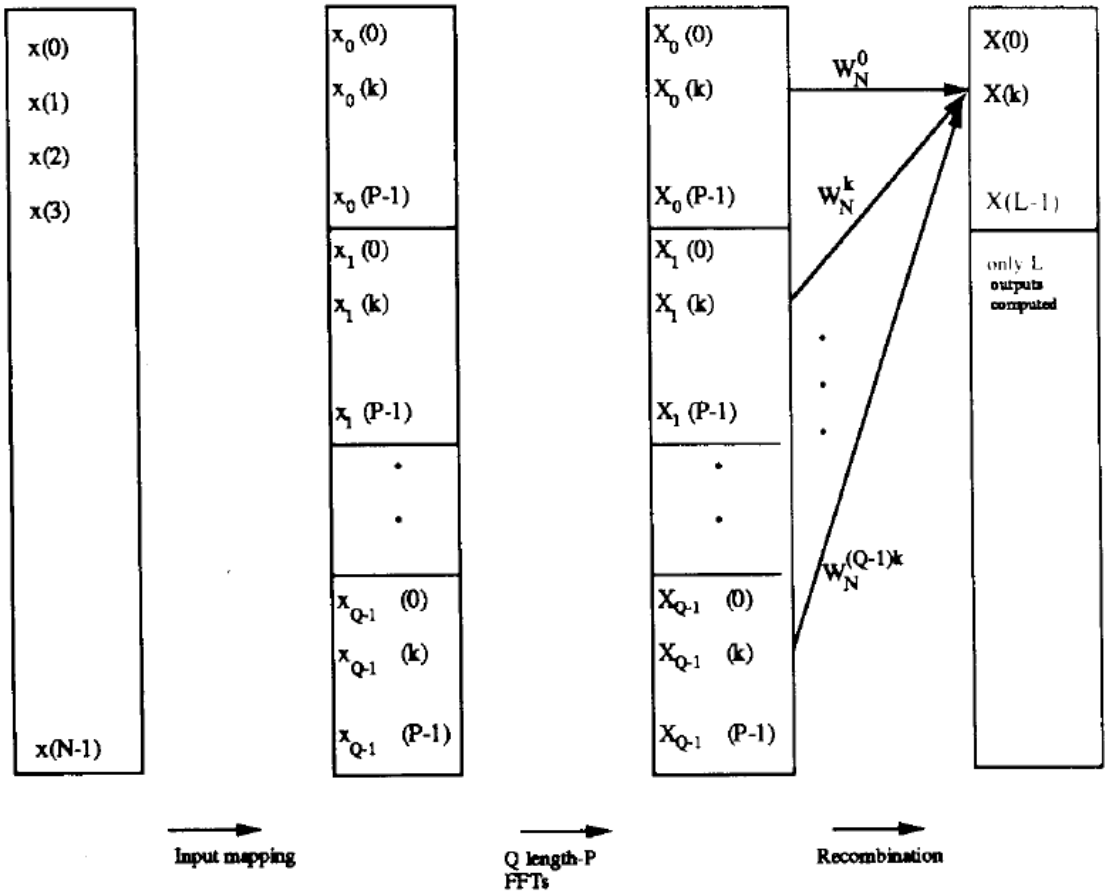

Fig. 2.14: Block diagram of the transform decomposition method of DFT for a subset of outputs (from [11]).

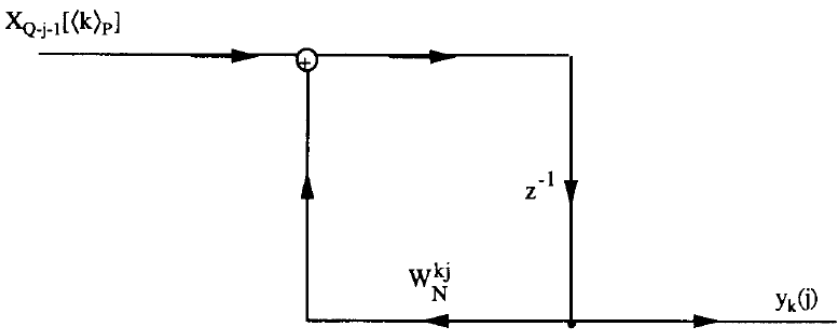

Fig. 2.15: Flow graph of first order network to compute (2.3.10) (from [11]).

(2.3.5) as follows:

$$X(k) = \sum_{n_2=0}^{Q-1} X_{n_2}(< k >_P)(W_N^k)^{n_2} \tag{2.3.8}$$

$$= \sum_{m=0}^{Q-1} X_{Q-m-1}(< k >_P)(W_N^k)^{Q-m-1} \tag{2.3.9}$$

with the variable substitution $m = Q - n_2 - 1$. Now define

$$y_k(j) = \sum_{m=0}^{j-1} X_{Q-m-1}(< k >_P)(W_N^k)^{j-m-1} \tag{2.3.10}$$

from which we can find $X(k)$ as

$$X(k) = y_k(j)|_{j=Q}. \tag{2.3.11}$$

Equation (2.3.10) can be recognized as a shifted cyclic convolution between the sequence $X_{Q-j-1}(< k >_P)$ and $(W_N^k)^{j-1}$ in the variable j and hence $y_k(j)$ can be viewed as the output of a system with impulse response $(W_N^k)^{j-1}$ driven by the input $X_{Q-j-1}(< k >_P)$.

Fig. 2.15 shows a flow graph that implements (2.3.10), but a quick analysis will show that this implementation requires 4 real multiplications per iteration assuming the input is complex, and hence requires the same amount of operations as a direct implementation of (2.3.5).

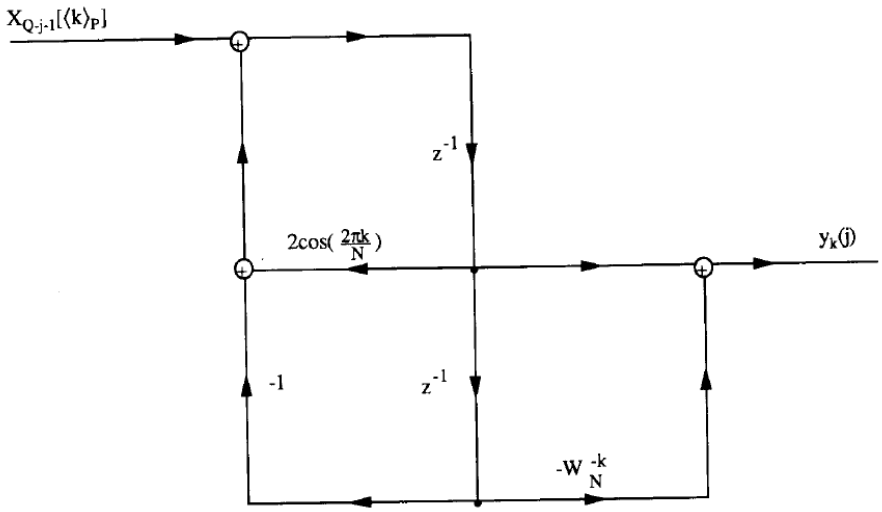

Fig. 2.16: Flow graph of second order network to compute (2.3.14) (from [11]).

The transfer function of the system in Fig. 2.15 is

$$H_k(z) = \frac{z^{-1}}{1 - z^{-1}W_N^k} \tag{2.3.12}$$

which can be rewritten as

$$\begin{aligned} H_k(z) &= \frac{z^{-1}(1 - z^{-1}W_N^{-k})}{(1 - z^{-1}W_N^k)(1 - z^{-1}W_N^{-k})} && (2.3.13) \\ &= \frac{z^{-1}(1 - z^{-1}W_N^{-k})}{1 - 2\cos(\frac{2\pi k}{N})z^{-1} + z^{-2}}. && (2.3.14) \end{aligned}$$

This last equation can be implemented using the flow graph in Fig. 2.16. Assume that the input is complex. Then each iteration only takes two real multiplications since the multiplication by -1 need not be counted. This is half of what was needed in the first order case. Because we are only interested in $y_k(Q)$, but not the intermediate values, it can be seen that the zero of the system is only needed once.

The derivations of (2.3.10) and (2.3.11) are not based on the actual values of the indices of the computed output values, i.e., does not rely on the specified values of k. Unlike the standard FFTs, efficient computation of (2.3.10) and (2.3.11) by the flow graph in Fig. 2.16 does not depend on combining computations for several

different output points (several different k). Hence the number of output points to be computed can be any length L subset of the N possible output points. This is a very powerful result that shows that transform decomposition is not just more efficient than pruning, but also more flexible. Where pruning restricts you to L subsequent output values, transform decomposition allows any length L subset to be computed.

2.3.4 Complexity Analysis

For the transform decomposition method, the computational complexity is discussed in [11]. Given that N is a power of two, then we need

$$\#MUL_{TD} = N\log_2 P - 3N + 4(L+1)\frac{N}{P} - 4L \tag{2.3.15}$$

real multiplications and

$$\#ADD_{TD} = 3N\log_2 P - 3N + 4(L+1)\frac{N}{P} - 4L \tag{2.3.16}$$

real additions.

The computational complexity for transform decomposition with filtering is

$$\#MUL_{TD-FILT} = N\log_2 P - 3N + 2(L+2)\frac{N}{P} + 2L \tag{2.3.17}$$

real multiplications and

$$\#ADD_{TD-FILT} = 3N\log_2 P - 3N + 4(L+1)\frac{N}{P} - 4L + 4P \tag{2.3.18}$$

real additions.

It still needs to be determined what values to use for the factor P. For most applications the number of output points L is given and the optimum P has to be found. To minimize the total number of operations, P should be chosen as

$$P_{TOT-MIN-TD} = [2(L+1)\log_e 2]_{close} \tag{2.3.19}$$

for the transform decomposition method, where $[\ ]_{close}$ indicates closest power of two. Unfortunately, the problem is nonlinear, and hence it is not "closest" in any easily determined sense, so both the larger and smaller possible values of P should be examined. If instead the lowest possible of multiplications is required, P should be chosen as

$$P_{MUL-MIN-TD} = [4(L+1)\log_e 2]_{close}. \tag{2.3.20}$$

The lower number of multiplications may be more useful for us because the multiplication operations are fewer than addition operations.

For transform decomposition with filtering method, the P can be chosen as

$$P_{TOT-MIN-TD-FILT} = [\frac{\sqrt{(\frac{N}{\log_e 2})^2 + 6LN + 8N} - (\frac{N}{\log_e 2})}{2}]_{close} \tag{2.3.21}$$

to minimize the total number of operations, and

$$P_{MUL-MIN-TD-FILT} = [2(L+2)\log_e 2]_{close} \tag{2.3.22}$$

to minimize the number of multiplications. Hence if the total number of operations is to be minimized, P should be chosen slightly larger than L, while if the number of multiplications is to be minimized, P should be chosen about three times the size of L (from the simulation results in [11]). This result will become the major reason that we do not adopt the transform decomposition algorithm for our implementation.

There is more discussion about these methods in [11].

2.3.5 Discussion

Because the TMS320C6416 DSP chip can perform 6 additions but only 2 multiplications at the same time, we consider the multiplication complexity in this section.

In downlink transmission, the carriers we need to use are 166 pilot carriers plus user data carriers. So the output points we need to compute are $L = 166 + 48 \times k$, where k is the number of subchannels assigned to the users (SSs).

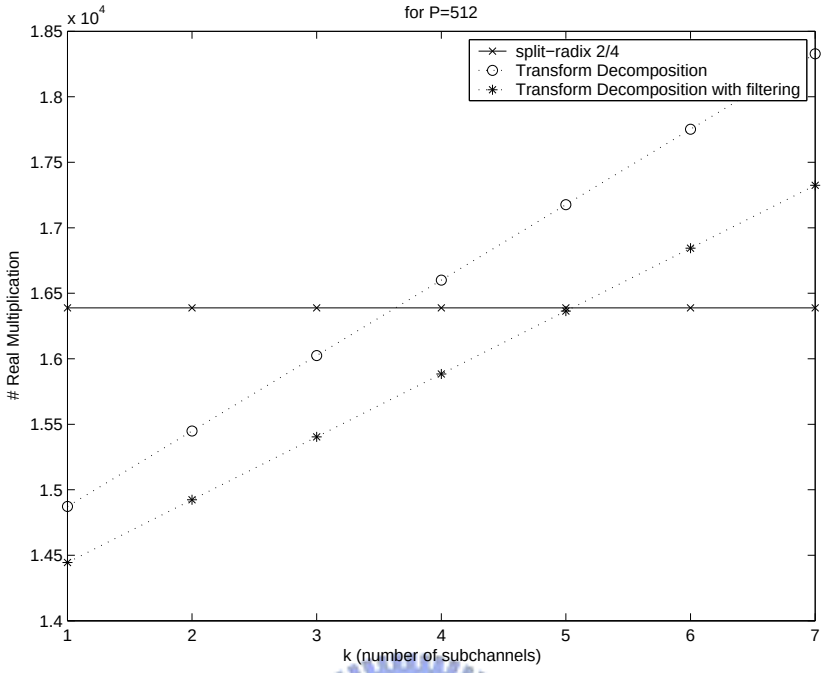

Fig. 2.17: Number of multiplications needed for transform decomposition when $P = 512$.

From the simulation results in [11], the value of P should be chosen about three times the size of L to minimize the number of multiplications, so the only proper values of P are 512 and 1024. For these vales of P, the numbers of subchannel which can be assigned to the SSs are bounded by $\lfloor (512-166)/48 \rfloor = 7$ and $\lfloor (1024-166)/48 \rfloor = 17$ respectively.

Figs. 2.17 and 2.18 show the number of multiplications needed at $P = 512$ and 1024 respectively. In these figures, we also show the multiplication complexity of split-radix 2/4 algorithm which is one of the most efficient algorithms for complete-points FFT.

For $P = 512$, we can find that if the number of subchannels used is larger than 4 or 5, we would be better off using the split-radix algorithm to compute all the points. For $P = 1024$, it is more efficient using the transform decomposition algorithm when $k \leq 7$. Further, the filtering approach performance is even worse

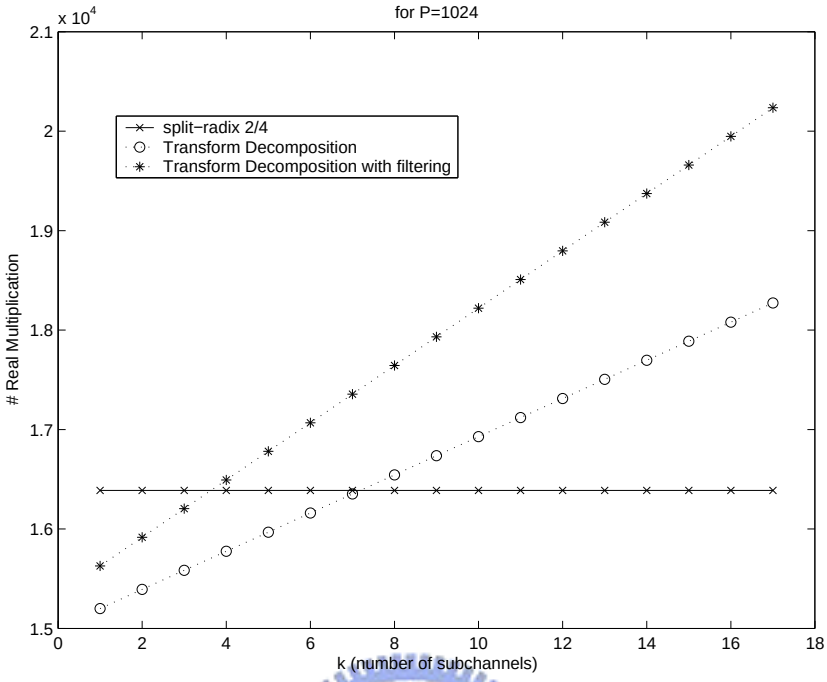

Fig. 2.18: Number of multiplications needed for transform decomposition when $P = 1024$.

than transform decomposition. According to our observation, it results from that the filter taps are left to 2 when $P = 1024$, so we cannot obtain enough advantage from the computation of the poles of (2.3.14) while we have to pay the computation of the zero.

Based on the above, we decide not to adopt the transform decomposition algorithm in our implementation of 802.16a DL transmission. In the 802.16a specification [3], we may assign all the subchannels to one SS. Besides, Texas Instruments provides high performance FFT functions in their DSPLIB [22]. The analysis of TI's FFT functions is given in chapter 4.

As a final remark, we note that we have only discussed the "many to few" case of transform decomposition algorithm above, which means that the number of FFT output points L is smaller than the number of FFT input points N. The case of "few to many" can be applied to the uplink transmission of 802.16a. We refer to

[11] for details of the methods.

Chapter 3

Introduction to the DSP Implementation Platform

We introduced the 802.16a DL transmission system in the last chapter. In this work, we conduct a DSP (digital signal processor) implementation of a DL transmitter-receiver pair. This chapter introduces the Quixote DSP-FPGA baseboard made by Innovative Integration (II) and the on-board DSP which is Texas Instruments' TMS320C6416. Our discussion will concentrate on the DSP chip and the associated system development environment because our implementation is purely software on the DSP.

3.1 The Quixote Baseboard [15]

The DSP-FPGA embedded card used in our implementation is Innovative Integration's Quixote baseboard, which is illustrated in Fig. 3.1. Quixote is one of Innovative Integration's Velocia-family baseboards for various applications requiring high-speed computation. Fig. 3.2 shows a block diagram of the Quixote board. It combines a 600 MHz C6416 32-bit fixed-point DSP with a Virtex-II FPGA, and system-level peripherals.The FPGAs on our boards are six-million-gate version. The TI C6416 DSP operating at 600 MHz offers a processing power of 4800 MIPS. Some detailed features of the board are as follows:

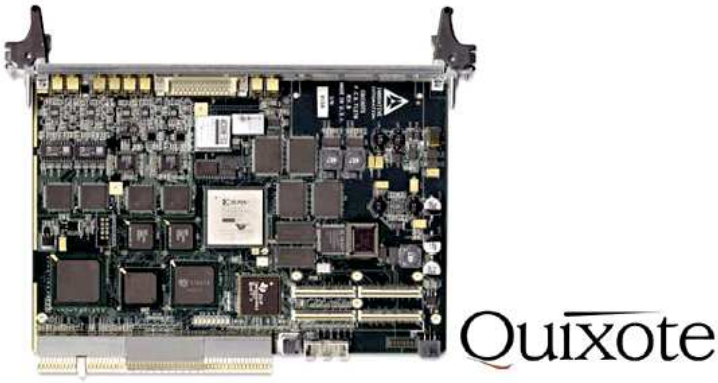

Fig. 3.1: Picture of the Quixote card [15].

- TMS 320C6416 processor running at frequency up to 600 MHz.
- Onboard 32 MB SDRAM for DSP chip, enhanced cache controllers, 64 DMA channels, 3 McBSP synchronized serial ports and two 32 bits timers.
- A 32/64 bits PCI bus host interface with direct host memory access capability for busmastering data between the card and the memory.
- 2 input, 2 output A/D and D/A conversion, 14 bit, DC to 105 MHz.

3.2 Quixote's Transfer Mechanisms [15]

Many applications in DSP baseboard may involve communication with the host CPU in some manner. They may have to interact with a host program during the lifetime of the program. Some examples are:

- Passing parameters to the program at start time.
- Receiving progress information and results from the application.
- Passing updated parameters during the run time of the program, such as the frequency and amplitude of a wave to be produced on the target.

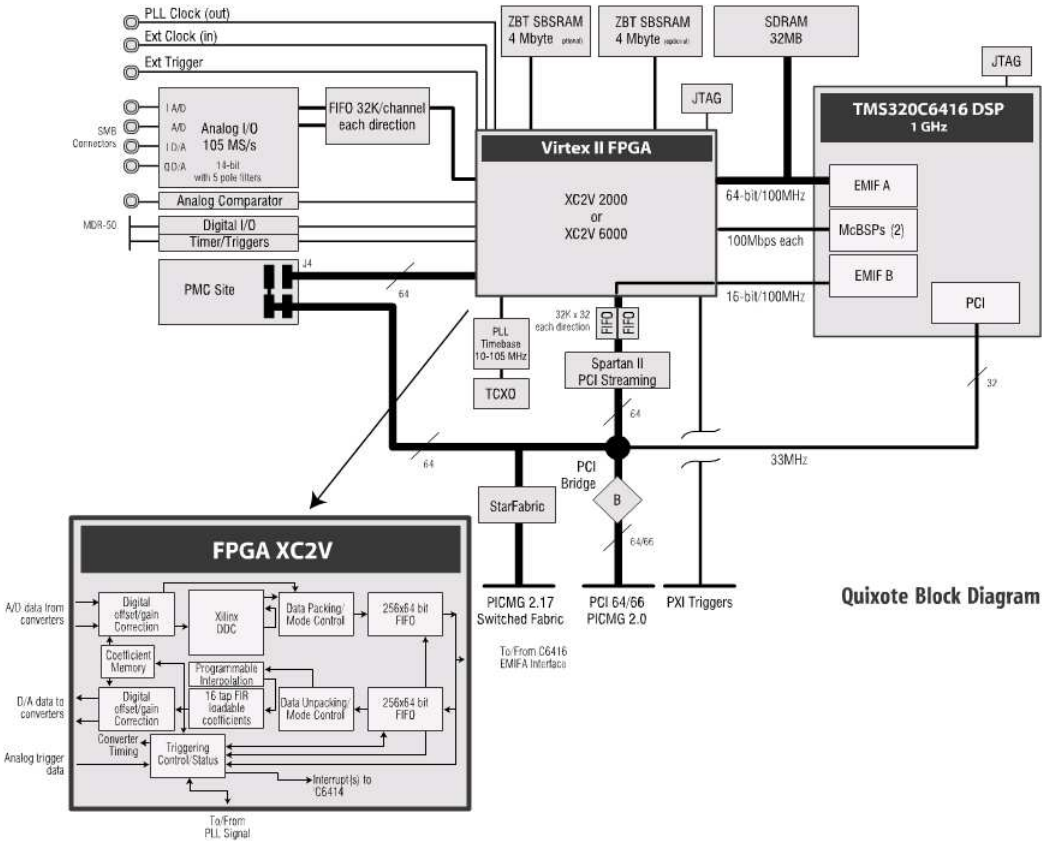

Fig. 3.2: Block diagram of Quixote (from [23]).

- Receiving alert information from the target.
- Receiving snapshots of data from the target.
- Sending a sample waveform to be generated to the target.
- Receiving full rate data.
- Sending data to be streamed at full rate.

There are three transfer methods on Quixote, which are DSP streaming interface, CPU busmastering interface, and packetized message interface. The following text is mainly taken from [15].

3.2.1 DSP Streaming Interface

The DSP streaming interface is continuous block based streaming transfer. It is designed for non-stop operation such as A/D and D/A.

The DSP streaming interface is bi-directional. Two stream can run simultaneously, one running from the analog peripherals through the DSP into the application. This is called the "incoming stream." The other stream runs out of the analog peripherals. This is the "outgoing stream." The mechanism is shown in Fig. 3.3. In both cases, the DSP needs to act as a mediator, since there is no direct access to analog peripherals from the host. This arrangement allows the DSP to process the streams as they move from the application to the hardware.

3.2.2 CPU Busmastering Interface

This method of target-to-host communication is on the Velocia baseboards only. The TI 64x baseboard is capable of using PCI busmastering to move data between target and host memories. This additional busmaster channel can be used to transfer data between host and target applications.

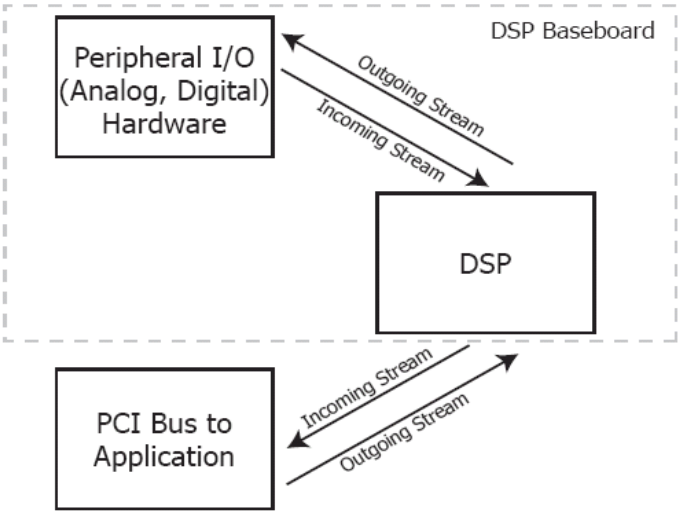

Fig. 3.3: DSP streaming mode (from [15]).

The CPU busmastering interface is packet based transfers which transfer discrete blocks between source and destination. Each data buffer is transferred completely to the destination in a single operation. The data buffers transferred can be of different sizes. Each requested buffer is interrogated for its size and fully transmitted. At the destination, the destination buffer is re-sized to allow the incoming data to fit. Reallocating buffers can take some time, for best performance buffers should be pre-sized to be large enough for the largest transfer expected.

CPU busmastering uses a simple blocking interface for its sending and receiving functions. The sending function will not return until the transfer has completed and the buffer is ready for reuse. Similarly, the receiving function waits until data have arrived from the data source and transferred into the data buffer before returning.

Since the transfer functions are blocking, they are best avoided in the main user interface thread of a Windows application. The GUI will appear to be frozen until the transfer has completed. For best results, the data transfer function should be

placed in separate threads in target and host applications. In fact, each direction of transfer should have its own thread, so that the two directions of transfer can interleave as much as possible.

The CPU busmastering interface allows separate channels of data between the target and the host. Using separate channels allows multiple, independent data streams to be maintained between the target and host. At present, only a single channel is supported. The largest transfer allowed is half of the total size of the DMA buffer allocated by the INF file (a kind of files used for software/firmware installation in windows system) when the driver is installed. Half of the memory is dedicated to each direction. The default buffer size in the INF is 0x200000 bytes, so the maximum transfer block is 1 MB.

3.2.3 Packetized Message Interface

In addition to the busmastering streaming interface, the DSP and host have a lower bandwidth (limited to about 56 kB/sec) communications link for sending commands or out-of-band information between target and host. Software is provided to build a packet-based message system between the target and host software. These packets can provide a simple yet powerful means of sending commands and information across the link between the two processes.

As shown in Fig. 3.4, the message system's arrangement provides one bi-directional link between the target and the host. The "CIIMessage" and "IImessage" are host and target side message objects declarations respectively. The detailed contents of the packet formatting are shown in Table 3.1. The "CIIbaseboard::OnMessage" and "Unsolicited Message Handler" are the messages handler used to handle the message when messages are received for host and target sides respectively. The "Post" function is just used for sending the message out.

In this study, we use the methods of CPU busmastering and message interface for

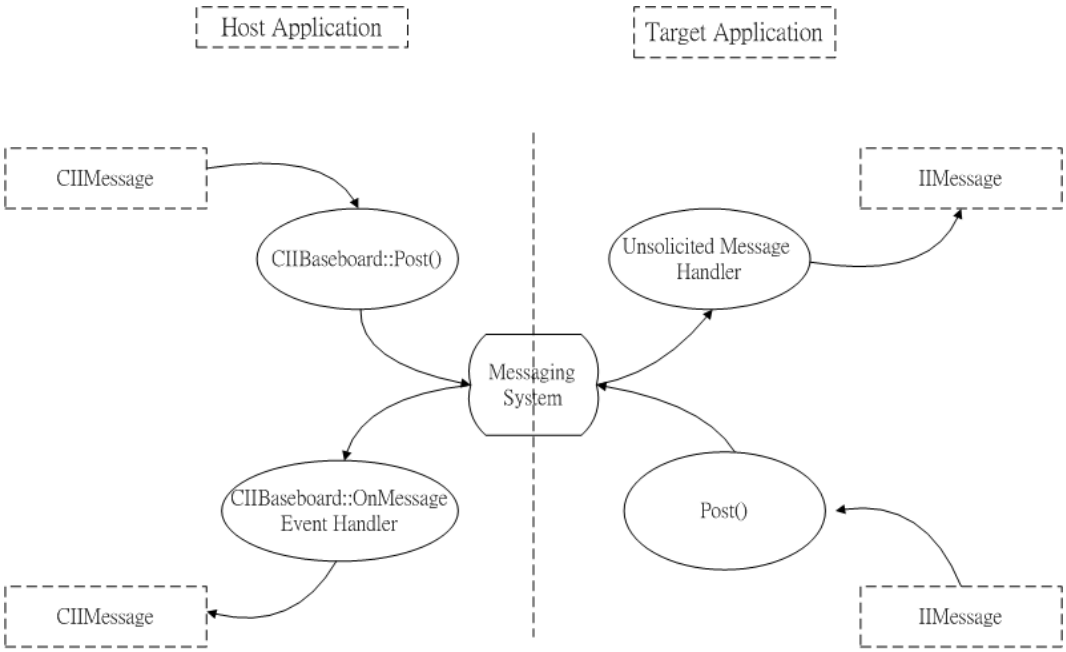

Fig. 3.4: The message system (from [15]).

Table 3.1: Message Packet Formatting (from [15])

Function Name	Property
Channel	Message Channel
TypeCode	Message or Command type
MessageId	Message counter or other user data
IsReplyExpected	Set if reply is needed. Free for use in application
Data[ ]	Access the data region as 32-bit integers (index 0–13)
AsFloat[ ]	Access the data region as floating point data (index 0–13)
Asshort[ ]	Access the data region as 16-bit integers (index 0–27)
AsChar[ ]	Access the data region as 8-bit characters (index 0–55)

communication between the host and the target. The CPU busmastering interface provides higher bandwidth for data transmission. But the disadvantage is that only one channel is supported. Packetized message interface supports sixteen channels in each direction. But the bandwidth is limited to 56 kB/sec.

3.3 The TMS320C6416 DSP Chip [23]

The following text is mainly taken from references [2] and [23].

3.3.1 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 TI. The C6416 device has two high-performance embedded coprocessors, Viterbi Decoder Coprocessor (VCP) and Turbo Decoder Coprocessor (TCP) that can significantly speed up channel-decoding operations on-chip, but we do not make use of these coprocessors now.

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 devices 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.

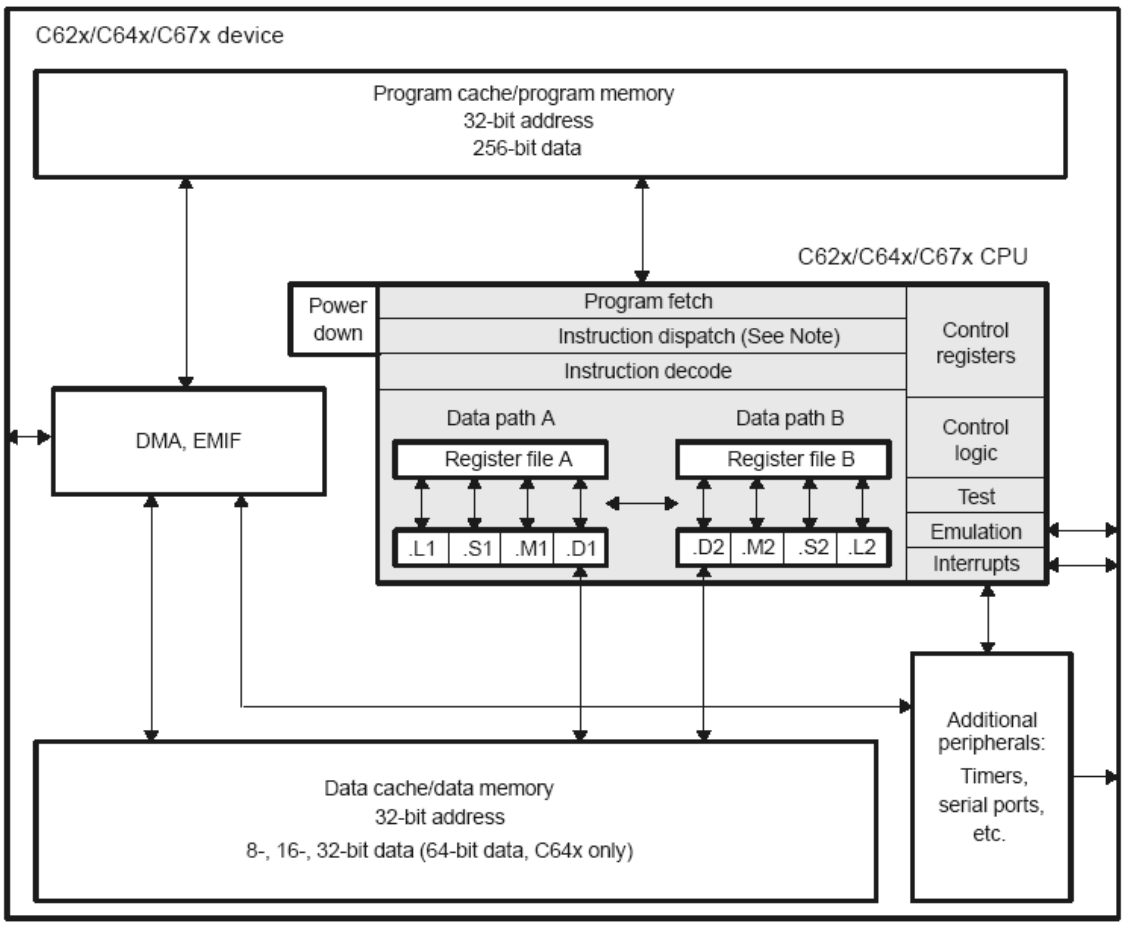

Fig. 3.5: Block diagram of TMS320C6416 DSP (from [20]).

- 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.

3.3.2 Central Processing Unit Features [20]

The block diagram of C6416 DSP is shown in Fig. 3.5. The DSP contains: program fetch unit, instruction dispatch unit, instruction decode unit, two data paths which each has four functional units, 64/32-bit registers, control registers, control logic, and logic for test, emulation, and logic.

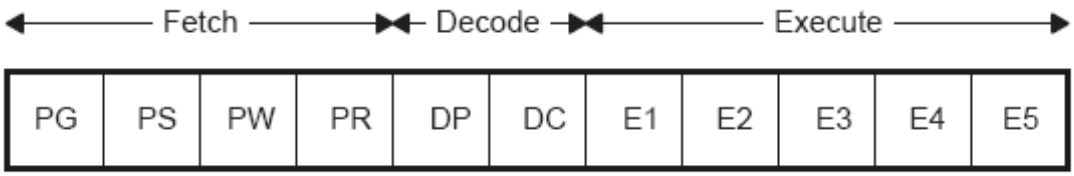

Fig. 3.6: Pipeline phases of TMS320C6416 DSP (from [20]).

The TMS320C64x DSP pipeline provides flexibility to simplify programming and improve performance. The pipeline can dispatch eight parallel instructions every cycle. The following two factors provide this flexibility: Control of the pipeline is simplified by eliminating pipeline interlocks, and the other is increasing pipelining to eliminate 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, and 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. 3.6.

Reference [20] contains detailed information regarding the fetch and decode phases. The pipeline operation of the C62x/C64x instructions can be categorized into seven instruction types. Six of these are shown in Table 3.2, which gives a mapping of operations occurring in each execution phase for the different instruction types. The delay slots associated with each instruction type are listed in the bottom row.

The execution of instructions can be defined in terms of delay slots. A delay slot is a CPU cycle that occurs after the first execution phase (E1) of an instruction. Results from instructions with delay slots are not available until the end of the last

Table 3.2: Execution Stage Length Description for Each Instruction Type (from [20])

		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‡

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 3.3.

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

Table 3.3: Functional Units and Operations Performed (from [20])

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 and 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

ADD4 instruction performs four 8-bit additions with a single instruction on a .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 (see Fig. 3.7). All units ending in 1 (for example, .L1) write to register file A, and all units ending in 2 write to register file B. Each functional unit has two 32-bit read ports for source operands src1 and src2. Four units (.L1, .L2, .S1, and .S2) have an extra 8-bit-wide port for 40-bit long writes, as well as an 8-bit input for 40-bit long reads. Because each unit has its own 32-bit write port, when performing 32-bit operations all eight units can be used in parallel every cycle.

3.3.3 Cache Memory Architecture Overview [19]

The C64x memory architecture consists of a two-level internal cache-based memory architecture plus external memory. Level 1 cache is split into program (L1P) and data (L1D) cache. The C64x memory architecture is shown in Fig. 3.8. On C64x devices, each L1 cache is 16 kB. All caches and data paths are automatically managed by cache controller. Level 1 cache is accessed by the CPU without stalls. Level 2 cache is configurable and can be split into L2 SRAM (addressable on-chip memory) and L2 cache for caching external memory locations. On a C6416 DSP, the size of L2 cache is 1 MB, and the external memory on Quixote baseboard is 32 MB. More detailed introduction to the cache system can be found in [19].

3.4 TI's Code Development Environment [16], [26]

TI provides a useful GUI development interface to DSP users for developing and debugging their projects: Code Composer Studio (CCS). The CCS development

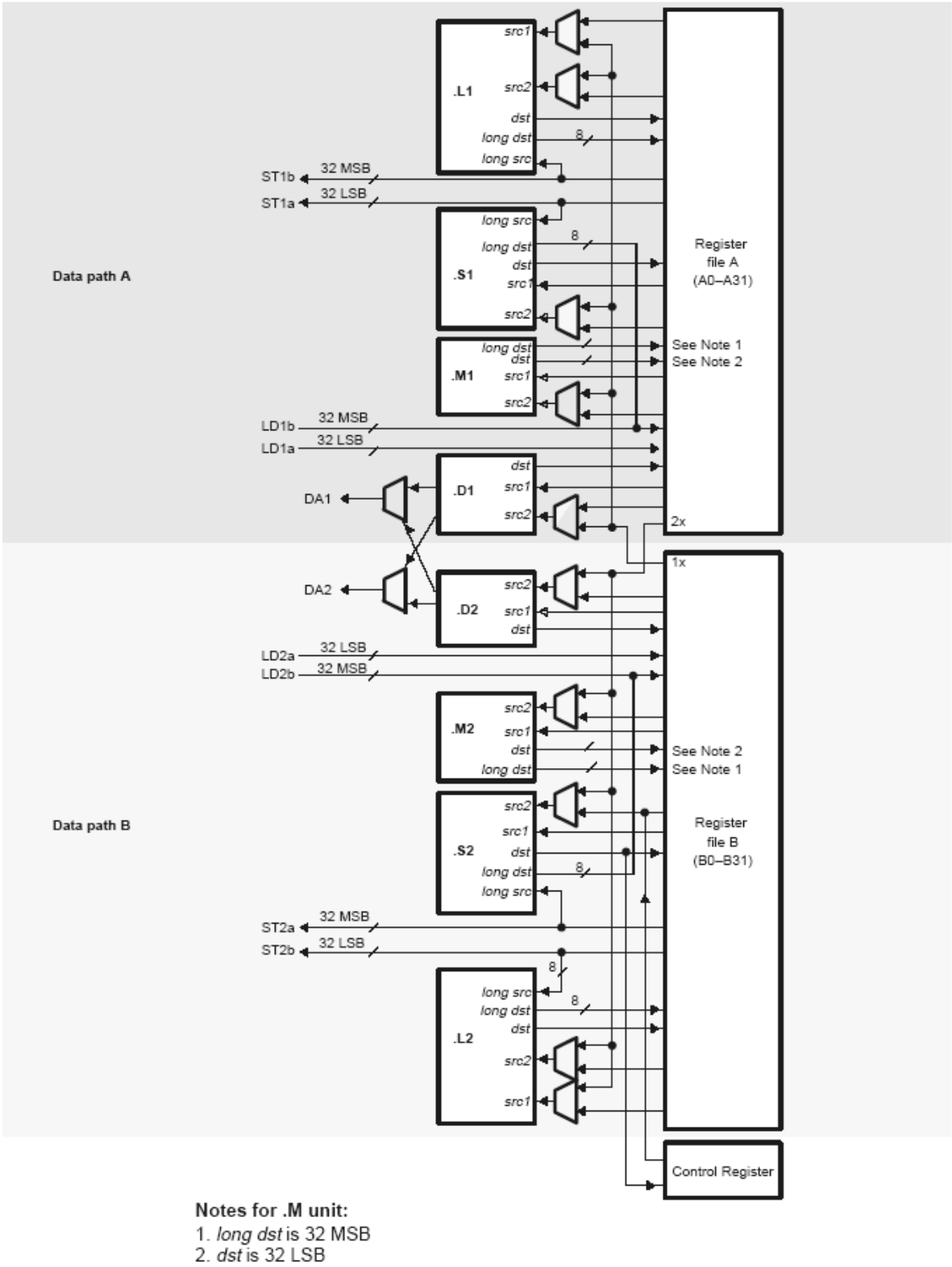

Fig. 3.7: TMS320C64x CPU data path (from [20]).

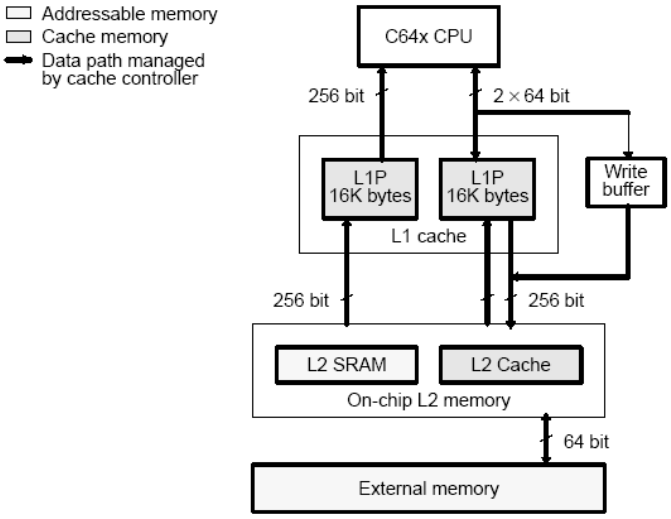

Fig. 3.8: C64x cache memory architecture (from [19]).

tools are a key element of the DSP software and development tools from Texas Instruments. The fully integrated development environment includes real-time analysis capabilities, easy to use debugger, C/C++ compiler, assembler, linker, editor, visual project manager, simulators, XDS560 and XDS510 emulation drivers and DSP/BIOS support.

Some of CCS's fully integrated host tools include:

- Simulators for full devices, CPU only and CPU plus memory for optimal performance.
- Integrated visual project manager with source control interface, multi-project support and the ability to handle thousands of project files.
- Source code debugger common interface for both simulator and emulator targets:
  - C/C++/assembly language support.
  - Simple breakpoints.

  - Advanced watch window.
  - Symbol browser.

- DSP/BIOS host tooling support (configure, real-time analysis and debug).
- Data transfer for real time data exchange between host and target.
- Profiler to 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 some optimized DSP functions for the TMS320C64x devices: the TMS320C64x digital signal processor library (DSPLIB). The routines included in the DSP library are organized into seven groups:

- Adaptive filtering.

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

In this study, we use the FFT and IFFT functions from this library.

3.5 Code Development Flow [21]

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

The recommended code development flow for the C6000 involves the phases described in Fig. 3.9. The tutorial section of the Programmers Guide [21] focuses on phases 1–2 and the Guide also instructs 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 the goal for each phase in the 3-phase software development flow shown in Fig. 3.9.

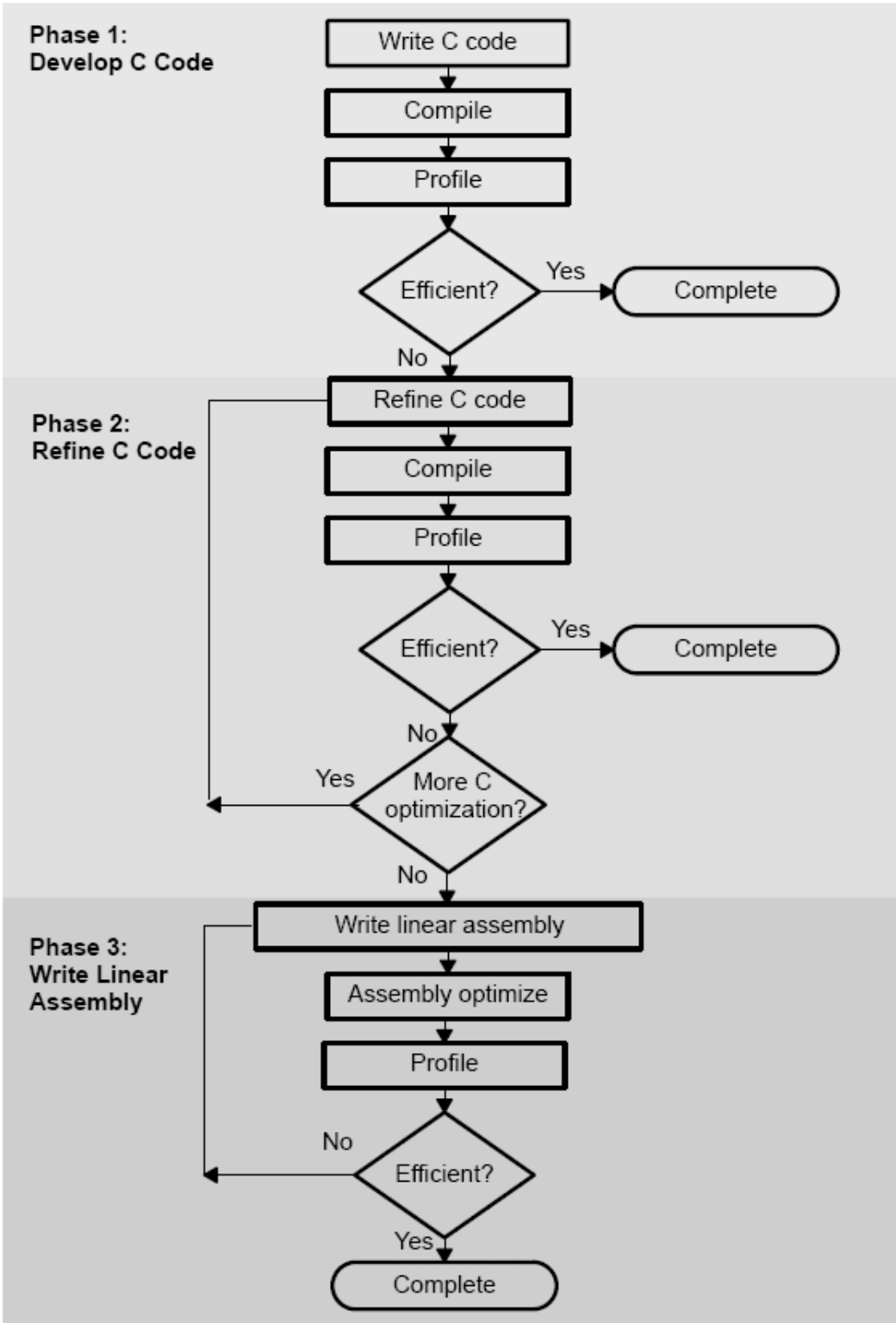

Fig. 3.9: Code development flow for TI C6000 DSP (from [21]).

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

- Use techniques described in [21] to improve the C code. Use the C6000 profiling tools to check its performance. If the code is still not as efficient as we would like it to be, proceed to phase 3.

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

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

3.5.1 Compilier Optimization Options [21]

The compiler supports several options to optimize the code. The compiler options can be used to optimize code size or execution performance. Our primary concern in this work is the execution performance. Hence we do not care very much about the code size (at least in this work). 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 through the optimization and code generation passes of the compiler. Because the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization:

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

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

Chapter 4

DSP Implementation

In this chapter, we discuss how we implement the DL transmission system on the Quixote baseboard based on the synchronization programs developed in [2] and the channel estimation programs developed in [6].

First, we introduce how we organize the system on the DSP and determine the fixed-point data formats employed. Then we discuss the system performance.

4.1 System Structure

The 802.16a DL system that we implement includes the transmitter and the receiver on the DSP and a channel simulator on the host PC, as shown in Fig. 4.1. The transmitter does data modulation, framing, IFFT, up-samping and SRRC filtering. The channel simulator can simulate multipath fading, AWGN, and frequency offset. The receiver contains synchronizer, channel estimator, de-modulation and de-framing. The reason why we put the channel simulator on the host PC is because it is computationally very expensive.

First, the transmitter generates one symbol worth of transmitted signal and transfers it to the host as one block. After the host PC has received 16 blocks (i.e., 16 symbols per frame), it applies the channeling effect. After the simulated multipath fading and AWGN effect, we send the signal back to the DSP into the receiver and

Table 4.1: System Memory Arrangement

	Total Size	Used for Cache	Used for Memory
L1 Cache	32 KBytes	32 KBytes	None
L2 Cache	1 MBytes	256 KBytes	400 KBytes
External Memory	32 MBytes	None	16.14 MBytes

perform synchronization, channel estimation and other receiver function.

The program controller is the host PC program. We develop our system based on the examples "CpuInRate" and "CpuOutRate" provided by Inovative Integration. The simple examples use the CPU busmastering interface and the message system for communication between the host and the DSP. As described in the last chapter, we use CPU busmastering interface for data exchanges and packetized message interface for debugging and controlling message exchanges.

4.1.1 Memory Arrangement

As introduced in section 3.3.3, the DSP chip and the baseboard contain a two-level cache and one external memory. Table 4.1 describes the usages of the cache and the memory. Level 1 cache consists of program and data cache and it is used for cache purpose only. Level 2 cache is split into cache and on-chip memory areas. There are 256 kB of the level 2 cache reserved for the cache system and 400 kB are used by our DL system. The external memory is used for memory only and a total of 16.14 MB are used in our system.

4.1.2 Fixed-Point Data Formats

In this section, we introduce the fixed-point data formats used in the implemented system. As shown in Fig. 4.2, the transmitted source data are generated randomly and fed into the modulator. The output format of the modulator is Q1.14 because

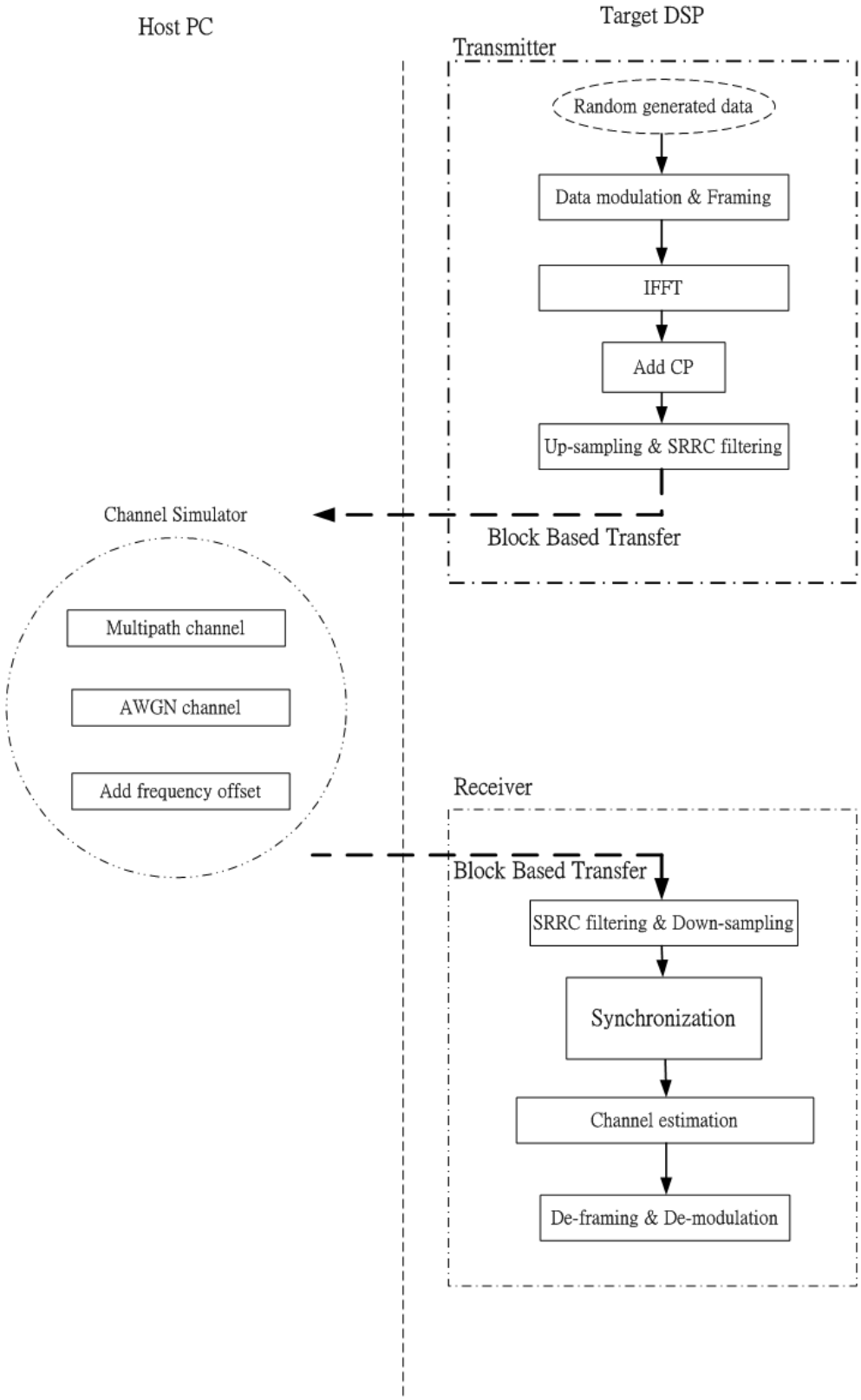

Fig. 4.1: System integration structure.

Table 4.2: Performance Comparision of Frequency Lock Between Floating-Point and Fixed-Point Implementation (from [2])

Doppler shift f_dT_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

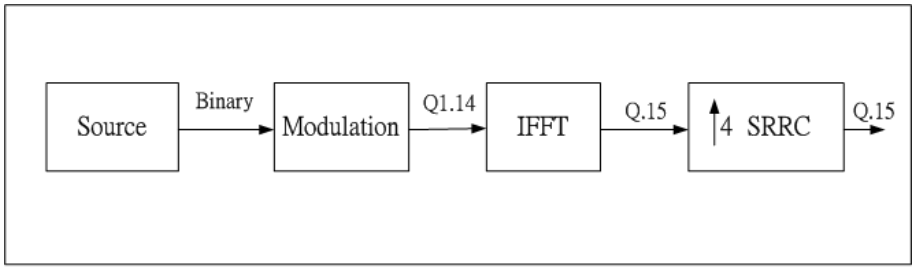

Fig. 4.2: Fixed-point data formats used in the transmitter.

the pilots may have values of $\pm\frac{4}{3}$. And the format after the IFFT is Q.15. Fig. 4.3 shows the formats used in the receiver. Almost everywhere from SRRC output to the FFT input uses the format Q.15 except where dealing with the frequency offset. The format after FFT is Q5.10 because the multipath fading channel may cause gains to the modulated data. In the channel estimator, we could find out the channel response and then compensate for it, so the format is changed back to Q1.14 for de-framing and de-modulation. From [2], we can get the performance differences of the synchronization between floating-point and fixed-point data type, as shown in Tables 4.2 and 4.3. We can find that the Q.15 format fixed-point computation is precise enough for the synchronization process.

Table 4.3: Performance Comparision of Frame Lock Between Floating-Point and Fixed-Point Implementation (from [2])

Doppler shift f_dT_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

4.2 System Performance

In our simulation, we allocate 5 bursts (users) in the downlink part of one 802.16a frame. Source data are generated randomly, and are modulated into 64-QAM. There are 12 OFDMA symbols in one DL subframe and 4 OFDMA symbols in each UL subframe. The TTG and RTG are 136 samples. The frame structure and the bursts allocation are shown in Fig. 4.4. The frame is repeated several times in transmission. The above are arbitrary choices of parameter for purposes of system design. The programs are quite general and can use other sets of parameters.

We employ the multipath ETSI "Vehicular A" channel model [1]. Information about this channel model is given in Table 4.4. And the maximum Doppler shifts of our simulation are shown in Table 4.5 for several speeds between 0 and 120 km/hr.

4.2.1 Execution Cycles of the Original Programs

In our system, one symbol duration is 201.6 μs and there are 2304 samples in a symbol. The clock frequency of the DSP is 600 MHz. The execution clock cycles are 120960 in a symbol duration and average to 52.5 in a sample duration. For real-time operation, therefore, everything must complete in 120960 cycles per

Table 4.4: Characteristics of the ETSI "Vehicular A" Channel Environment [14]

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 4.5: Relations Between Speed 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

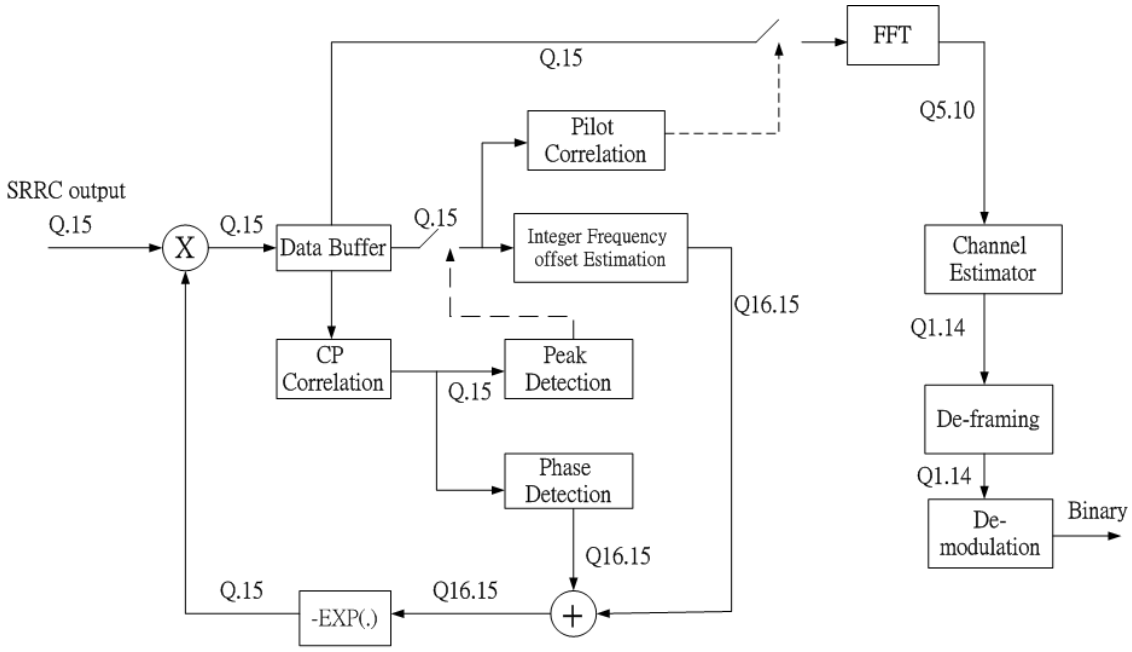

Fig. 4.3: Fixed-point data formats used in the receiver (based on [2]).

symbol or 52.5 cycles per sample unless multiple DSPs are used. In the following analysis in this chapter, we define a metric called "multiples of real-time" which means that how many DSP processors we need to finish the function in time. Multiples of Real-Time $= \frac{\text{Practical Avg. Execution Cycles per Sample}}{\text{Computation Capacity of Real\_Time per Sample}}$.

The original program cycles information based on [2] is shown in Tables 4.6 and 4.7. Each time when the modulation/de-modulation functions are performed, they generate 1536 data samples, so we can divide the average cycles per symbol by 1536 to get average cycles per sample. With the same reason, we divide 1702 for framing/de-framing functions, 2048 for FFT/IFFT functions and 2304 for the others functions to get the average cycles per sample. And we use the average cycles per sample to calculate the multiples of real-time. The statistics illustrated in the tables are from [2] with some modifications which drop out uses of "fread" and "fwrite" functions.

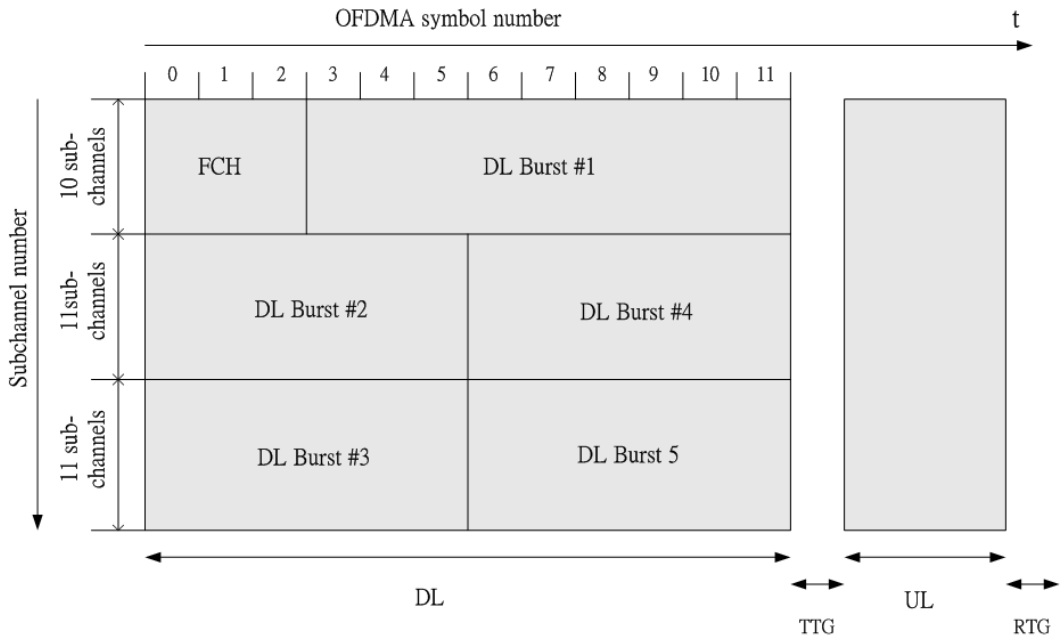

Fig. 4.4: Allocation of bursts in a frame.

We can find that some functions are particularly time-consuming. In next section, we will introduce some techniques to accelerate the programs.

Table 4.6: Profile of the Original 802.16a DL Transmitter Function Blocks (based on [2])

	Code Size (Bytes)	Avg. Cycles per Symbol / #Samples per Symbol = Avg. Cycles per Sample	Multiples of Real-Time
Modulation	544	188973/1536 = 123.02	2.34
Framing	2464	187916/1702 = 110.40	2.10
IFFT	964	35728/2048 = 17.44	0.332
Tx_SRRC_filter	1624	6199452/2304 = 2690.73	51.28

Table 4.7: Profile of the Original 802.16a DL Receiver Function Blocks (based on [2]).

	Code Size (Bytes)	Avg. Cycles per Symbol / #Samples per Symbol = Avg. Cycles per Sample	Multiples of Real-Time
SRRC_downsample	348	520704/2304 = 226	4.30
CP_correlation	1320	232704/2304 = 101	1.92
initial_freq_sync	300	66816/2304 = 29	0.55
integer_freq_sync	932	96768/2304 = 42	0.8
pilot_corre	2824	539136/2304 = 234	4.456
sync	784	1290240/2304 = 560	10.66
FFT	276	32256/2048 = 15.75	0.26
de_framing	1064	833350/1702 = 489.62	9.32
de_modulation	3544	125326/1536 = 81.59	1.55

4.2.2 Efficiency Enhancement

4.2.2.1 Modulation Functions

In this section, we will describe the techniques used to improve the performance of the modulation function. Fig. 4.5 shows a part of the original modulation program and we see that some "if" and "else" statements are used to check the modulation type inside the outer "for" loop. This is inefficient because we do not change the modulation type within one data block. In addition, the compiler cannot do software pipelining for this kind of coding style. Because the modulation can only have three types (QPSK, 16QAM, and 64QAM), we separate their handling into three sub-functions, as shown in Figs. 4.6 and 4.7. Table 4.8 compares the execution cycles before and after modification. The compiler optimization information is shown in Fig. 4.8 and Fig. 4.9 is a main section of the assembly code of the modulation function together with the corresponding C code.

```
1 for(j=0;j<(coded_block_size/3);j++)
2 {
3         datain=(unsigned int)(input[3*j]);
4         datain=(datain<<8)^(unsigned int)(input[3*j+1]);
5         datain=(datain<<8)^(unsigned int)(input[3*j+2]);
6         datain=datain<<8;
7         if(coding_modul==0 || coding_modul==1)
8         {
9                 for(i=0;i<24;i++)
10                {
11                temp=(unsigned char)( (datain & (0xC0000000)) >> 31 );
12                switch(temp)
13                {
14                        case 0:
15                                IQ=1;
16                                break;
17                        case 1:
18                                IQ=-1;
19                                break;
20                }
21                out[DataIndex]=c4*IQ;
22                DataIndex++;
23                }
24        }
25        else if(coding_modul==2 || coding_modul==3)
26        {
27                for(i=0;i<12;i++)
28                {
29                temp=(unsigned char)( (datain&(0xC0000000)) >> 30 );
30                switch(temp)
31                {
32                        case 0:
33                                IQ=1;
34                                break;
35                        case 1:
36                                IQ=3;
37                                break;
38                        case 2:
39                                IQ=-1;
40                                break;
41                        case 3:
42                                IQ=-3;
43                                break;
44                }
45                datain<<=2;
46                out[DataIndex]=c16*IQ;
47                DataIndex++;
48
49                }
50        }
```

Fig. 4.5: A part of the original modulation program.

```
1 //==========================================================================================
2 //                        modified program of the modulation function
3 //==========================================================================================
4 void modulation(unsigned char *input,int coding_modul, int  coded_block_size,FIXED *out)
5 {
6                 if(coding_modul==0 || coding_modul==1)
7                 {
8                         modulation4 (input, coding_modul, coded_block_size,out);
9                 }
10
11                else if(coding_modul==2 || coding_modul==3)
12                {
13                        modulation16 (input, coding_modul, coded_block_size,out);
14                }
15
16                else if(coding_modul==4 || coding_modul==5)
17                {
18                        modulation64 (input, coding_modul, coded_block_size,out);
19                }
20 }
21
22 void modulation4(unsigned char *input,int coding_modul, int  coded_block_size,FIXED *out)
23 {
24
25         FIXED QAM4[2]={11585,-11585};
26
27         coded_block_size=coded_block_size/3;
28
29
30         DataIndex=0;
31
32         for(j=0;j<(coded_block_size/3);j++)
33         {
34                 datain=(unsigned int)(input[3*j]);
35                 datain=(datain<<8)^(unsigned int)(input[3*j+1]);
36                 datain=(datain<<8)^(unsigned int)(input[3*j+2]);
37                 datain=datain<<8;
38
39
40                 for(i=0;i<24;i++)
41                 {
42                         temp=(unsigned char)( (datain & (0xC0000000)) >> 31 );
43                         out[DataIndex]=QAM4[temp];
44                         datain<<=1;
45                         DataIndex++;
46                 }
47         }
48 }
49
```

Fig. 4.6: A part of the modified program in the modulation function.

Table 4.8: Comparison of the Modulation Function Before and After Optimization

Original Code		Revised Code		Improvement
Cycles/Symbol	Cycles/Sample	Cycles/Symbol	Cycles/Sample	
188973	123.02	8310	5.41	95.60%

```
1 void modulation16(unsigned char *input,int coding_modul, int  coded_block_size,FIXED *out)
2 {
3         short   i,j;
4         unsigned int    datain;
5         unsigned char   temp;
6         int DataIndex;
7         FIXED QAM16[4]={5181,15543,-5181,-15543};
8
9         coded_block_size=coded_block_size*2/3;
10        DataIndex=0;
11        for(j=0;j<(coded_block_size/3);j++)
12        {       datain=(unsigned int)(input[3*j]);
13                datain=(datain<<8)^(unsigned int)(input[3*j+1]);
14                datain=(datain<<8)^(unsigned int)(input[3*j+2]);
15                datain=datain<<8;
16
17                for(i=0;i<12;i++)
18                {
19                        temp=(unsigned char)( (datain&(0xC0000000)) >> 30 );
20                        out[DataIndex]=QAM16[temp];
21                        datain<<=2;
22                        DataIndex++;
23                }
24        }
25 }
26 void modulation64(unsigned char *input,int coding_modul, int  coded_block_size,FIXED *out)
27 {
28        short   i,j;
29        unsigned int    datain;
30        unsigned char   temp;
31        int DataIndex;
32        FIXED QAM64[8]={7584,2528,12640,17696,-7584,-2528,-12640,-17696};
33
34        DataIndex=0;
35        for(j=0;j<(coded_block_size/3);j++)
36        {
37                datain=(unsigned int)(input[3*j]);
38                datain=(datain<<8)^(unsigned int)(input[3*j+1]);
39                datain=(datain<<8)^(unsigned int)(input[3*j+2]);
40                datain=datain<<8;
41
42                for(i=0;i<8;i++)
43                {
44                        temp=(unsigned char)( (datain&(0xE0000000)) >> 29 );
45                        out[DataIndex]=QAM64[temp];
46                        datain<<=3;
47                        DataIndex++;
48                }
49        }
50 }
```

Fig. 4.7: The other part of the modified program in the modulation function.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 57
;*      Loop opening brace source line   : 58
;*      Loop closing brace source line   : 63
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 12
;*      Known Maximum Trip Count         : 12
;*      Known Max Trip Count Factor      : 12
;*      Loop Carried Dependency Bound(^) : 2
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     3*       1
;*      .D units                     1        3*
;*      .M units                     0        0
;*      .X cross paths               2        2
;*      .T address paths             2        2
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          3        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        1
;*      Bound(.L .S .D .LS .LSD)     3*       3*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 5 iterations in parallel
;*
```

Fig. 4.8: Compiler feedback of the modulation4 function.

4.2.2.2 Framing and De-framing Functions

In Table 4.7, the execution cycles of framing/de-framing seem extraordinarily large. In this section, we analyze the reasons of the inefficiency of the original code and find ways of improvement through loop unrolling and software pipelining by the compiler.

First, we introduce the original code of de-framing function and propose a better coding style. As shown in Fig. 4.10, the problem of the original code consists in the waste of cycles in the large number of "or" operations in the "if" statement in every iteration, as shown in the circle denoted "part 1." The same problem exists in the framing function. The proposed C code uses simple skills to prevent this waste of cycles and does away with the modulo operation, as shown in the "part 1" code in Fig. 4.11.

Another modification of the de-framing function is done to "part 2" in Fig. 4.10 and results "part 2" in Fig. 4.11. We just remove the variable "carrier_n_s" by

```
7052 ;------------------------------------------------------------------------
7053 ;   57 | for(i=0;i<24;i++)                                  C code
7054 ;------------------------------------------------------------------------
7055
7056            MVK     .D1     0x3,A1          ; init prolog collapse predicate
7057 ||         SUB     .D2     B6,4,B4
7058 ||         AND     .L1     2,A5,A3         ; |60| (P) <0,1>
7059 ||         SHRU    .S2X    A3,29,B5        ; |60| (P) <0,1>  ^
7060 ||         SHL     .S1     A3,2,A4         ; |61| (P) <0,1>  ^
7061
7062 ;** --------------------------------------------------------------------------*
7063 L9:     ; PIPED LOOP KERNEL
7064          .line   28
7065 ;------------------------------------------------------------------------
7066 ;   59 | temp=(unsigned char)( (datain & (0xC0000000)) >> 31 );
7067 ;   60 | out[DataIndex]=QAM4[temp];                                        C code
7068 ;   61 | datain<<=1;
7069 ;   62 | DataIndex++;
7070 ;------------------------------------------------------------------------
7071
7072    [ A2]   SUB     .L1     A2,1,A2         ; <0,11>
7073 || [!A1]   STH     .D2T2   B7,*++B4(4)     ; |60| <0,11>
7074 || [ A0]   BDEC    .S1     L9,A0           ; |63| <1,8>
7075 || [!A2]   LDH     .D1T1   *A5,A6          ; |60| <2,5>
7076 ||         ADD     .S2     8,SP,B6         ; |60| <3,2>
7077 ||         AND     .L2     2,B5,B5         ; |60| <3,2>
7078
7079    [ A1]   SUB     .D1     A1,1,A1         ; <0,12>
7080 || [!A1]   STH     .D2T1   A6,*+B4(2)      ; |60| <0,12>
7081 ||         ADD     .L1X    8,SP,A3         ; |60| <3,3>
7082 ||         ADD     .S2X    B6,A3,B6        ; |60| <3,3>
7083 ||         SHRU    .S1     A4,30,A5        ; |60| <4,0>
7084
7085          .line   33
7086
7087            ADD     .D1X    A3,B5,A5        ; |60| <3,4>
7088 ||         LDH     .D2T2   *B6,B7          ; |60| <3,4>
7089 ||         AND     .L1     2,A5,A3         ; |60| <4,1>
7090 ||         SHL     .S1     A4,2,A4         ; |61| <4,1>  ^
7091 ||         SHRU    .S2X    A4,29,B5        ; |60| <4,1>  ^
7092
7093 ;** --------------------------------------------------------------------------*
7094 L10:    ; PIPED LOOP EPILOG
7095
7096            ADD     .S1     3,A7,A7         ; |64|
7097 ||         STH     .D2T2   B7,*++B4(4)     ; |60| (E) <2,11>
7098 ||         LDH     .D1T1   *A5,A6          ; |60| (E) <4,5>
7099 ||         SUB     .S2     B0,1,B0         ; |64|
7100
7101    [ B0]   BNOP    .S2     L7,1            ; |64|
7102 ||         STH     .D2T1   A6,*+B4(2)      ; |60| (E) <2,12>
```

Fig. 4.9: Kernel of the assembly code of the modulation4 function.

Table 4.9: Comparison of Framing/De-framing Functions Before and After Optimization

	Original Code		Revised Code		Improvement
	Cycles per Symbol	Cycles per Sample	Cycles per Symbol	Cycles per Sample	
framing	187916	110.40	25676	15.08	86.34%
de-framing	833350	489.62	7373	4.33	99.11%

replacing it with a look-up table, which is the framing/de-framing indexing number.

As illustrated in Table. 4.9, we can get huge improvement after the modifications. This is because the original C code cannot result in software pipelining and loop unrolling with the use of large numbers of "if," "else," and "or" operations. We can get detailed information about how the compiler is able to optimize the code from the CCS compiler feedback information shown in Fig. 4.12. We find that the software pipelining is 6 stages deep from the sentence "Schedule found with 6 iterations in parallel." Fig. 4.13 is the kernel of the assembly code of the de-framing function, where the corresponding C code is also illustrated. We can compare the kernels of the assembly code before and after revision. The assembly code for the original program is shown in Fig. 4.14 and we can see that it cannot be software pipelined, so the assembly programs are very inefficient.

4.2.2.3 FFT and IFFT Functions

The FFT/IFFT functions we use are from TI's DSPLIB [22]. The original programs [2] have used FFT/IFFT functions that employ 32-bit operations. Because the C6416 DSP chip could perform four 16-bit multiplication operations but only two 32-bit multiplication operations during one cycle, it is more efficient if we could use 16-bit multiplications. The Table 4.10 compares the performance of the FFT functions provided in the DSPLIB.

DSP_fft32x32 is the complex mixed radix 32×32-bit FFT with rounding, while

```
1 //=========================================================================================//
2 //                The original code of the de-framing function                              ,
3 //=========================================================================================//
4 void de_framing(FIXED *fft_in, int IDcell, int *symbol, FIXED *symbol_out){
5     short     s,n,k;
6     int     carrier;
7     int     carrier_n_s;
8         int             L;
9         int             i;
10        int             carrier_map[1536];
11
12        unsigned char   perbase[32]={3,18,2,8,16,10,11,15,26,22,6,9,27,20,25,1,29
13                                    ,7,21,5,28,31,23,17,4,24,0,13,12,19,14,30};
14
15    if(*symbol%4==0)        L=0;
16    else if(*symbol%4==1)    L=2;
17    else if(*symbol%4==2)    L=1;
18    else                    L=3;
19    (*symbol)++;
20
21    carrier=0;                                          //Find the data location indices
22    for(i=0;i<dl;i++) {
23        if((i==0)||(i==39)||(i==261)||(i==330)||(i==342)  // these are pilots location indices
24            ||(i==351)||(i==522)||(i==636)||(i==645)
25            ||(i==651)||(i==708)||(i==726)||(i==756)
26            ||(i==792)||(i==849)||(i==855)||(i==918)
27            ||(i==1017)||(i==1143)||(i==1155)||(i==1158)
28            ||(i==1185)||(i==1206)||(i==1260)||(i==1407)
29            ||(i==1419)||(i==1428)||(i==1461)||(i==1530)
30            ||(i==1545)||(i==1572)||(i==1701)||((i-3*L+12)%12==0))
31                {
32                        ;
33                }
34        else {                                                Part 1
35            carrier_map[carrier] = i;
36            carrier++;
37        }
38    }
39
40
41    i=0;
42    for(s=0,k=0;s<32;s++)                          //  framing
43    {   //s:index num of a subchannel
44        for(n=0;n<Nsubcarrier;n++,k++)
45        {                                                          //  calcualate the index
46            carrier_n_s=Nsubchannels*n+(perbase[(n+ps*s)%(Nsubchannels)]
47            +IDcell*(unsigned short)ceil(((float)(n+1)/(float)Nsubchannels)))%(Nsubchannels);
48
49            symbol_out[k*2]=fft_in[carrier_map[carrier_n_s]*2];
50            symbol_out[k*2+1]=fft_in[carrier_map[carrier_n_s]*2+1];
51            i++;
52        }                                                          Part 2
53    }
54 }
```

Fig. 4.10: Original C code of the de-framing function.

```
1 //==========================================================================================//
2 //                          The modified C code of de-framing function
3 //==========================================================================================//
4 void de_framing(FIXED *fft_in, int IDcell, int *symbol, FIXED *symbol_out){
5         short           s,n,k;
6         int             carrier;
7         int             L;
8         int             i;
9         int             carrier_map[1536];
10        char            data_loca[1702];
11        int fixed_pilot[32]={0,39,261,330,342,351,522,636,645,651,708,726,756,792,849
12                        ,855,918,1017,1143,1155,1158,1185,1206,1260,1407,1419,1428,1461,1530,1545
13                        ,1572,1701};
14
15    if(*symbol%4==0)        L=0;
16    else if(*symbol%4==1)    L=2;
17    else if(*symbol%4==2)    L=1;
18    else                    L=3;
19    (*symbol)++;
20
21        for(i=0;i<1702;i++)                     // Initialize
22                data_loca[i]=0;
23
24        for(i=0;i<32;i++)                       //  Find fixed-pilot location
25                data_loca[fixed_pilot[i]]=1;
26
27        for(i=3*L;i<1702;i=i+12)                //  Find variable-pilot location
28                data_loca[i]=1;
29
30        carrier=0;
31        for(i=0;i<1702;i++)                     // find the data carrier indices
32        {
33                if(data_loca[i]==0)
34                {
35                        carrier_map[carrier] = i;
36                        carrier++;
37                }
38        }
39
40
41    i=0;
42    for(s=0,k=0;s<32;s++)                        // framing
43    {   //s:index num of a subchannel
44        for(n=0;n<Nsubcarrier;n++,k++)
45        {   symbol_out[k*2]=fft_in[carrier_map[carrier_n_s_table[i]]*2];
46            symbol_out[k*2+1]=fft_in[carrier_map[carrier_n_s_table[i]]*2+1];
47            i++;
48        }
49    }
50 }
```

Part 1

Part 2

Fig. 4.11: Revised C code of the de-framing function.

```
27170 ;*----------------------------------------------------------------------------*
27171 ;*   SOFTWARE PIPELINE INFORMATION
27172 ;*
27173 ;*      Loop source line                 : 45
27174 ;*      Loop opening brace source line   : 46
27175 ;*      Loop closing brace source line   : 52
27176 ;*      Known Minimum Trip Count         : 1702
27177 ;*      Known Maximum Trip Count         : 1702
27178 ;*      Known Max Trip Count Factor      : 1702
27179 ;*      Loop Carried Dependency Bound(^) : 1
27180 ;*      Unpartitioned Resource Bound     : 1
27181 ;*      Partitioned Resource Bound(*)    : 1
27182 ;*      Resource Partition:
27183 ;*                                A-side   B-side
27184 ;*      .L units                     0        0
27185 ;*      .S units                     1*       0
27186 ;*      .D units                     1*       1*
27187 ;*      .M units                     0        0
27188 ;*      .X cross paths               0        0
27189 ;*      .T address paths             1*       1*
27190 ;*      Long read paths              0        0
27191 ;*      Long write paths             0        0
27192 ;*      Logical  ops (.LS)           0        0     (.L or .S unit)
27193 ;*      Addition ops (.LSD)          1        1     (.L or .S or .D unit)
27194 ;*      Bound(.L .S .LS)             1*       0
27195 ;*      Bound(.L .S .D .LS .LSD)     1*       1*
27196 ;*
27197 ;*      Searching for software pipeline schedule at ...
27198 ;*         ii = 1  Schedule found with 6 iterations in parallel
27199 ;*
```

Fig. 4.12: Software pipelining information of the revised code for the de-framing function.

Table 4.10: Comparison of Performance of FFT Functions in DSPLIB for N = 2048

	Code Size (Bytes)	Execution Cycles per Symbol	Minimum Cycles Needed per Symbol	Efficiency
DSP_fft32x32	932	28811	11351	39.39%
DSP_ifft32x32	932	28811	11351	39.39%
DSP_fft16x16r	868	15510	11351	73.18%

inverse FFT of the same type is DSP_ifft32x32. DSP_16x16r is the complex mixed radix 16×16-bit FFT with rounding. TI DSPLIB does not provide functions for 16-bit IFFT, so we have to do IFFT using the 16-bit FFT function. As shown in Fig. 4.15, we just need to do conjugation before and after FFT. More detailed usage of these functions can be found in [22].

Table 4.11 compares the computational complexity of different FFT algorithms. The mixed radix FFT needs 19974 real multiplications and 68102 real additions theoretically in our application which uses 2048-point FFT/IFFT. So the absolutely

```
27255 ;------------------------------------------------------------------------
27256 ;  45 | for(i=0;i<1702;i++)                                  C code
27257 ;------------------------------------------------------------------------
27258
27259            ADD      .D2      8,SP,B4            ; |44|
27260 ||         ZERO     .S2      B5                 ; |45|
27261 || [ A1]   BDEC     .S1      L142,A1            ; |52| (P) <4,0>
27262 ||         ADD      .L1      1,A3,A3            ; |52| (P) <4,0>  ^ Define a twin register
27263 ||         LDB      .D1T1    *+A3[A4],A0        ; (P) <4,0>  ^
27264
27265 ;** ---------------------------------------------------------------------------*
27266 L142:    ; PIPED LOOP KERNEL
27267         .line   35
27268 ;----------------------  ---------  ------------------------------
27269 ;  47 | if(data_loca[i]==0)
27270 ;  49 |          carrier_map[carrier] = i;                                C code
27271 ;  50 | carrier++;
27272 ;------------------------------------------------------------------------
27273         .line   41
27274 ;------------------------------------------------------------------------
27275 ;  73 | i=0;
27276 ;------------------------------------------------------------------------
27277
27278            ADD      .S2      1,B5,B5            ; |52| <0,5>  ^
27279 || [!A0]   STW      .D2T2    B5,*B4++           ; |49| <0,5>  ^
27280 || [ A1]   BDEC     .S1      L142,A1            ; |52| <5,0>
27281 ||         ADD      .L1      1,A3,A3            ; |52| <5,0>  ^ Define a twin register
27282 ||         LDB      .D1T1    *+A3[A4],A0        ; <5,0>  ^
27283
27284 ;** ---------------------------------------------------------------------------*
27285 L143:    ; PIPED LOOP EPILOG
27286
27287            MV       .D1X     B10,A5
27288 ||         ZERO     .S2      B9                 ; |74|
27289 ||         MVKL     .S1      _carrier_n_s_table,A4
27290 ||         ADD      .L2      1,B5,B5            ; |52| (E) <1,5>  ^
27291 || [!A0]   STW      .D2T2    B5,*B4++           ; |49| (E) <1,5>  ^
27292
27293            MVKH     .S1      _carrier_n_s_table,A4
27294 ||         ADD      .S2      1,B5,B5            ; |52| (E) <2,5>  ^
27295 || [!A0]   STW      .D2T2    B5,*B4++           ; |49| (E) <2,5>  ^
```

Fig. 4.13: Kernel of the assembly code of the revised de-framing function.

```
21697 ;------------------------------------------------------------------------
21698 ;  28 | if((i==0)||(i==39)||(i==261)||(i==330)||(i==342)
21699 ;  29 |      ||(i==351)||(i==522)||(i==636)||(i==645)
21700 ;  30 |      ||(i==651)||(i==708)||(i==726)||(i==756)
21701 ;  31 |      ||(i==792)||(i==849)||(i==855)||(i==918)
21702 ;  32 |      ||(i==1017)||(i==1143)||(i==1155)||(i==1158)
21703 ;  33 |      ||(i==1185)||(i==1206)||(i==1260)||(i==1407)
21704 ;  34 |      ||(i==1419)||(i==1428)||(i==1461)||(i==1530)
21705 ;  35 |      ||(i==1545)||(i==1572)||(i==1701)||((i-3*L+12)%12==0))
21706 ;  37 |                  ;
21707 ;------------------------------------------------------------------------
21708             ZERO    .D1     A9                  ; |27|
21709 ;*----------------------------------------------------------------------------*
21710 ;*   SOFTWARE PIPELINE INFORMATION
21711 ;*      Disqualified loop: Loop contains control code
21712 ;*----------------------------------------------------------------------------*
21713 L88:
21714         .line   28
21715 ;------------------------------------------------------------------------
21716 ;  39 | else {
21717 ;------------------------------------------------------------------------
21718             CMPEQ   .L2X    A9,B16,B4           ; |38|
21719             CMPEQ   .L1     A9,0,A5             ; |38|
21720
21721             CMPEQ   .L1     A9,A30,A5           ; |38|
21722 ||          OR      .D1X    B4,A5,A6            ; |38|
21723
21724             OR      .D1     A5,A6,A5            ; |38|
21725 ||          CMPEQ   .L2X    A9,B20,B5           ; |38|
21726
21727             CMPEQ   .L1     A9,A16,A6           ; |38|
21728 ||          CMPEQ   .L2X    A9,B18,B4           ; |38|
21729
21730             OR      .D2X    B4,A5,B4            ; |38|
21731             OR      .D2X    A6,B4,B4            ; |38|
21732             OR      .D2     B5,B4,B4            ; |38|
21733             CMPEQ   .L1     A9,A17,A5           ; |38|
21734
21735             CMPEQ   .L1     A9,A18,A5           ; |38|
21736 ||          OR      .D1X    A5,B4,A6            ; |38|
21737
21738             CMPEQ   .L1     A9,A19,A6           ; |38|
21739 ||          OR      .D1     A5,A6,A5            ; |38|
21740
21741             OR      .D1     A6,A5,A5            ; |38|
21742 ||          CMPEQ   .L2X    A9,B7,B5            ; |38|
21743
21744             CMPEQ   .L1     A9,A20,A6           ; |38|
21745 ||          CMPEQ   .L2X    A9,B6,B4            ; |38|
21746
21747             OR      .D2X    B4,A5,B4            ; |38|
21748             OR      .D2X    A6,B4,B4            ; |38|
21749             OR      .D2     B5,B4,B4            ; |38|
```

C code

cannot perform software pipelining

Fig. 4.14: Kernel of the assembly code of the original de-framing function.

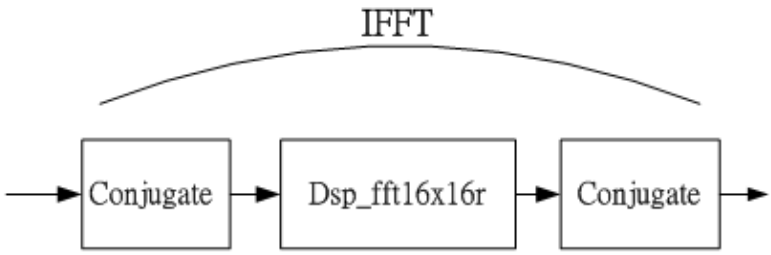

Fig. 4.15: IFFT implementation using FFT function.

minimum number of execution cycles is $\max\{19974/2, 68102/6\} = 11351$ for the 32-bit FFT/IFFT operation and $\max\{19974/4, 68102/6\} = 11351$ for the 16-bit FFT. Practically, as shown in Table 4.10, DSP_fft32x32 and DSP_ifft32x32 need 28811 clock cycles and DSP_16x16r needs 15510 clock cycles, so the efficiencies are 39.39% and 73.18%, respectively, where the efficiency is defined as

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

which indicates how well the compiler schedules the assembly code.

Fig. 4.16 shows the core loop in DSP_fft16x16r. The assembly code shown in the figure uses "_dotp2" and "_dotpn2" instructions to compute intermediate results. For example, the following code:

$$\begin{aligned}
\text{x2[l1]} &= (\text{si10} * \text{yt1_0} + \text{co10} * \text{xt1_0} + \text{0x8000}) \gg 16 \\
\text{x2[l1+1]} &= (\text{co10} * \text{yt1_0} - \text{si10} * \text{xt1_0} + \text{0x8000}) \gg 16 \\
\text{x2[l1+2]} &= (\text{si11} * \text{yt1_1} + \text{co11} * \text{xt1_1} + \text{0x8000}) \gg 16 \\
\text{x2[l1+3]} &= (\text{co11} * \text{yt1_1} - \text{si11} * \text{xt1_1} + \text{0x8000}) \gg 16
\end{aligned}$$

is mapped to the assembly code below:

```
DOTP2 .M2   B_xt0_0_yt0_0,   B_co20_si20,  B_x_l1_0 ;
DOTPN2 .M2  B_yt0_0_xt0_0,   B_co20_si20,  B_x_l1_1 ;
```

Table 4.11: Comparison of Computational Complexity of 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 4.12: Comparison of FFT/IFFT Before and After Optimization

	Original Code		Revised Code		Improvement
	Cycles per Symbol	Cycles per Sample	Cycles per Symbol	Cycles per Sample	
FFT	32256	15.75	17046	8.32	47.17%
IFFT	35728	17.44	24360	11.89	31.82%

```
DOTP2  .M2   B_xt0_1_yt0_1,   B_co21_si21,   B_x_l1_2 ;
DOTPN2 .M2   B_yt0_1_xt0_1,   B_co21_si21,   B_x_l1_3 ;
```

as indicated by the ovals in Fig. 4.16.

By this modification, the execution cycles of the IFFT and FFT functions in Tables 4.6 and 4.7 become 24360/2048 = 11.89 (cycles/sample) and 17046/2048 = 8.32 (cycles/sample) respectively, as shown in Table 4.12. The DSP_fft16x16 function is used inside the FFT/IFFT function. The excess clock cycles of FFT/IFFT over the DSP_fft16x16r cycle counts are from the data movement inside our FFT/IFFT functions.

4.2.2.4 SRRC Filter

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 functions, which TI provides, provide an another method for optimizing the program at C level. Detailed introduction to the intrinsic functions can be found in

```
4066          BDEC    .S1     LOOP_Y,           A_i                                ;[28,1]
4067 ||        ADD     .L1     A_rnd,            A_x_h2_0,         A_x_h2_0         ;[18,2]
4068 ||        DOTP2   .M2     B_xt0_0_yt0_0,    B_co20_si20,      B_x_l1_0         ;[18,2]
4069 ||[!A_ifj]ZERO    .L2     B_j                                                  ;[ 8,3]
4070 ||[!A_ifj]ADD     .S2     B_x,              B_fft_jmp,        B_x              ;[ 8,3]
4071 ||        MVD     .M1X    B_x,              A_x_                               ;[ 8,3]
4072 ||        ADD2    .D2X    B_xl2_3_xl2_2,    A_xh2_3_xh2_2,    B_xh21_1_xh20_1;[ 8,3]
4073 ||[!A_p0]STDW     .D1T1   A_xh2_3_2:A_xh2_1_0,                *A_x__[A_l1]     ;[28,1]
4074
4075 LOOP_Y5:
4076   [!A_p0]STDW     .D1T2   B_xl2_3_2:B_xl2_1_0,                *A_x__[A_l2]     ;[29,1]
4077 ||        PACKH2  .L2     B_x_l1_3,         B_x_l1_2,         B_xl1_3_2        ;[19,2]
4078 ||        ADD     .S2     B_rnd,            B_x_l2_0,         B_x_l2_0         ;[19,2]
4079 ||        DOTPN2  .M2     B_yt0_0_xt0_0,    B_co20_si20,      B_x_l1_1         ;[19,2]
4080 ||        DOTP2   .M1     A_xt1_1_yt1_1,    A_co11_si11,      A_x_h2_2         ;[19,2]
4081 ||        PACKLH2 .S1     A_xl21_0_xl20_0,A_xl21_0_xl20_0,  A_xl20_0_xl21_0;[ 9,3]
4082 ||        SUB2    .L1X    A_x_1_x_0,        B_xl1_1_xl1_0,    A_xl1_0_xl0_0    ;[ 9,3]
4083 ||        ADD2    .D2X    B_xl1_3_xl1_2,    A_x_3_x_2,        B_xh1_1_xh0_1    ;[ 9,3]
4084
4085 LOOP_Y6:
4086          ADD     .D2     B_rnd,            B_x_l2_1,         B_x_l2_1         ;[20,2]
4087 ||        DOTPN2  .M2X    B_co31_si31,      A_yt2_1_xt2_1,    B_x_l2_3         ;[20,2]
4088 ||        DOTPN2  .M1     A_yt1_1_xt1_1,    A_co11_si11,      A_x_h2_3         ;[20,2]
4089 ||        STDW    .D1T2   B_x_3_x_2:B_x_1_x_0,                *A_x_[0]         ;[20,2]
4090 ||        SUB2    .S1     A_xl1_0_xl0_0,    A_xl20_0_xl21_0,  A_yt1_0_xt2_0    ;[10,3]
4091 ||        ADD2    .L1     A_xl1_0_xl0_0,    A_xl20_0_xl21_0,  A_yt2_0_xt1_0    ;[10,3]
4092 ||        SUB2    .S2     B_xh1_1_xh0_1,    B_xh21_1_xh20_1,  B_yt0_1_xt0_1    ;[10,3]
4093 ||        ADD     .L2     B_x,              8,                B_x              ;[10,3]
4094
4095 ;LOOP_Y7:
4096          ADD     .L2     B_rnd,            B_x_l2_2,         B_x_l2_2         ;[21,2]
4097 ||        MVD     .M1     A_x_,             A_x__                              ;[21,2]
4098 ||        DOTPN2  .M2     B_yt0_1_xt0_1,    B_co21_si21,      B_x_l1_3         ;[11,3]
4099 ||        PACKLH2 .S2     B_yt0_1_xt0_1,    B_yt0_1_xt0_1,    B_xt0_1_yt0_1    ;[11,3]
4100 ||        PACKLH2 .L1     A_yt1_0_xt2_0,    A_yt2_0_xt1_0,    A_xt2_0_yt2_0    ;[11,3]
4101 ||        PACKLH2 .S1     A_yt2_0_xt1_0,    A_yt1_0_xt2_0,    A_xt1_0_yt1_0    ;[11,3]
4102 ||        SUB2    .D1X    A_xh2_3_xh2_2,    B_xl2_3_xl2_2,    A_xl21_1_xl20_1;[11,3]
4103 ||        LDDW    .D2T1   *B_x[B_h2],       A_xh2_3_xh2_2:A_xh2_1_xh2_0        ;[ 1,4]
4104
4105 ;LOOP_Y8:
4106   [A_p0]SUB      .L1     A_p0,             1,                A_p0             ;
4107 ||        ADD     .L2     B_rnd,            B_x_l1_0,         B_x_l1_0         ;[22,2]
4108 ||        DOTP2   .M2     B_xt0_1_yt0_1,    B_co21_si21,      B_x_l1_2         ;[12,3]
4109 ||        ROTL    .M1     A_xt1_0_yt1_0,    16,               A_yt1_0_xt1_0    ;[12,3]
4110 ||        PACKLH2 .S1     A_xl21_1_xl20_1,A_xl21_1_xl20_1,  A_xl20_1_xl21_1;[12,3]
4111 ||        SUB2    .D1X    A_x_3_x_2,        B_xl1_3_xl1_2,    A_xl1_1_xl0_1    ;[12,3]
4112 ||        ADD2    .S2X    B_xl1_1_xl1_0,    A_x_1_x_0,        B_xh1_0_xh0_0    ;[12,3]
4113 ||        LDDW    .D2T2   *B_x[B_l2],       B_xl2_3_xl2_2:B_xl2_1_xl2_0        ;[ 2,4]
```

Fig. 4.16: A part of the assembly code in DSP_16x16r.

Table 4.13: Simulation Data for SRRC_downsample

	Inclusive Cycles	Exclusive Cycles
SRRC_downsample	226	140

[21].

In Table 4.7, the reason for the inefficiency in the SRRC_downsample function is the data movement for the SRRC filter buffer, as shown in Fig. 4.17. We can get proof from the simulation data shown in Table 4.13, where the inclusive cycles are the cycle count for the entire SRRC_downsample function and the exclusive cycles are the cycle count other than the cycles for the functions called inside the SRRC_downsample function. In our program, the function called does the SRRC filtering and the exclusive cycles are just for data movement in the data buffer, so the multiples of real-time for filtering is $(226-140)/52.5 = 1.63$.

By using intrinsics, we can accelerate the speed of data movement. As shown in Fig. 4.17, the function "_amemd8" and "_amemd8_const" are intrinsic functions that provide aligned loads and stores of 8 bytes to memory in single instruction. So we can perform four 16-bit load and store within one instruction. The speedup of the SRRC_downsample function is shown in Table 4.14. Here, we find the Tx_SRRC_filter has obtained huge improvement in performance. The reason is not only due to the use of intrinsics but also because the better coding style by removing of conditionals like the method to improve the framing function as introduced before. More detailed analyses can be found in [5].

4.3 Overall Performance

First, we show the overall system performance after optimization in Tables 4.15 and 4.16 including channel estimation. More detailed introduction about the channel

```
1 //==============================================================//
2 //      Original program in SRRC_downsample                     //
3 //==============================================================//
4 for(j=56;j>3;j--)
5 {
6         SRRC_buffer_real[j]=SRRC_buffer_real[j-4];
7         SRRC_buffer_imag[j]=SRRC_buffer_imag[j-4];
8 }
9
10
11 //==============================================================//
12 //      Modification with Intrinsic function                    //
13 //==============================================================//
14 for(j=53;j>3;j=j-4)
15 {
16         _amemd8(&SRRC_buffer_real[j])=_amemd8_const(&SRRC_buffer_real[j-4]);
17         _amemd8(&SRRC_buffer_imag[j])=_amemd8_const(&SRRC_buffer_imag[j-4]);
18 }
```

Fig. 4.17: Using intrinsics in SRRC filter.

Table 4.14: Performance Improvement of SRRC_downsample by Using Intrinsics

	Original Code		Revised Code		Improvement
	Cycles per Symbol	Cycles per Sample	Cycles per Symbol	Cycles per Sample	
Tx_SRRC_filter	6199452	2690.73	72166	31.32	98.83%
SRRC_downsample	520704	226	288000	125	44.69%

Table 4.15: Optimized Profile of the 802.16a DL Transmitter Function Blocks

	Code Size (Bytes)	Optimized Cycle Count		Original Cycle Count		Multiples of Real-Time
		per Symbol	per Sample	per Symbol	per Sample	
Modulation	544	8310	5.41	188973	123.02	0.10
Framing	3032	25676	15.08	187916	110.40	0.28
IFFT	1420	24360	11.89	35728	17.44	0.22
Tx_SRRC_filter	3728	72166	31.32	6199452	2690.73	0.59

Table 4.16: Optimized Profile of the 802.16a DL Receiver Function Blocks

	Code Size (Bytes)	Optimized Cycle Count		Original Cycle Count		Multiples of Real-Time
		per Symbol	per Sample	per Symbol	per Sample	
SRRC_downsample	348	288000	125	520704	226	2.38
sync	820	1234944	536	1290240	560	10.2
FFT	412	17046	8.32	32256	15.75	0.15
channel estimation	2964	240780	141.46	none	none	2.65
de_framing	2236	7373	4.33	833350	489.62	0.08
de_modulation	3544	125326	81.59	125326	81.59	1.55

estimation function can be found in [6], where the method of channel estimation that we have used in this work is "2D-interpolation."

We can find that most of the functions have better performance than before. We have introduced the techniques for improvement of the framing/de-framing, FFT/IFFT, modulation/de-modulation and SRRC filter functions before. In the synchronization function, the improvement just comes from better setup of the simulator compiler. From the detail information shown in Table 4.17, we find that the pilot correlation function dominates the computational complexity. This is because the need of several times of FFT computation inside the pilot correlation function. Many optimization techniques used in the synchronization function have been discussed in [2], which includes use of shift-FFT, intrinsics, circular buffer, loop unrolling, and skipping of the function execution when it is unnecessary.

Table 4.17: Detailed Information of Synchronization Function

sync	Code Size	Average Execution Cycles/Sample	Multiples of Real-Time
CP_correlation	712	76	1.44
Integer_freq_sync	1448	28	0.53
pilot_corre	3288	204	3.88

4.4 Graphical User Interface

In our implementation of the DSP program, we also implement a host PC graphical interface to control the running of the DSP program and show the results of the synchronization and channel estimation immediately. The program of the GUI interface is from [25]. We upload the DSP program first, and then start running the program. As shown in Fig. 4.18, we can input the SNR and speed (km/hr) first and show the timing synchronization offset, frequency offset, frame synchronization status, and estimated channel response on the graphical interface.

The architecture we adopted is shown in Fig. 4.19 which comes from the original structure in Fig. 4.1 besides the addition of one more block transfer to the host of the results. The contents of the added block buffer are the synchronization and channel estimation information like estimated timing, frequency offset, frame synchronization, and estimated channel response. Because the channel simulator is placed allocated in the host PC side, we can change the channel simulator function easily. In our implementation, we can have noiseless channel, multipath channel, AWGN channel and multipath fading channel, and we can also add frequency offset.

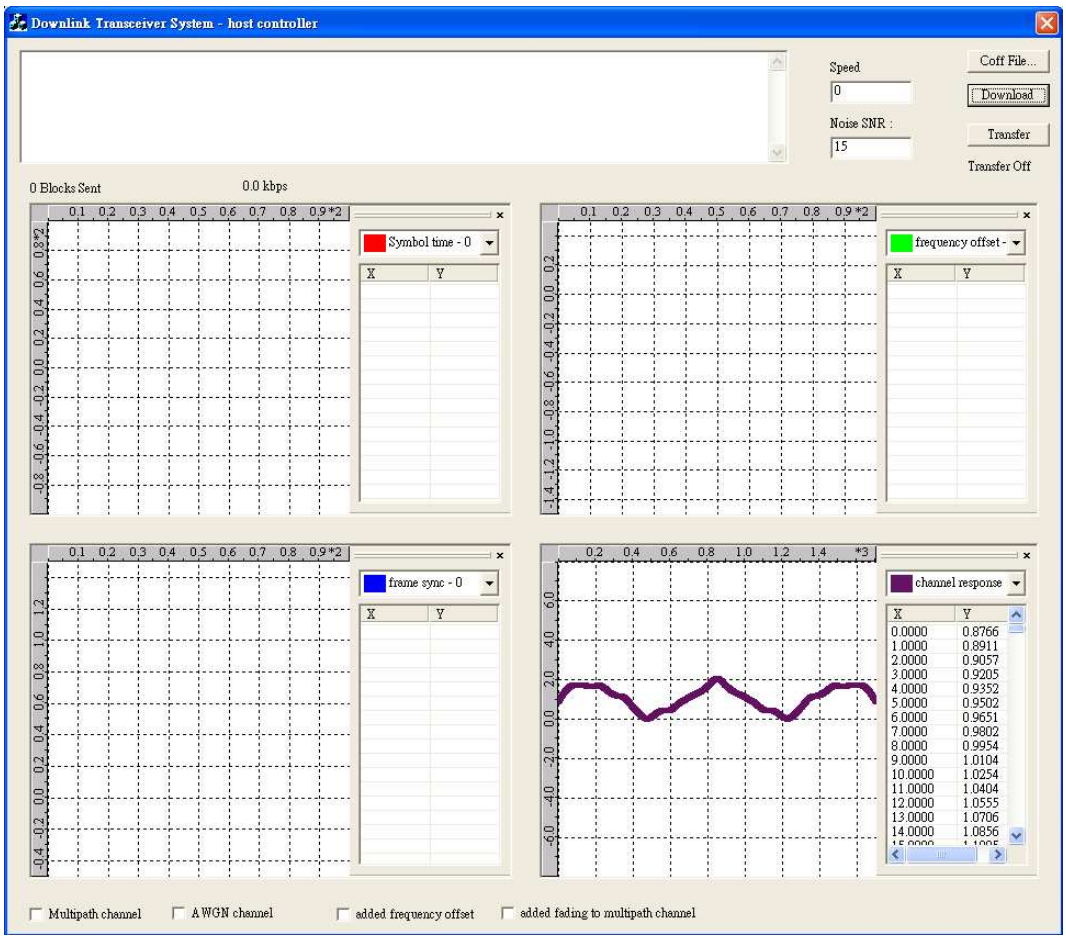

Fig. 4.18: Host PC graphical interface.

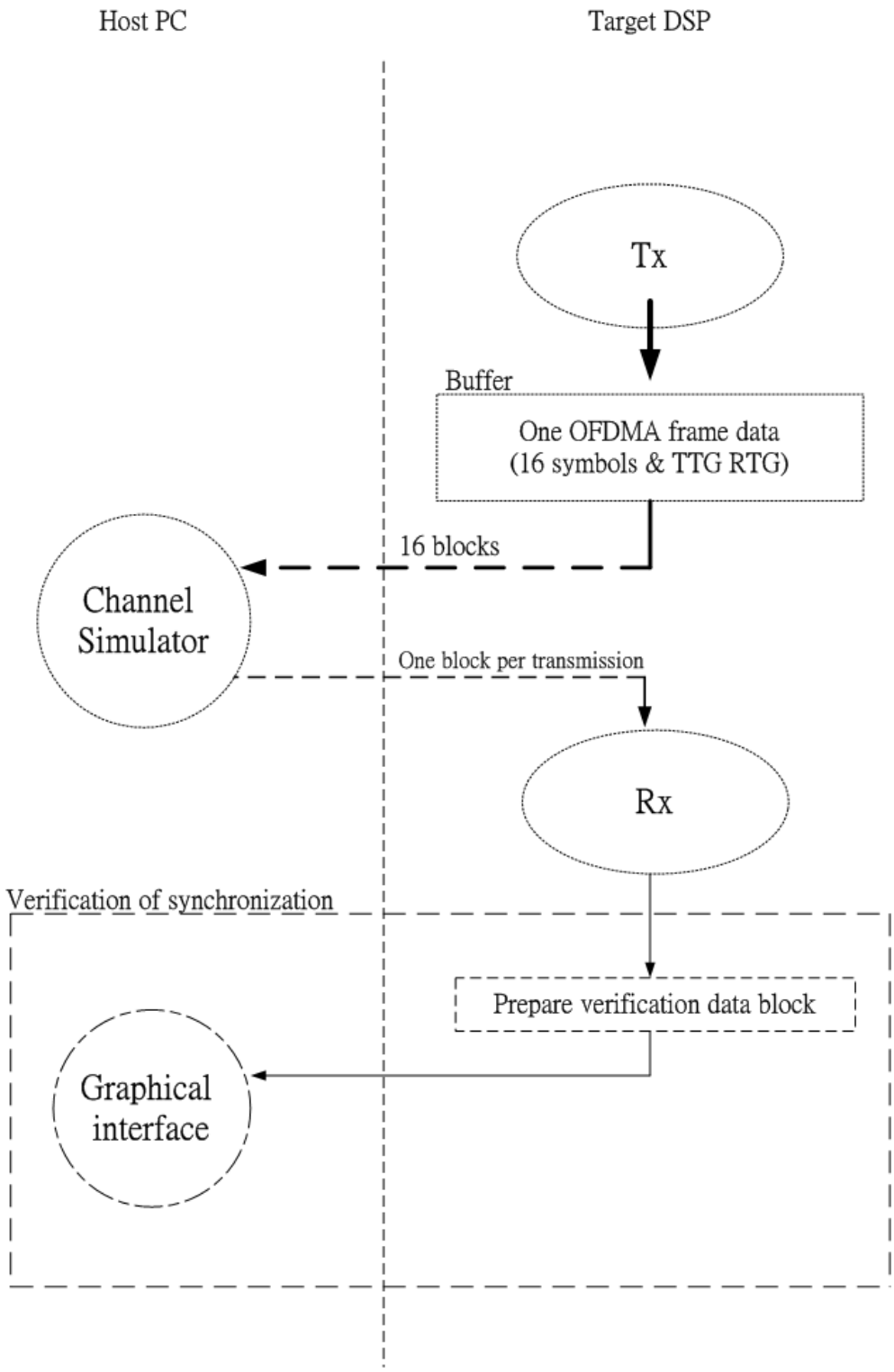

Fig. 4.19: Verification structure of the DL transceiver system.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we considered implementation of a 802.16a DL transceiver system on DSP platform, including transmitter, channel simulator, synchronizer and channel estimator. The overall TDD OFDMA DL system supports QPSK, 16QAM and 64QAM three kinds of modulation schemes. The implementation was based on the simulation results from [1] and programs from [2] and [6].

Synchronization was divided into four stages, which were symbol time synchronization, fractional frequency synchronization, integer frequency synchronization and frame synchronization. The introduction about synchronization was introduced in chapter 2. Data type of the overall system was chosen as 16-bit, which was the most efficient use of 16-bit multiplication for the DSP chip. We modified the inefficient function with better coding style to improve computational complexity such as framing/de-framing functions and used intrinsics to provide faster memory's load/store for SRRC functions. We also replaced original 32-bit FFT/IFFT by 16-bit FFT/IFFT, which was from TI's DSPLIB, to increase computational efficiency. More techniques used in synchronization can be found in [2].

Table 5.1: Improvement After Modifications

	Improvement
modulation	95.60%
framing	86.34%
IFFT	47.17%
FFT	31.82%
de-framing	99.11%

Table 5.2: Execution Time of the DL Receiver

	Practical Execution Time (second/frame)	Real Time Requirement (second/frame)
without optimization option	0.7	0.0032256
with optimization option	0.11	0.0032256

After optimizations, the performance of modulation function was increased 95.60%, framing was increased 86.34%, de-framing was increased 99.11%, FFT function was increased 47.17%, IFFT function was increased 31.82%, Tx_SRRC_filter function was increased 98.83%, SRRC_downsampple function was increased 44.69%, as shown in Table 5.1. Except for synchronization, SRRC_downsample, channel estimation, and de-modulation functions, other functions were all satisfied the real-time requirements.

Besides, we also calculate the execution time of the receiver from the host, as shown in Table. 5.2. The execution time we estimated is from the host side clock timer. It is not the real timer on the DSP environment, but can be a reference time to the program flow. The symbol duration is 201.6μs per symbol, so the real time requirement is 201.6*16μs per frame. If we do not open the compiler optimization option, the execution of the program is quite slow, which is almost 0.7/0.0032256 = 217 times of the real time requirement. After opening the optimization option, it is 0.11/0.0032256 = 34 times of the real time requirement.

5.2 Potential Future Work

In this thesis, our main goal is implement the DL system on the DSP platform. And we have been optimized the inefficient functions, but the synchronization function is still complex. The bottleneck of synchronization function is the pilot correlation. This is because that we have to do 65 times of FFT in initial synchronization or 33 times of FFT in tracking mode. Although the shift-FFT[2] have been used to reduce the computational complexity, the computation of FFT is still a huge loading in synchronization. Besides complexity, we still find that the pilot correlation function may search the wrong symbol time even without adding noise and channel. If we will modify the synchronization algorithm, we suggest that we can modify the frame synchronization algorithm first for this reason. In IEEE Std 802.16-2004, the preamble is allocated in front of the DL subframe and it may help us to improve frame synchronization algorithm.

To fulfill the real time requirement, we can still make more effort on the program. We may notice the coding style to prevent the waste of the computation unnecessary or use intrinsics to accelerate the program. One another way is skipping a function call when it is idle operation. But we may notice that if we adopt this method, it may make lots of conditionals in the program and then make the compiler hard to do the optimization. The tradeoff should be estimated carefully.

In our DSP program, we do not implement FEC encoder/decoder yet. We can find the associated reference in [5].

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] T.-S. Chiang, "Study and DSP implementation of IEEE 802.16a TDD OFDM downlink synchronization," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., July 2004.

[3] 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, Apr. 1, 2003.

[4] IEEE Std 802.16-2004, *IEEE Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Fixed Broadband Wireless Access Systems.* New York: IEEE, Oct. 1, 2004.

[5] C.-C. Tung, "IEEE 802.16a OFDMA TDD uplink transceiver tystem integration and optimization on DSP platform," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2005.

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

[7] 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.

[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] 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.

[10] C. D. Murphy, "Low-complexity FFT structure for OFDM transceivers," *IEEE Trans. Commun.*, vol. 50, no. 12, pp. 1878–1881, Dec. 2002.

[11] H. V. Sorenson, C. S. Burrus, "Efficient computation of the DFT with only a subset of input or output points," *IEEE Trans. Signal Processing,* vol. 41, no. 3, pp. 1184–1200, Mar. 1993.

[12] J. D. Markel, "FFT pruning," *IEEE Trans. Audio Electroacoust.*, vol. AU-19, no. 4, pp. 305–311, Dec. 1971.

[13] G. Goertzel, "An algorithm for the evaluation of finite trigonometric series," *Amer. Math. Monthly,* vol. 65, no. 1, pp. 34–35, Jan. 1958.

[14] 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.

[15] Innovative Integration, *Quixote User's Manual*, June 2004.

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

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

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

[19] Texas Instruments, *TMS320C6000 DSP Cache User's Guide*. Literature number SPRU656A, May 2003.

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

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

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

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

[24] IEEE 802.16 Working Group, *IEEE 802.16 Working Group Website*, http://www.ieee802.org/16/.

[25] The Code Project, *The Code Project Website*, http://www.codeproject.com/miscctrl/graph2d.asp.

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

自傳

陳昱昇，男，民國七十年一月二十五日出生於台灣省桃園縣。高中就讀於桃園武陵高中，民國 92 年六月畢業於交通大學電子工程學系，並於九月進入交通大學電子工程研究所繼續就讀，於民國 94 年取得碩士學位，論文題目為:『IEEE 802.16a 分時雙工正交分頻多重進接下行傳收系統之數位訊號處理器軟體實現與整合』，是有關無線通訊領域的相關研究。