

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之上行
同步技術研討與在數位訊號處理器上的實現



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

研究生：林筱晴

指導教授：林大衛 博士

中華民國九十三年六月

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

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

研究生：林筱晴

Student : Hsiao Ching Lin

指導教授：林大衛 博士

Advisor : Dr. David W. Lin



Submitted to Department of Electronics Engineering & Institute of Electronics
College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

in

Electronics Engineering

June 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年六月

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

研究生：林筱晴

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



正交分頻多工 (OFDM) 技術可有效地解決通訊系統中的許多問題，如多重路徑衰落、窄頻干擾等，多用戶正交分頻多工系統能依據使用者之需求將頻寬作更有效之分配。在本篇論文中，我們使用數位訊號處理器去實現分時雙工正交分頻多重進接環境下的上行同步機制。此數位訊號處理器的環境是 Innovative Integration 公司的 Quixote 個人電腦插板，其上裝置為德州儀器公司的 TMS320C6416，是個擁有強大數學運算功能的處理器。

我們所處理的上行同步架構如下。上行傳輸需要作時間同步以偵測信號到達的時間，如果估測錯誤會降低間格區間 (guard interval) 用來防止多重路徑延遲造成符元間 (ISI) 干擾的能力。我們將上行同步分為兩級，第一級利用 OFDM 系統特有之間格區間 (guard interval) 估測 OFDM 符元 (symbol) 大略的開始時間，此乃由於間格區間使單一符元內具有高度的自相關。第二級利用上行傳輸前置資

訊 (preamble) 判斷估測 OFDM 符元(symbol) 精確的開始時間。我們嘗試用兩種方式作時間同步的第二級，分別為在時間域及頻率域對收到的訊號與上行傳輸前置資訊 (preamble) 作相關性 (correlation) 分析，找到具有最大相關性的時間。

為了降低在數位訊號處理器上的運算複雜度，我們先將原始的浮點運算 C 程式版本修改為實數運算的程式版本，接著再考慮數位訊號處理器—TMS320C64X 的特性來修改之前的程式。最後，我們在數位訊號處理器上加速了上行同步機制達 374 倍。

在本篇論文中，我們首先簡介分時雙工正交分頻多重進接環境下的上行同步機制。接著，我們介紹數位訊號處理器的運作環境。最後，我們描述利用數位訊號處理器的特點以加速程式的方法並且提供一些關於執行速度與同步機制效能方面的實驗結果。



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

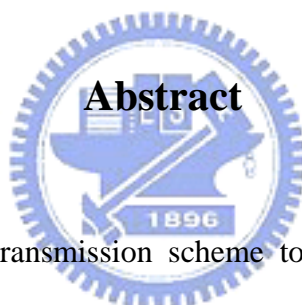
Student: Hsiao Ching Lin

Advisor: Dr. David W. Lin

Department of Electronics Engineering

Institute of Electronics

National Chiao Tung University



OFDM is an effective transmission scheme to cope with many transmission impairments, such as multipath fading and narrowband interference. Multiuser OFDM can provide highly flexible to allocate the bandwidth according to the needs of users. In this thesis, we focus on the TDD OFDMA uplink synchronization based on IEEE 802.16a. We use digital signal processor to implement uplink synchronization schemes. The digital signal processing environment is Innovative Integration's Quixote personal computer card, which hosts Texas Instruments' TMS320C6416 which is a powerful signal processor with strong arithmetic operation capability.

Time synchronization is performed to detect the start time of symbols for uplink synchronization. Time synchronization errors would decrease the ability of guard interval to avoid ISI introduced by multipath channel. There are two stages in the uplink synchronization. The first stages use the guard interval to estimate the OFDM

symbol start time roughly. The reason of using the guard interval is that it provides strong autocorrelation within an OFDM symbol. The second stage uses the preamble information to detect the symbol start time exactly. We present two schemes to do the second stage. One is using the correlation of received signal with preamble in the time domain and the other is in the frequency domain. The symbol start time is determined as the location with maximum correlation value.

In order to decrease the computation complexity on the DSP, we rewrite the original floating-point C programs to fixed-point version and further refine our codes by taking into account the features of the DSP chip, TMS320C6416, to produce a more efficient program. Overall, the final version for uplink synchronization schemes is 374 times faster than the original version.

In this thesis, we first introduce to the TDD OFDMA uplink synchronization schemes. Second, we describe the environment of DSP implementation. Finally, we discuss the optimization methods using the features of C64x and present experimental results on the speed and the synchronization performance.

誌謝

本論文承蒙恩師林大衛教授細心的指導與教誨，方得以順利完成。在兩年的研究所生涯中，林教授不僅在學術研究上予以學生指導，在研究態度上亦給予相當多的建議，在此對林教授獻上最大的感激之意。

此外，感謝通訊電子與訊號處理實驗室所有的成員，包含各位師長、同學、學長姐與學弟妹們。我要感謝吳俊榮學長、洪昆健學長與林郁男學長給予我在研究過程上的指導與建議，還有宗書、盈縈、明哲、明瑋、建統、仰哲、岳賢等同學與學弟妹與我彼此勉勵、互相討論，讓我在這兩年的研究生涯充滿歡樂與回憶。

最後，我要感謝我的家人和朋友，在我的求學過程當中總是不斷的鼓勵我，提供我心靈上的支持，陪我走過我的不安、徬徨、憂愁，也與我分享我的驕傲、快樂、心得。

在此，我誠摯的對這些幫助過我的人表達我的謝意。



林筱晴

民國九十三年六月 於新竹

Contents

1	Introduction	1
2	Techniques for Uplink Synchronization	3
2.1	Background	3
2.2	Overview of IEEE 802.16a	5
2.2.1	OFDMA Carrier Allocation	5
2.2.2	OFDMA Frame Structure	6
2.2.3	System Architecture	8
2.3	UL Synchronization Approach	11
2.4	UL Synchronization	11
2.4.1	Stage I: Using CP Correlation Property	11
2.4.2	Stage II: Using Preamble Correlation Property	13
2.5	UL Synchronization Result	17
2.5.1	Preamble Correlation in Frequency Domain Approach	20
2.5.2	Preamble Correlation in Time Domain Approach	21
2.5.3	Comparison of UL Synchronization Using Time Domain Approach and Frequency Domain Approach	23
3	DSP Introduction	26
3.1	DSP Board Introduction [11]	26
3.2	DSP Core Introduction [13]	28
3.3	Data Transmission Mechanism [15]	37
3.4	Code Composer Studio Introduction [16], [17]	40
4	DSP Implementation	42
4.1	Procedure of the Implementation Work	42
4.2	Optimization Method	43
4.2.1	Configuring the Setting of Compiler Options	43
4.2.2	Using Intrinsics [19]	46
4.2.3	Software Pipelining	46
4.2.4	Data Type Modification	48
4.3	Framing/Deframing Structure	50
4.3.1	Framing	50
4.3.2	Deframing	51
4.4	IFFT/FFT Structure	53

4.5	Transmission Filtering	61
4.5.1	Oversampling and SRRC Filter in the Transmitter	61
4.5.2	Downsampling and SRRC Filter in the Receiver	61
4.6	Uplink Synchronization Using Time Domain Approach	64
4.6.1	CP_Correlation	64
4.6.2	Preamble_correlation	65
4.6.3	Complexity Analysis	72
4.7	Conclusion in Optimization	74
5	Conclusion and Future Work	76
5.1	Conclusion	76
5.2	Potential Future Work	77



List of Figures

2.1	OFDM symbol structure in time.	3
2.2	Illustration of carrier usage in OFDMA UL.	5
2.3	Carrier allocation in the OFDMA UL (from [4]).	6
2.4	Frame structure of the TDD OFDMA system (from [4]).	7
2.5	UL transmitter structure.	8
2.6	UL receiver structure.	9
2.7	Pseudo Random Binary Sequence (PRBS) generator for pilot modulation.	10
2.8	Method of UL synchronization.	11
2.9	The structure of the ML time offset estimator (from [8]).	12
2.10	The structure of the proposed symbol time estimator.	13
2.11	Three UL signals arrive at different times, and the CP correlation peak may occur between them (from [5]).	14
2.12	The received samples and the time plan of the UL synchronization stage II (from [5]).	14
2.13	Illustration of UL synchronization stage II in frequency domain (from [5]).	15
2.14	Illustration of UL synchronization stage II in time domain (from [5]).	16
2.15	Frame structure used in UL synchronization.	19
2.16	Error distribution under different maximum Doppler shifts using frequency domain approach.	20
2.17	Error distribution under different maximum Doppler shifts using time domain approach.	21
2.18	The multipath delay spread and the relative average power (The definition of the Ref in the next figure).	22
2.19	Performance of UL time synchronization under different Doppler spreads.	23
2.20	Comparison of UL synchronization using frequency domain and time domain approach at velocity of 60 km/hr.	24
3.1	Block diagram of Quixote (from [12]).	27
3.2	Technical specification of Quixote (from [12]).	29
3.3	Block diagram for C6416 DSP (from [14]).	32
3.4	TMS320C6416 DSP core data paths (from [14]).	33
3.5	Block diagram for C62x and C64x DSP core (from [15]).	37
3.6	Block diagram of DSP streaming mode (from [11]).	39
3.7	Simplified code composer studio development flow (from [17]).	40
4.1	Code development flow of C6000 (from [19]).	44

4.2	C64x fixed-point pipeline phases.	47
4.3	The fixed-point data formats at the TX side and the RX side.	49
4.4	Error distribution under different maximum Doppler shifts using time domain approach in fixed-point version.	50
4.5	C code for PRBS generator.	51
4.6	Compiler's feedback for PRBS generator loop.	52
4.7	Two versions of C programs for framing.	53
4.8	Compiler's feedback for framing loop before and after optimization.	54
4.9	A part of C code for framing.	55
4.10	A part of C code for deframing.	55
4.11	A part of assembly code for DSP_fft32x32.	59
4.12	C code for mul_sum() in Tx_SRRC().	62
4.13	C code and compiler's feedback for mul_sum() loop.	63
4.14	C code and compiler's feedback for Rx_SRRC() loop.	64
4.15	C code in CP_correlation() before optimization.	65
4.16	C code in CP_correlation() after optimization.	66
4.17	Compiler's feedback for CP_correlation() loop before optimization.	67
4.18	Compiler's feedback for CP_correlation() loop after optimization.	68
4.19	C code in Preamble_correlation() before optimization.	69
4.20	C code in Preamble_correlation() after optimization.	70
4.21	Compiler's feedback for Preamble_correlation() loop before and after optimization.	71
4.22	Comparison between floating-point version and fixed-point version.	74

List of Tables

2.1	OFDMA UL Carrier Allocations	7
2.2	Complexity for ML estimator and the Proposed Symbol Time Estimator .	13
2.3	Comparisons of Computational Complexity for Different FFT Algorithms	17
2.4	Complexity for Time Domain Approach and Frequency Domain Approach	17
2.5	System Parameters Used in Our Study	18
2.6	Characteristics of the ETSI “Vehicular A” Channel Environment	18
2.7	Relations Between Spread and Maximum Doppler Shift at Carrier Frequency 6GHz and Subcarrier Spacing 5.58 kHz	19
3.1	Characteristics of TI C6416T Processors (from [14])	30
3.2	Functional Units (.L, .S) and Operations Performed (from [15])	34
3.3	Functional Units (.M, .D) and Operations Performed (from [15])	36
4.1	Compiler Options to Avoid on Performance Critical Code (from [19]) . .	45
4.2	Compiler Options for Performance (from [19])	47
4.3	Breakdown of Clock Cycles for Framing()	52
4.4	Breakdown of Clock Cycles for Deframing()	55
4.5	IFFT/FFT Function	56
4.6	Comparison of Different IFFT/FFT	58
4.7	Complexity and Performance of IFFT/FFT Implementation	58
4.8	Used Compiler Intrinsics in DSP_ifft32x32/DSP_fft32x32	59
4.9	Breakdown of Clock Cycles for IFFT()	60
4.10	Breakdown of Clock Cycles for FFT()	60
4.11	Breakdown of Clock Cycles for TX_SRRC()	61
4.12	Breakdown of Clock Cycles for RX_SRRC()	64
4.13	Breakdown of Clock Cycles for CP_correlation()	66
4.14	Breakdown of Clock Cycles for Preamble_correlation()	72
4.15	Complexity and Performance of CP Correlation Implementation	73
4.16	Complexity and Performance of Preamble Correlation Implementation . .	74

Chapter 1

Introduction

Orthogonal frequency-division multiple access (OFDMA) technique has attracted serious attention in the last few years and has been proposed for the uplink of wireless communication systems [1], [2] and cable TV (CATV) networks [3]. In this thesis, we focus on uplink synchronization based on IEEE 802.16a WirelessMAN OFDMA system [4].

Our intent is to implement the uplink synchronization scheme by using digital signal processor (DSP). In order to verify the accuracy of the fixed-point uplink synchronization scheme, the framing/deframing structure, IFFT/FFT block and Tx/Rx SRRC filter have been also implemented.

The environment of our DSP implementation involves a host PC, DSP board and DSP chip on the board. The DSP chip is Texas Instruments (TI)'s TMS320C6416. The TMS320C6416 is a fixed-point DSP with 1.67 ns instruction cycle time. It adopts the advanced VelociTI Very Long Instruction Word (VLIW) architecture that enables sustained throughput of eight instructions in parallel and thus allows the processor running faster. In addition, the C64x device comes with on-chip program and data memories, which may be configured as cache on some devices. The DSP board we use is Innovative Integration (II)'s Quixote. It is a PCI bus compatible DSP card housing one TI TMS320C3416 processor.

Our work is based on the code from [5]. In order to reduce the computation complexity, we rewrite the original 32-bit floating-point version to 16-bit fixed-point version. We

also do some optimization methods to facilitate better parallelism after compilation.

The thesis is organized as follows. In chapter 2, we introduce the techniques for uplink synchronization in detail. Chapter 3 introduces the DSP board and the DSP chip. Chapter 4 discusses the optimization methods based on DSP properties and presents the optimization results. Finally, the conclusion is given in chapter 5 and we point out some potential future work.



Chapter 2

Techniques for Uplink Synchronization

2.1 Background

The basic idea of orthogonal frequency-division multiplexing (OFDM) is to divide the available spectrum into a number of subchannels. To obtain high spectral efficiency, the frequency response of the subchannels are overlapping but orthogonal, hence the name OFDM. By introducing a cyclic prefix (CP), the orthogonality can be completely maintained even through the signal passes through a time-dispersive channel. The cyclic prefix is a copy of the last part of the OFDM symbol which is prepended to the transmitted symbol, as shown in Figure 2.1 [6].

Orthogonal frequency-division multiaccess (OFDMA) is a multiplexing technique in which several users simultaneously transmit their own data by modulating an exclusive set of orthogonal subcarriers. Its main advantage is that separating different users through frequency-division multiaccess (FDMA) techniques at the subcarrier level can mitigate

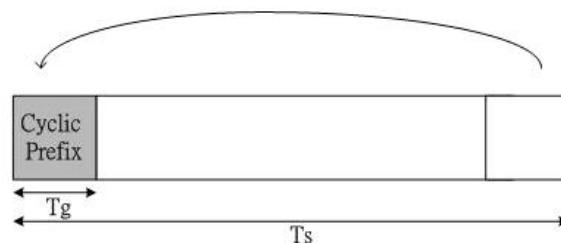


Figure 2.1: OFDM symbol structure in time.

multiaccess interference (MAI) within a cell [7]. Also, compared with single-carrier multiaccess, OFDMA offers increased robustness to narrowband interferences, allows straightforward dynamic channel assignment, and does not need adaptive time-domain equalizers, since channel estimation is performed in the frequency domain through one-tap multipliers.

For all this to be true, however, proper time and frequency synchronization is necessary to maintain orthogonality among the active users. Frequency offset due to Doppler shifts and/or oscillator instabilities produce interchannel interference (ICI) that must be counteracted to avoid severe error-rate degradations. Timing errors result in intersymbol interference (ISI) between consecutive OFDM symbols. Using a guard interval (cyclic prefix) provides intrinsic protection against timing errors at the expense of some reduction in the data throughput due to the extra overhead. However, timing accuracy becomes a stringent requirement in practical applications where, to minimize the overhead, the cyclic prefix is made only just greater than the length of the channel impulse response (CIR).

In this thesis, we consider the IEEE 802.16a WirelessMAN OFDMA system [4]. According to the IEEE 802.16a standard, the duplexing method of OFDMA system in 2–11 GHz band shall be either FDD or TDD in licensed bands and TDD in license-exempt bands. The traffic requirements of the downlink (DL) and uplink (UL) transmissions are usually different. Compared with FDD mode, TDD mode supports more flexibility for different traffic transport capacity. That is why we choose to study the TDD mode in this thesis.

In this work, we focus on IEEE 802.16a TDD OFDMA uplink synchronization techniques. According to IEEE 802.16a standard, all SSs shall acquire and adjust their timing such that all uplink OFDM symbols arrive time coincident at the base station to a accuracy of 50% of the minimum guard-interval or better. For the same reason, both the transmitted center frequency and the symbol clock frequency shall be synchronized to the BS with a

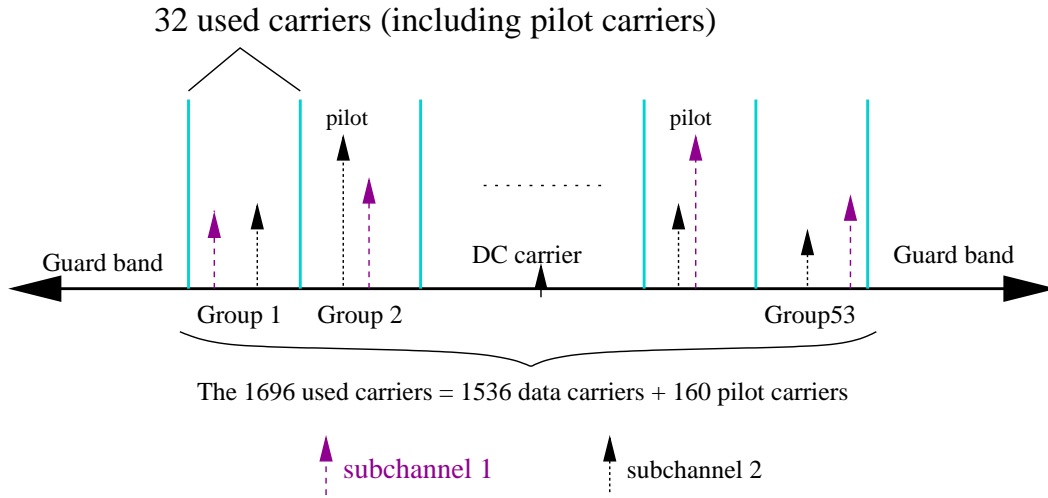


Figure 2.2: Illustration of carrier usage in OFDMA UL.

tolerance of maximum 2% of the carrier spacing, which equals to 111.6 Hz in our work. These limitations are very useful for UL synchronization scheme.

2.2 Overview of IEEE 802.16a

2.2.1 OFDMA Carrier Allocation

The FFT size used in the 802.16a OFDMA system is 2048, so there are 2048 carriers in a channel. These carriers are divided into as three types: data carriers that are used for data transmission, pilot carriers for various estimation purposes, and null carriers (guard bands and DC carrier) which transmit nothing at all. The data and pilot carriers together are termed the used carriers for they transmit useful information. The allocation is as shown in Figure 2.2 for UL.

In the uplink, the set of used carriers is first partitioned into 32 subchannels, and then the pilot carriers are allocated within each subchannel. Each subchannel may be transmitted from a different SS. The used carriers of the UL transmission are partitioned into fixed-location pilots, variable location pilots, and data subchannels. Within each subchannel, there are 48 data carriers, 1 fixed-location pilot carrier, and 4 variable-location pilot carriers. The allocation of pilot carriers is illustrated in Fig. 2.3.

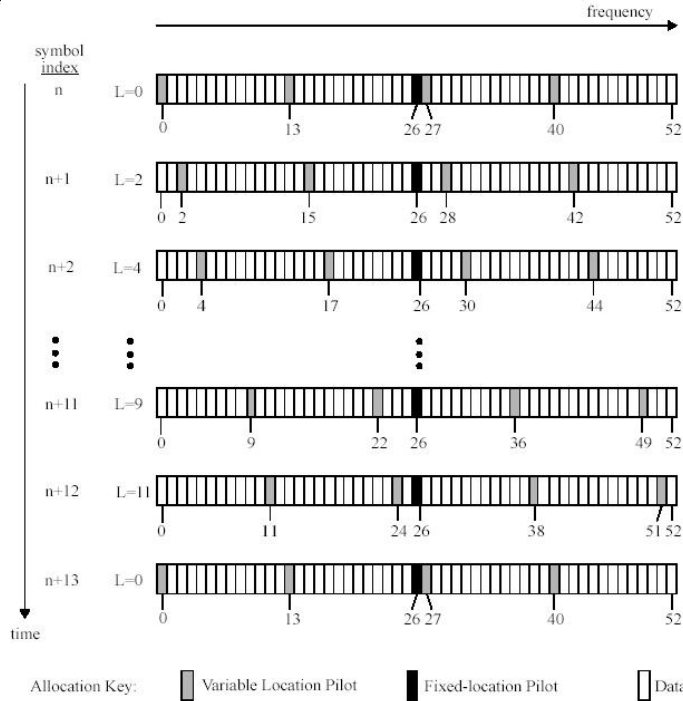


Figure 2.3: Carrier allocation in the OFDMA UL (from [4]).

The fixed-location pilot is always at carrier 26 in the subchannel. The variable-location pilots change locations in each symbol, repeating every 13 symbols, according to $L_k = 0, 2, 4, 6, 8, 10, 12, 1, 3, 5, 7, 9, 11$, where $k = 0$ to 12. For $k = 0$ the variable location pilots are positioned at indices 0, 13, 27, 40. For other k values these locations change by adding L_k to each index. Thus due to the motion of the variable-location pilots, the locations of data carriers also change with each symbol [4]. The parameters of the UL are also shown in Table 2.1.

2.2.2 OFDMA Frame Structure

Figure 2.4 shows the TDD OFDMA frame structure. The frame structure is built from BS and SS transmissions. Each TDD OFDMA frame is composed of a DL subframe and a UL subframe. The duration of a frame is allowed from 2 ms to 20 ms and is specified by the frame duration code. A subframe contains several transmission bursts, which are composed of multiples of FEC blocks.

Table 2.1: OFDMA UL Carrier Allocations

Parameter	UL Value
Number of DC carriers	1
Number of guard carriers, left	176
Number of guard carriers, right	175
Number of used carriers (N_{used})	1696
Number of total carriers (N)	2048
$N_{varLocPilots}$	128
Number of fixed-location pilots	32
Number of variable-location pilots which coincide with fixed-location pilots	0
Number of total pilots	160
Number of data carriers	1536
$N_{subchannels}$	32
$N_{subcarriers}$ per subchannel	53
Number of data carriers per subchannel	48
$PermutationBase_0$	3,18,2,8,16,10,11,15,26,22, 6, 9,27,20,25,1,29,7,21,5,28,31,23,17,4,24,0,13,21,19,14,30

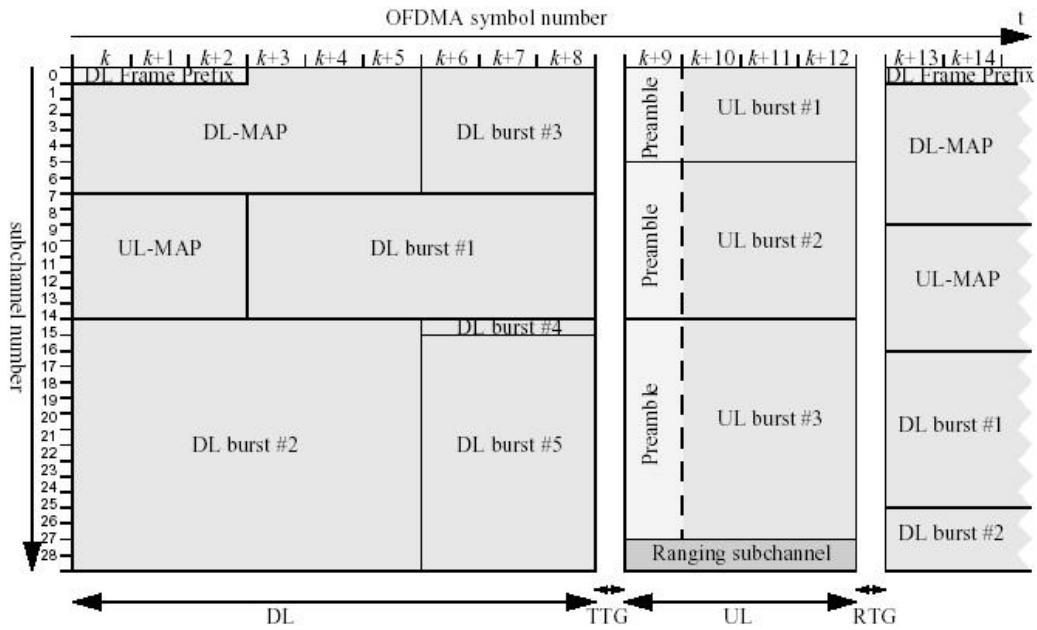


Figure 2.4: Frame structure of the TDD OFDMA system (from [4]).

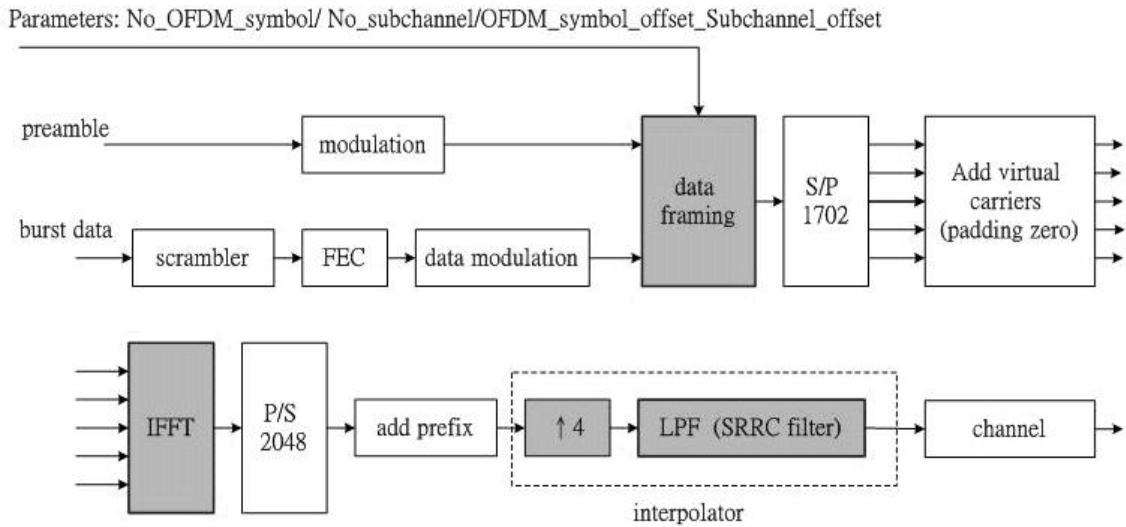


Figure 2.5: UL transmitter structure.

From the UL-MAPs, the subscribers know their usable subchannels and transmission time. The first symbol of the UL subframe is the all-pilot preamble where the SS should send a preamble on all its allocated subchannels. The number of symbols of the UL is $3N + 1$, one for the preamble and the others data transmitted bursts. The Tx/Rx transition gap (TTG) and Rx/Tx transition gap (RTG) shall be inserted between the downlink and uplink and at the end of each frame respectively to allow the BS to turn around. After the TTG, the BS receiver shall look for the first symbols of a UL burst. After the RTG, the SS receivers shall look for the first symbols of QPSK modulated data in the DL burst. TTG and RTG shall be at least $5 \mu s$ and an integer multiple of four samples in duration.

2.2.3 System Architecture

Figure 2.5 shows the system structure of the UL transmitter.

The data is scrambled and FEC coded, while the preambles and pilots are not coded. The BS has to receive various bursts from different SSs at the same time. Each SS has to support one kind of coding and modulation types in a frame. The framing is used to arrange the coded data, MAPs, preamble or pilots to the corresponding carriers and

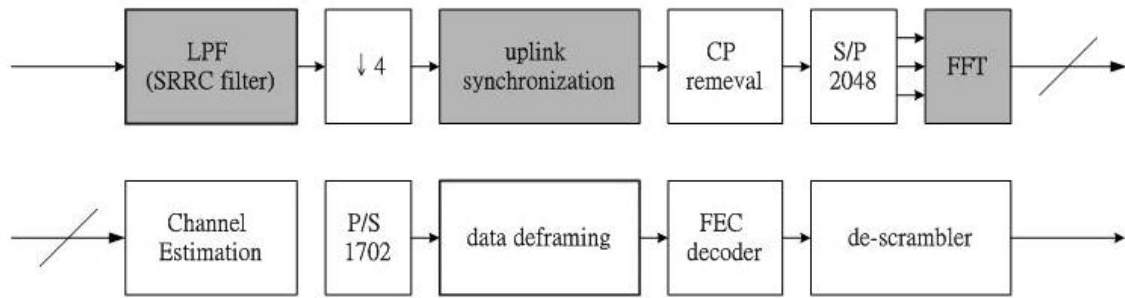


Figure 2.6: UL receiver structure.

symbols following the specified frame structure and carrier allocation. After framing, the used carriers and null carriers are ordered properly and fed into the 2048-point IFFT block in parallel. The IFFT results are output sequentially and shaped by the pulse shaping block.

The system structure of the UL receiver is as shown in Figure 2.6. The receiver operation is in some sense the reverse of the transmitter. Two blocks are added: synchronizer and channel estimator. These two blocks and the FEC decoder are the most sophisticated elements of the receiver.

In framing/deframing structure, we need some information such as carrier allocation and UL parameters shown in Table 2.1. Pilot carriers shall be inserted into each data burst in order to constitute the symbol and they shall be modulated according to their carrier location within the OFDMA 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 Figure 2.7 shows.

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 boosting of 2.5 dB over the average power of each data tone. The pilot carriers

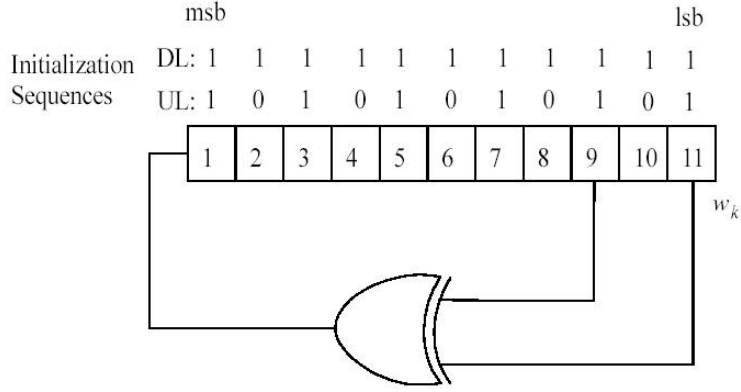


Figure 2.7: Pseudo Random Binary Sequence (PRBS) generator for pilot modulation.

shall be modulated according to the following formulas:

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

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

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

The details for the Tx/Rx SRRC filter we use are based on [5]. In order to provide the ability to simulate path delays at non-integer sample times, an interpolator is added to the transmitter to yield 4-times oversampled transmitter output. As the ideal lowpass interpolation filter cannot be implemented exactly, the easier realized square root raised cosine (SRRC) filter is used instead. The impulse response of the filter is given by

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

where α is the roll-off factor. The reason of adopting the SRRC filter is that for this filter the transmitter and receiver filters are matched to each other and there is no inter-sample interference introduced in the receiver. In our work, the pulse-shaping block is regard as the interpolator with 4-time oversampling and the roll-off factor 0.155 SRRC filter.

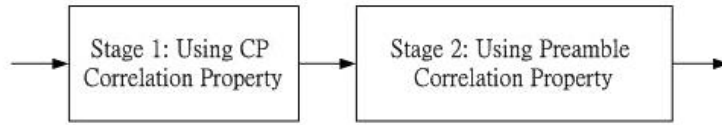


Figure 2.8: Method of UL synchronization.

2.3 UL Synchronization Approach

After doing DL synchronization, the mobile enters the time and frequency grid with a low offset in time and frequency. The UL synchronization is unlike the DL synchronization which requires complex frame synchronization at initialization. No frequency synchronization is done in UL normal transmission. What the BS has to do is to detect the exact UL symbol arrival time. The BS shall detect the arrival time of the first coming signal to keep the symbol ISI free.

There are two stages in UL synchronization, which is shown in Figure 2.8. The first stage uses cyclic prefix information to detect symbol start time roughly. The second stage uses preamble information to detect symbol start time exactly. We present two schemes to do the second stage. One is using the correlation of received signal with preamble in the time domain and the other is in the frequency domain. The symbol start time is determined as the location with maximum correlation value.

2.4 UL Synchronization

2.4.1 Stage I: Using CP Correlation Property

OFDM/OFDMA signals have strong auto-correlation properties of the waveforms. This autocorrelation is a consequence of the cyclic prefix part of the waveform. The algorithm in [2] and [8] uses the maximum likelihood (ML) criterion to estimate the time offset. Under the assumption that received samples are jointly Gaussian distributed and uncorrelated except for the pairs of identical samples contained in the cyclic prefix, symbol time

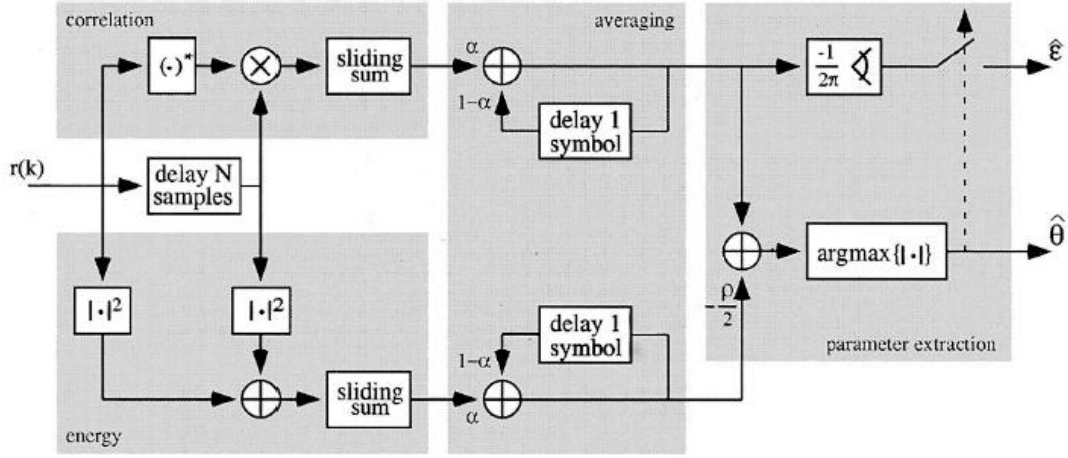


Figure 2.9: The structure of the ML time offset estimator (from [8]).

offset $\hat{\theta}$ is given by

$$\hat{\theta} = \arg \max \{ |\Gamma(\theta)| - \rho \Phi(\theta) \}, \quad (2.1)$$

where

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

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

and $\rho = \frac{SNR}{SNR+1}$ with SNR being signal to noise ratio. Estimator (2.1) exploits the correlation introduced by the cyclic prefix to estimate the offsets. The structure of the time offset estimator is shown in Figure 2.9. Its strength is that it is independent of the modulation and it does not need pilot symbols. It is a one-shot estimator in the sense that the estimates are based on the observation of one OFDM symbol.

The symbol time offset estimator can be viewed as consisting of two parts: the correlation $\Gamma(\theta)$ which correlates the received sampled baseband signal, r , with a delayed version of itself, and a part that compensates for the difference in energy in the correlated samples. In order to reduce the complexity, we only employ the correlation part in our work. 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 $r(\theta), \dots, r(\theta+N+L-1)$ are

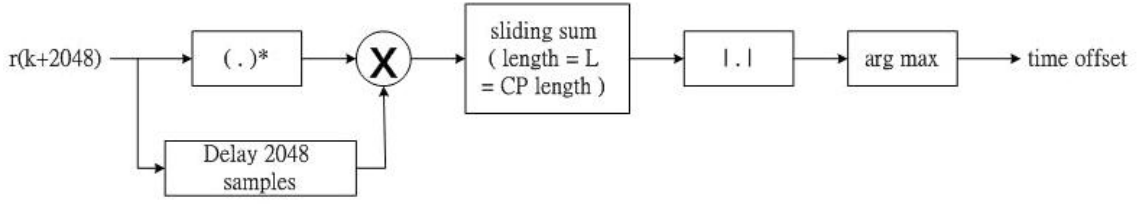


Figure 2.10: The structure of the proposed symbol time estimator.

Table 2.2: Complexity for ML estimator and the Proposed Symbol Time Estimator

	No. of Real Multiplications	No. of Real Additions
ML time offset estimator	2048	2045
Proposed symbol time estimator	1024	1022

all within the same OFDM symbol. Then, the symbol time offset estimator becomes

$$\hat{\theta} = \arg \max \left| \sum_{k=\theta}^{\theta+L-1} r(k)r^*(k+N) \right|. \quad (2.2)$$

Figure 2.10 shows the structure of this estimator.

Table 2.2 shows a comparison of the complexity for ML time offset estimator and the proposed symbol time estimator. In this table, we consider the complexity for the first 256 samples.

Different users' transmitted signals may not arrive at the same time, but the correlation peak may occur between them, as shown in Figure 2.11 for an example of three users. If we use the detected peak location as the symbol start time, the corresponding useful time will include a part of the guard interval of the next symbol for the earlier arriving signals. Therefore, we have to find the exact instant of the first arriving signal to avoid ISI. This is why we use preamble information in stage II. In stage II, we use preamble correlation property to detect the symbol start time exactly.

2.4.2 Stage II: Using Preamble Correlation Property

In stage I, the symbol (frame) start time is roughly detected by using CP correlation peak. We know that the actual arrival time of the first arriving signal is likely before the detected

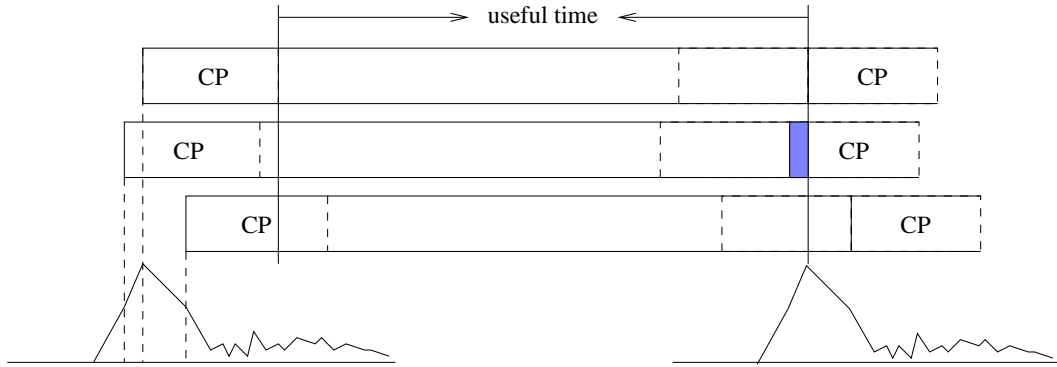


Figure 2.11: Three UL signals arrive at different times, and the CP correlation peak may occur between them (from [5]).

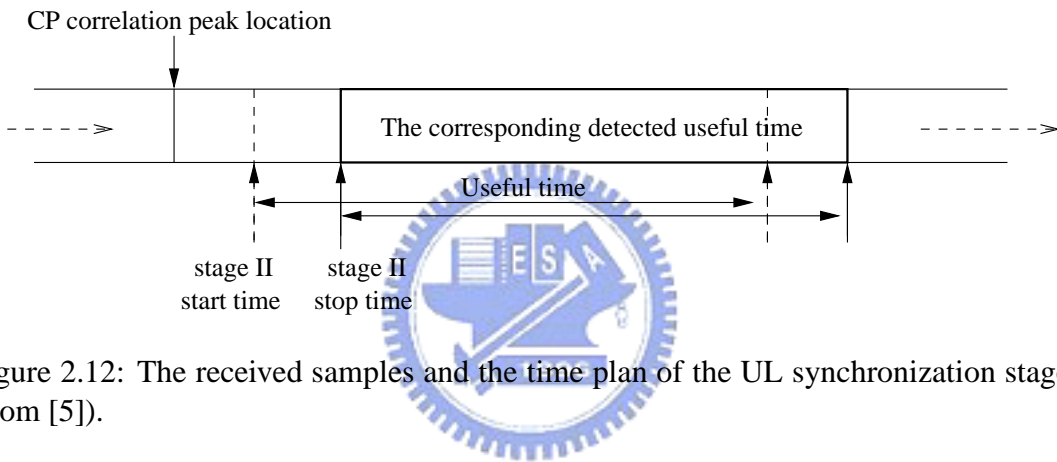


Figure 2.12: The received samples and the time plan of the UL synchronization stage II (from [5]).

time. In stage II, we use preamble information to detect the symbol start time exactly. We present two schemes to do stage II. One is using the correlation of received signal with preamble in the frequency domain and the other is in the time domain. Figure 2.12 shows the received samples of the BS and the time relation for stage II.

As the user arrival time may vary as much as 50% of the guard interval, we apply the FFT and preamble correlation for the samples up to 50% of the guard interval earlier than the corresponding detected useful time.

2.4.2.1 Frequency Domain Approach

In this section, we describe the UL synchronization stage II using the correlation of received signal with preamble in frequency domain. Figure 2.13 illustrates the processing

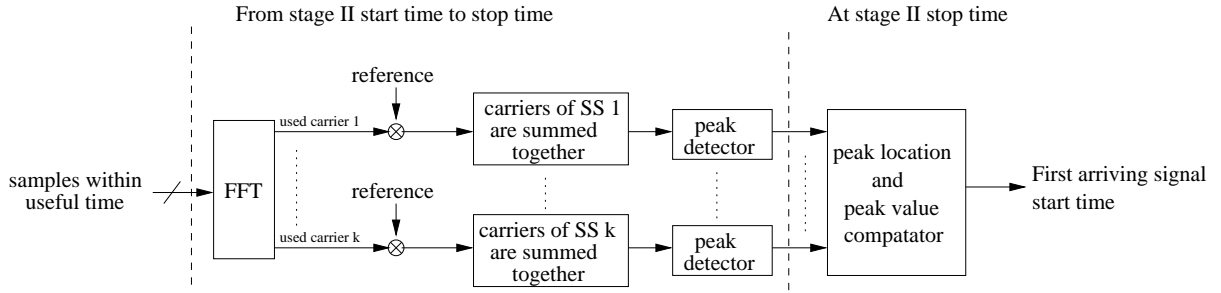


Figure 2.13: Illustration of UL synchronization stage II in frequency domain (from [5]).

conducted in stage II. The FFT outputs are correlated with the preamble reference values. As the BS knows the allocation status of UL subchannel, the frequency correlation is taken over all the subchannels used by each SS. When a new sample is received, the frequency is updated. The correlation peak value and location of each SS is recorded. This procedure is continued until the end of the corresponding useful time.

Then, the peak locations of different SSs are compared as follows. We start by assuming SS1 as the first coming signal. The peak location of SS2 is compared with that of SS1. If the peak location of SS2 is earlier than SS1, then we check the peak correlation value. The peak value is normalized by the number of subchannels each SS uses. If $(\text{peak_value}/\text{subchannel_num})$ of SS2 is larger than SS1, the first coming signal is set to SS2. After all SSs are compared, we get the start location of the first coming signal.

2.4.2.2 Time Domain Approach

In this section, we describe the UL synchronization stage II using the correlation of received signal with preamble in time domain.

Since the carriers are orthogonal to each other, so are the subchannels. After IFFT, the time domain signals which occupy different subchannels are uncorrelated if the channel has zero delay spread. For the UL preamble, the transmitted value of each carrier is specified by the BS. Thus the signal transmitted by each SS in the UL preamble is deterministic and the BS can produce the same signals as all SSs by taking IFFT. In this scheme, stage

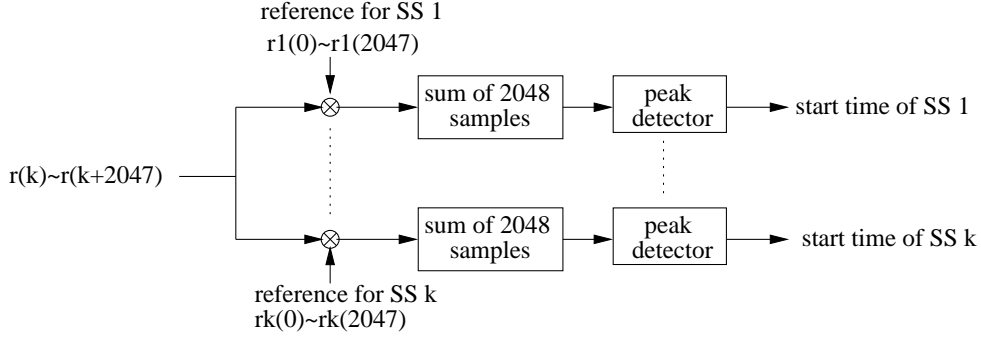


Figure 2.14: Illustration of UL synchronization stage II in time domain (from [5]).

I is the same as the previous scheme, and stage II is as shown in Figure 2.14.

The received samples are correlated with reference data string. Each reference data string is the IFFT output according to the subchannels used by each SS. When the next sample arrives, the correlation is calculated again. The start and stop times of the correlation are the same as shown in Figure 2.12.

The complexity of time domain correlation is less than frequency domain correlation. This is because we need to do FFT in frequency domain correlation. In order to reduce the complexity of FFT, the conventional FFT is only applied once. When a new data value is received, the simplified FFT below is used:

$$X_n(k) = [X_{n-1}(k) - x_{n-N} + x_n] e^{j \frac{2\pi k}{N}}, \quad (2.3)$$

where N is the FFT size, k is the carrier index, n is sample number, and x_n is the new incoming sample. The simplified FFT requires $2N$ complex additions and N complex multiplications. Table 2.3 shows a comparison of computational complexity for different FFT algorithm [9].

Table 2.4 shows a comparison of the complexity for time domain approach and frequency domain approach. For time domain correlation, only 2048 complex multiplications and 2047 complex additions are needed. In our simulation, the guard interval is 256 samples and hence stage II is applied to 128 sample locations. For frequency domain correlation, computation complexity depends on different type of FFT algorithm. After

Table 2.3: Comparisons of Computational Complexity for Different FFT Algorithms

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

Table 2.4: Complexity for Time Domain Approach and Frequency Domain Approach

Complexity	No. of Real Multiplications	No. of Real Additions
Time domain approach	1048576	1048320
Frequency domain approach		
Radix-2 + Simplified FFT	2115592	2108163
Radix-4 + Simplified FFT	2108163	2673155
Radix-8 + Simplified FFT	2106030	2671022
Split-radix-4/2 + Simplified FFT	2105348	2670340

calculation, the needed multiplications and additions of frequency domain correlation is about 2 times that of time domain correlation.

2.5 UL Synchronization Result

Table 2.5 specifies the transmission parameters for our simulation. The uplink and downlink use the same frequency bands. The intercarrier spacing is thus 5.58 kHz and the symbol length (without cyclic prefix) is 179.2 μsec .

In this section, we select the channel environment defined by ETSI for the evaluation of UMTS radio interface proposals. The time-varying channel impulse response for these models can be described by

$$h(\tau, t) = \sum_i \alpha_i(t) \delta(\tau - \tau_i). \quad (2.4)$$

This equation defines the channel impulse response at time t as a function of the lag τ . In this thesis, we will evaluate our synchronization algorithm for the choices of α_i and τ_i

Table 2.5: 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 μ sec (2304 samples)
Useful time (T_b)	179.2 μ sec (2048 samples)
Cyclic prefix time (T_g)	22.4 μ sec (256 samples)

Table 2.6: Characteristics of the ETSI “Vehicular A” Channel Environment

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

associated with the “Vehicular A” channel environment [10]. The channel taps $\alpha_i(t)$ are complex independent stochastic variables, fading with Jakes’ Doppler spectrum, with a maximum Doppler frequency of 240 Hz, reflecting a mobile speed of approximately 120 km/hr (and scatterers uniformly distributed around the mobile). The real-valued τ_i and the variance of the complex-valued α_i are given in [10] and repeated in Table 2.6.

The SNR is chosen to be 10 dB in the fading channels. Note that the receiver SNR specified in 802.16a is from 9.4 dB to 24.4 dB, so 10 dB, which is almost the worst condition, is a reasonable value for simulation. The maximum Doppler shifts of our simulation are shown in Table 2.7 for the speed from 0 km/hr to 100 km/hr.

The frame structure used in UL synchronization simulation is as shown in Figure 2.15. UL burst1 is transmitted by SS1 using 8 subchannels. UL burst2 is transmitted by SS2 using 16 subchannels. UL burst3 is transmitted by SS3 using 8 subchannels. The TTG and RTG each occupies 136 sample times. No ranging subchannel is provided.

Table 2.7: Relations Between Spread and Maximum Doppler Shift at Carrier Frequency 6GHz 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

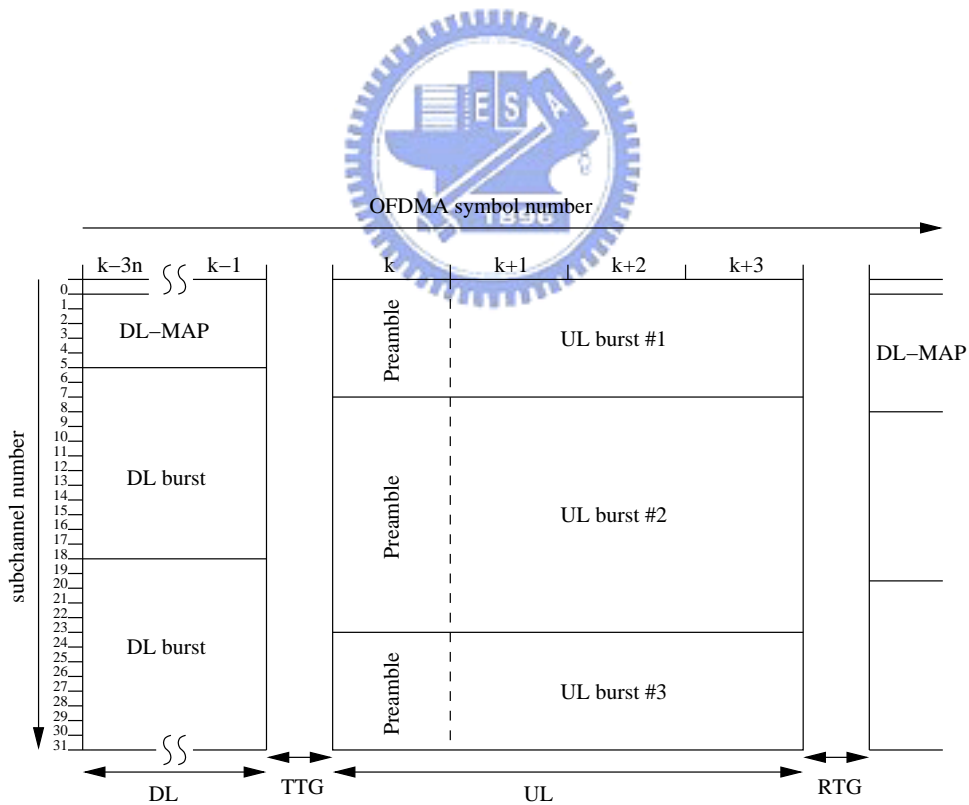


Figure 2.15: Frame structure used in UL synchronization.

Time synchronization errors under different Doppler shifts

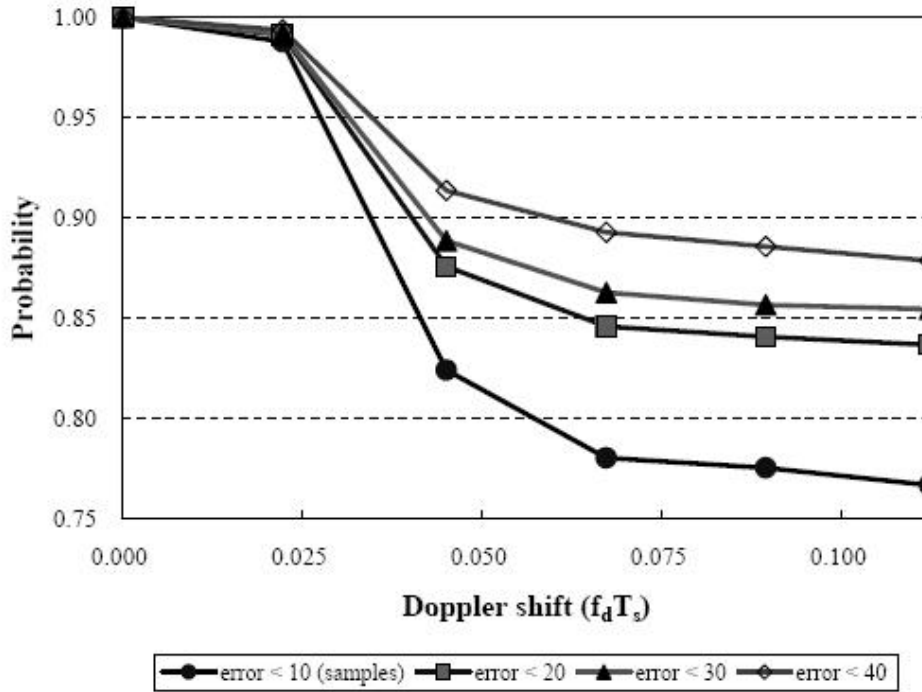


Figure 2.16: Error distribution under different maximum Doppler shifts using frequency domain approach.

The arriving times of burst1 and burst2 differ by 25% of the guard interval, which is 64 sample time, while burst3 lags burst1 by 50% of the guard interval, which is 128 sample times.

2.5.1 Preamble Correlation in Frequency Domain Approach

The probability of symbol time synchronization error for the first coming user is as shown in Figure 2.16.

The reason for using the carrier correlation to find the symbol start time is that if there is a time offset, the carrier phases will rotate. The phase rotation reduces the correlation. If there is no Doppler shift, the synchronization is always correct. For larger Doppler shifts, the inter-carrier interference causes serious variation of the post-FFT carrier values.

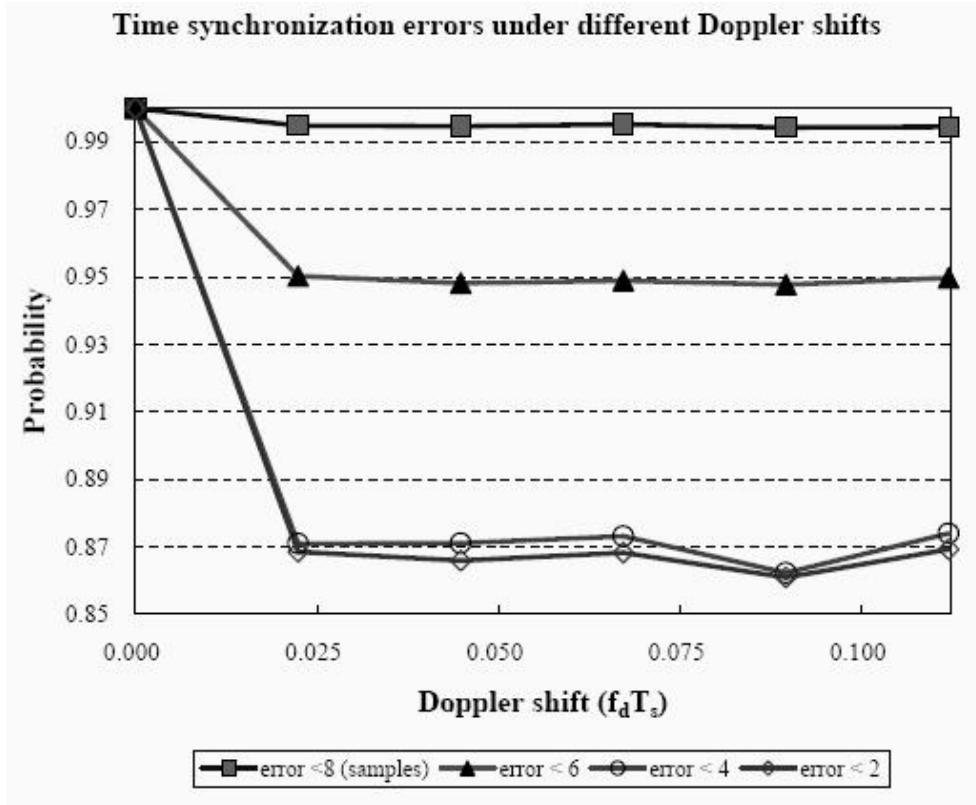


Figure 2.17: Error distribution under different maximum Doppler shifts using time domain approach.

Moreover, the signals passing through different fading channels of different SSs would affect each other. Thus the synchronization performance is decreased as the Doppler spread increases. We can see the performance drops significantly when the maximum Doppler shift is larger than $0.025 f_d T_s$.

2.5.2 Preamble Correlation in Time Domain Approach

Figure 2.17 shows the symbol time synchronization errors of the first coming signal under different Doppler spreads.

If the Doppler shift is zero (speed = 0 km/hr), we can always detect the correct symbol start time of the first coming signal. When the speed increases, the distribution of the time synchronization errors is closely related to the multipath channel. We have used this channel model to obtain the time synchronization error distribution shown in Figure 2.18.

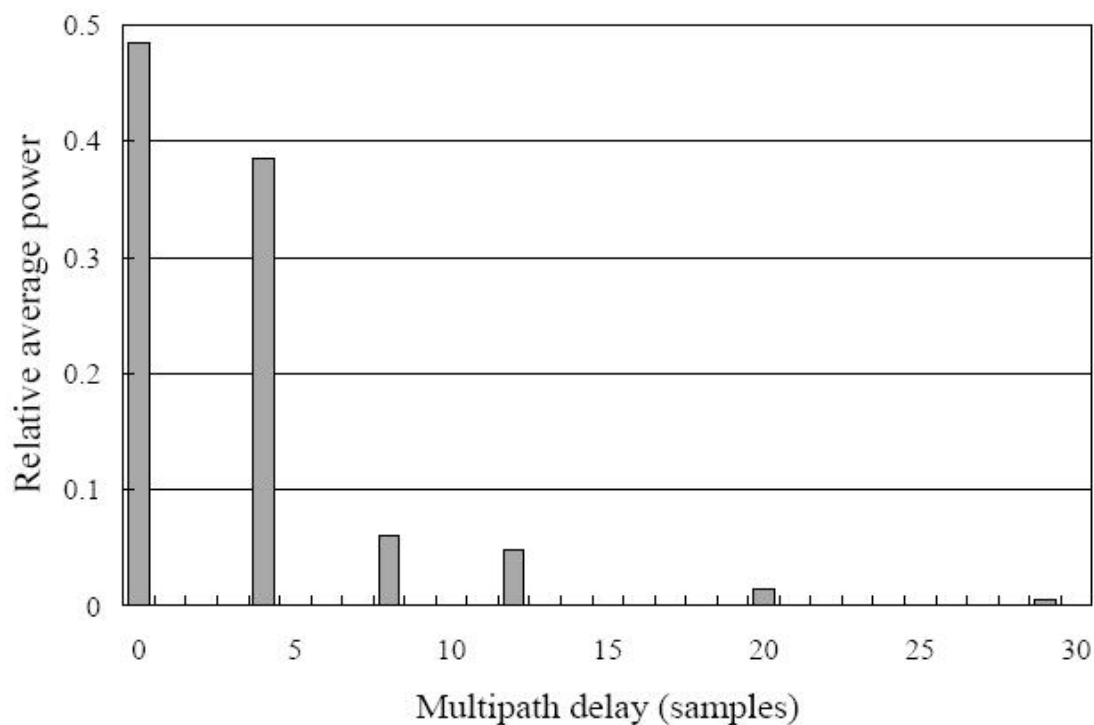


Figure 2.18: The multipath delay spread and the relative average power (The definition of the Ref in the next figure).



Comparing the time synchronization error distribution with the model, we see that the different time offsets obtained at synchronizer output almost concur with the sample number of the multipath delays. Furthermore, the occurrence probabilities at the different time offsets are proportional to the relative average power of the paths. The Doppler shift has no obvious effects on this synchronization scheme except when it is very small.

As the correlation is done for each SS, we can detect the arriving time of each later arriving signal. The time error distributions of the other SSs are similar to the previous condition. Thus, Figure 2.19 shows that correlation in time domain approach is ideal for fixed environments. For the mobile environments, the performance depends on how dispersive the multipath channel is. For different SSs, the errors under different Doppler shifts (excluding the zero shift) are averaged and the probabilities are shown in Figure 2.19.

Now that the estimated time offset is approximately equal to the multipath delay, we

**Time synchronization error distribution of UL synchronization
using time domain correlation approach**

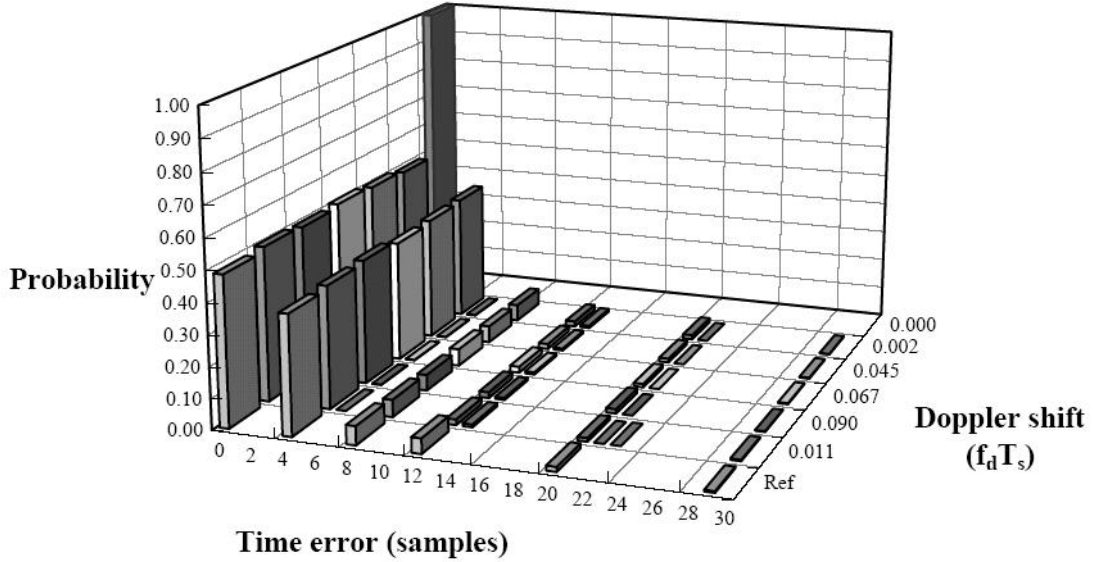


Figure 2.19: Performance of UL time synchronization under different Doppler spreads.

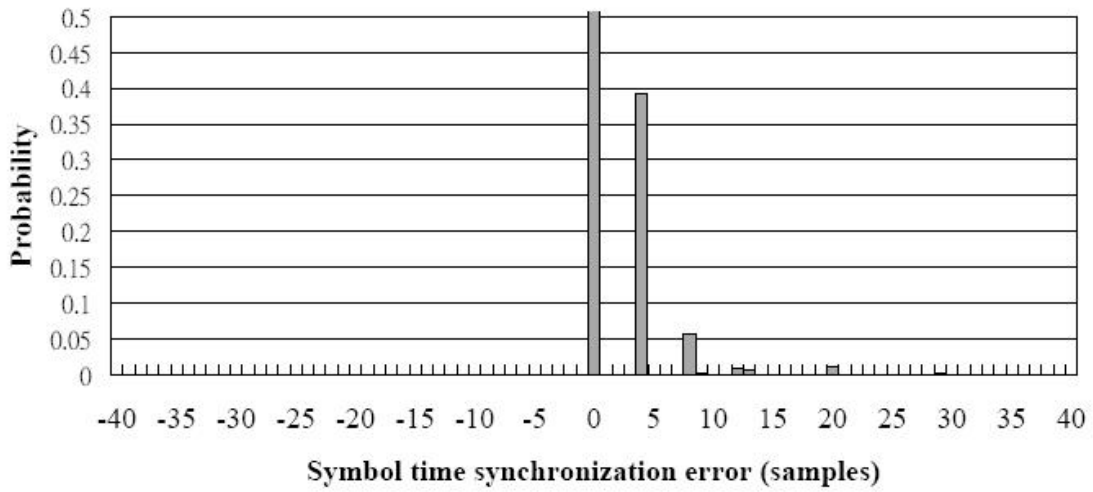
can safely say that, considering the guard interval, the minimum value of it should be larger than 2 times the channel delay spread plus the sacrificed part of guard interval due to pulse shaping. From Figure 2.17, 8 sample times earlier is reasonable for Doppler shift smaller than $0.1 f_d T_s$. In our simulation, this value is equal to

$$2 \times \underbrace{2.51}_{\text{maximum delay spread}} + \underbrace{0.7}_{\text{8 sample time}} = 5.72 \mu \text{sec} .$$

2.5.3 Comparison of UL Synchronization Using Time Domain Approach and Frequency Domain Approach

Figure 2.20 shows the time synchronization error distribution of UL synchronization using frequency domain and time domain approach when the maximum Doppler shift is 0.067 (velocity 60 km/hr).

Symbol time synchronization error for time domain approach
 (for speed = 60 km/hr, $f_d T_s = 0.067$)



Symbol time synchronization error for frequency domain approach
 approach
 (for speed = 60 km/hr, $f_d T_s = 0.067$)

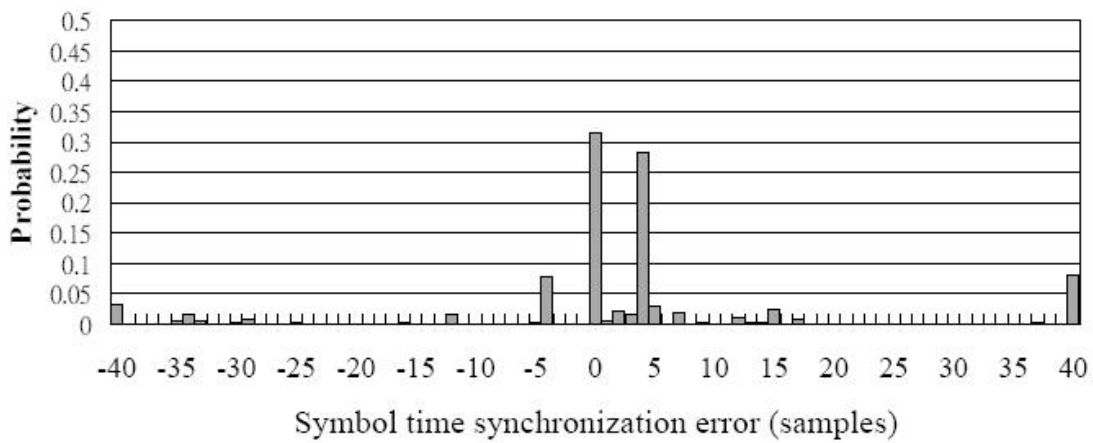


Figure 2.20: Comparison of UL synchronization using frequency domain and time domain approach at velocity of 60 km/hr.

Although the time offset estimated with correlation in frequency domain approach is to some degree related with the channel delay, it is more dispersive than correlation in time domain approach. The percentage of errors that are larger than 40 samples cannot be neglected. So the ability of the guard interval to counter the channel impulse response decreases. Moreover, the peak location of post-FFT correlation for the later signals cannot be used due to their low accuracy. This is because for larger Doppler shifts, the inter-carrier interference causes serious variation of the post-FFT carrier values. Comparing these two schemes, the correlation in time domain is more accurate and demands less complexity.



Chapter 3

DSP Introduction

In this thesis, we use digital signal processor (DSP) to implement the framing/deframing operation, the Tx/Rx SRRC filter, and the uplink synchronization scheme. The DSP board we use is Innovative Integration's Quixote, which is powered by the TMS320C6416 DSP from Texas Instruments (TI).

In this chapter, we focus on the environment of DSP implementation, which involves the host PC, the Quixote DSP board, and the C6416 DSP chip on the board. First, we introduce the DSP board and then the DSP core. The communication mechanism between the DSP core and the peripherals is also introduced. Last, we describe the code development on the TI DSP.

3.1 DSP Board Introduction [11]

Quixote is Innovative Integration's Velocia-family baseboard for wireless, RADAR, ultrasound, high energy physics and other demanding applications requiring speed and processing power. It combines a 600 MHz 32-bit fixed-point Texas Instruments C6416 DSP with two- or six- million-gate Xilinx Virtex-II FPGA. Figure 3.1 gives a block diagram of Quixote [12].

Quixote has a 32 MB SDRAM for use by the C6416 DSP. When used with the advanced cache controller on the C6416 DSP, the SDRAM provides a large, fast external memory pool for DSP data and code. The C6416 cache controller is said to be effective

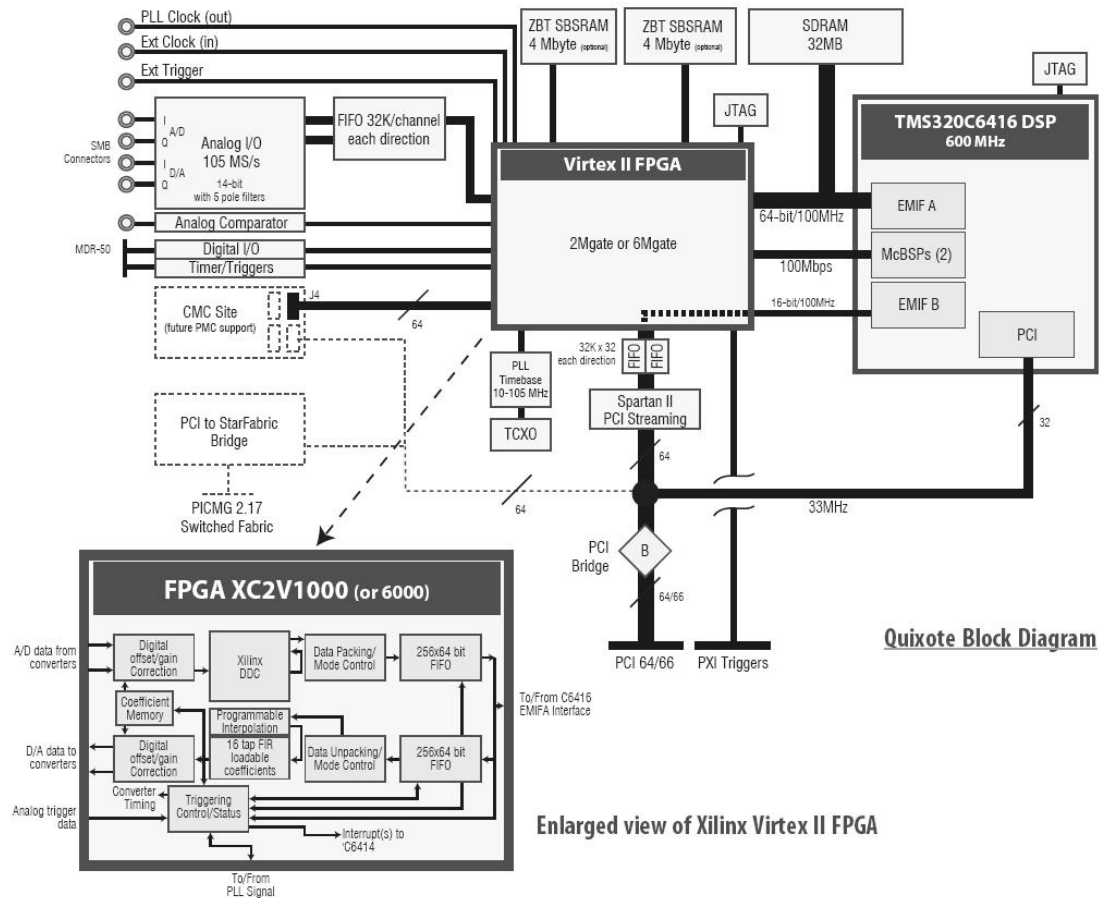


Figure 3.1: Block diagram of Quixote (from [12]).

to over 80% of on-chip memory performance for most DSP applications.

The analog interface offers 105 MHz 14-bit I/Q input channels and 105 MHz output channels, all tightly coupled to the FPGA external interface. A 64-bit 33 MHz PCI interface and one PMC site facilitate integration in PCI systems and support the addition of off-the-shelf and custom PMC mezzanine boards. Finally, a PCI-to-StarFabric bridge chip offers two full duplex 2.5 Gbps ports to the new PICMG 2.17 switched interconnect backplane, for up to 625 MBytes/sec board-to-board or chassis-to-chassis communication.

Figure 3.2 shows the technical specification of the Quixote [12]. In our work, we only focus on the C6416 DSP chip to implement OFDMA synchronization structure and

some related block. However, our goal is to implement the overall OFDMA system, including source coding, channel coding, framing/deframing, IFFT/FFT block, channel model, synchronization scheme and channel estimation, on several Quixote board. In the future work, we need to use the PCI-to-StarFabric bridge chip to do board-to-board communication.

3.2 DSP Core Introduction [13]

TMS320C6416T DSP core is the latest architecture of 32-bit fixed-point DSP generation in the C6000 DSP platform. It has 600 MHz clock rate and 4800 MIPS.

Table 3.1 provides an overview of the C6416 DSP. The table shows significant features of the C6416 devices, including the capacity of on-chip RAM, the peripherals, the CPU frequency, and the package type with pin count.

C6416 DSP uses a two-level cache-based architecture. The Level 1 program cache (L1P) is a 16K-Byte direct mapped cache and the Level 1 data cache (L1D) is a 16K-Byte 2-way set-associative cache. The Level 2 memory/cache (L2) consists of an 1024K-Byte memory space that is shared between program and data space. L2 memory can be configured as mapped memory or combinations of cache and mapped memory.

C6416 DSP chip also has two high-performance embedded coprocessors, which are Viterbi Decoder Coprocessor (VCP) and Turbo Decoder Coprocessor (TCP). The two coprocessors are very useful for channel decoding. Communications between the VCP/TCP and the CPU are carried through the EDMA controller. The enhanced direct memory access (EDMA) controller transfers data between the memory without passing through the DSP core.

The external memory interface (EMIF) provides the interface for the DSP core to connect with several external devices, allowing additional data and program space. C6416 DSP has two EMIFs: the 64-bit EMIF A is interfaced to the SDRAM and the Virtex-II FPGA while the 16-bit EMIF B is primarily used for the streaming PCI interface.

Digital Signal Processor Texas Instruments TMS320C6416 720MHz (1GHz when available) 32-bit fixed-point DSP 16 KB data L1 cache 16 KB program L1 cache 1Mbyte L2 cache (3) 32-bit timers 64 EDMA channels	With Prescaler and M/N type VCO Frequency Range 25-105MHz External clock input on SMB connector PXI compatible Synchronize with common triggers or clocks 5 PXI Triggers, 1 Star Trigger	External Clock - SMB connector External Interrupt - SMB connector Analog Comparator Input - SMB connector 32-bit DIO, Timers, Start/Stop Triggers - MDR50 connector DSP JTAG - 14-pin 0.1" shrouded male header Virtex-II JTAG - 14-pin 2mm male header
FPGA Xilinx Virtex-II XC2V2000 or XC2V6000	A/D Channels (2) 14-bit Analog Devices AD6645 converters Sampling Rate 30-105MHz DC-coupled input Input Range ± 2 V, single ended, 50 Ohm impedance Input Filter 5 pole, analog low pass anti-alias filter Standard config -3dB rolloff at 33MHz Digital gain/offset correction in logic	Physical Size 6U CompactPCI card
Interface DSP/Virtex EMIF A, 64 bit/100MHz interface Mirrored EMIFB to PCI interface (2) Sync serial ports (McBSP) at 100Mbit/sec. each	D/A Channels (2) 14-bit Analog Devices AD9764 converters Sampling Rate DC to 105MHz Output Range ± 2 V Single ended, 50 ohm impedance Output Filter 5 pole analog smoothing filter -3dB rolloff at 33MHz Digital gain/offset correction in logic Custom filters may be added	Cooling On-card fan for the FPGA with integrated heat sink
Memory 32 Mbytes of SDRAM; 8 Mbytes zero bus turnaround SBRAM (Quixote II or III only) 32 MB flash EEPROM for FPGA configuration bitstream 512 Byte serial EEPROM for converter calibration coeff.	Triggering Modes Continuous, single-shot, re-triggered modes Software selectable start/stop triggers include software driven, external stop/start, number of points	Power Requirements $+5$ V 1.9A; 3.3V 1.1A; ± 12 V 0.07A; Total < 15W
CompactPCI bus 64/32 bit, 3.3/5V, 66MHz, Local bus 33MHz Capable of 264/132Mbytes/sec respectively Controller auto-detects host bus type Busmaster or slave operation (16) 32-bit bi-directional mailboxes FIFO interface to DSP-100 MHz, 16-bit, EMIF B	Connectors PICMG 2.17 compliant cPCI interface (avail. Q2 2004) IEEE 1386 compliant PMC site, Jn1-Jn4 connectors A/D and D/A - SMB connectors (4)	Development Languages DSP C or assembler using Code Composer Studio and Pismo Toolset FPGA VHDL using Xilinx ISE and ModelSim Host Borland C++ Builder or Microsoft VC++
Digital I/O 40 bits I/O, 3.3V 64bit I/O via PMC site Jn4 connector	DSP Operating system DSP/BIOS II	Support PC OS Windows2000/XP
Timebase PLL timebase using 5ppm 14.4MHz TCXO oscillator	Host PC Intel processor recommended for max speed in applications using Channelized Mode and Analysis Components, which utilize MMX technology	

Software Selection Guide for Quixote

Software Package	Description	Usage/Requirements	Page	Recommendations
Pismo Toolset	Peripheral libraries needed for developing code on this card. Includes host applications, target examples in source form demonstrating use of peripherals on the card, DSP-BIOS peripheral device driver.	Requires CCStudio*. Windows2000/XP compatible.	98	Includes Caliente DLL and Armada.
Quixote VHDL	Source code of the Quixote Virtex II Framework controlling all interfaces to the FPGA and some signal processing.	Requires Xilinx ISE tools for further integration with custom firmware. ModelSim highly recommended.	128	Recommended for experienced FPGA design developers who use Quixote for custom firmware development/implementation.
Caliente DLL	Dynamic link library (DLL) for the Quixote.	Requires ANSI-compliant C/C++ compiler. For example, Microsoft Visual C/C++ or Borland C++ Builder.		Required for interfacing Host side code to DSP.
CCStudio *C6000	Integrated development environment (IDE) for Target side development/debugging from Texas Instruments.	Requires XDS-510 compatible JTAG emulator for debugging capabilities.	91	Required for all first time users. Recommend use with Innovative Integration plug-n-play PCI JTAG emulator.
Armada	Host side development package using a revolutionary integrated development environment (IDE). Allows user to build/debug sophisticated data acq apps fully using MS Windows graphical environment quickly with Innovative Integration's Visual Component Libraries (VCL) of MFC Classes.	Requires Borland C++ Builder* or Microsoft Visual C++.	103	Offers easiest interface while providing the most flexibility and performance. Ties into a plethora of 3rd party components.

The Quixote Development Package contains all software packages listed above.

*Contact Innovative Integration for current release version.

Figure 3.2: Technical specification of Quixote (from [12]).

Table 3.1: Characteristics of TI C6416T Processors (from [14])

HARDWARE FEATURES		C6414T, C6415T, and C6416T
Peripherals Not all peripherals pins are available at the same time. (For more details, see the Device Configuration section.) Peripheral performance is dependent on chip-level configuration.	EMIFA (64-bit bus width) (default clock source = AECLKIN)	1
	EMIFB (16-bit bus width) (default clock source = BECLKIN)	1
	EDMA (64 independent channels)	1
	HPI (32- or 16-bit user selectable)	1 (HPI16 or HPI32)
	PCI (32-bit)	1 [C6415T/C6416T only]
	McBSPs (default internal clock source = CPU/4 clock frequency)	3
	UTOPIA (8-bit mode)	1 [C6415T/C6416T only]
	32-Bit Timers (default internal clock source = CPU/8 clock frequency)	3
	General-Purpose Input/Output 0 (GP0)	16
Decoder Coprocessors	VCP	1 [C6416T only]
	TCP	1 [C6416T only]
On-Chip Memory	Size (Bytes)	1056K
	Organization	16K-Byte (16KB) L1 Program (L1P) Cache 16KB L1 Data (L1D) Cache 1024KB Unified Mapped RAM/Cache (L2)
CPU ID + CPU Rev ID	Control Status Register (CSR.[31:16])	0x0C01
Device_ID	Silicon Revision Identification Register (DEVICE_REV [20:16]) Address: 0x01B0 0200	DEVICE_REV[20:16] Silicon Revision 10000 1.0 (14T/15T/16T)
Frequency	MHz	600, 720, 1000 (1-GHz)
Cycle Time	ns	1.67 ns (C6414T/15T/16T - 6 [600 MHz]) 1.39 ns (C6414T/15T/16T - 7 [720 MHz]) 1 ns (C6414T/15T/16T - 1 [1 GHz])
Voltage	Core (V)	1.1 V (-600) 1.2 V (-720, -1 G)
	I/O (V)	3.3 V
PLL Options	CLKIN frequency multiplier	Bypass (x1), x6, x12, x20
BGA Package	23 x 23 mm	532-Pin BGA (GLZ)
Process Technology	µm	0.09 µm
Product Status	Product Preview (PP) Advance Information (AI) Production Data (PD)	PP

Figure 3.3 shows the block diagram of the C6416 DSP chip.

The DSP core features two sets of functional units. Each set contains four units and a register file. One set contains functional units .L1, .S1, .M1, and .D1; the other set contains units .D2, .M2, .S2, and .L2. The two register files each contain 32 32-bit registers for a total of 64 general-purpose registers.

In addition to support the packed 16-bit and 32-/40-bit fixed-point data types found in the C62x VelociTI VLIW architecture, the C64x register files also support packed 8-bit data and 64-bit fixed-point data types. The two sets of functional units, along with two register files, compose sides A and B of the DSP core. The four functional units on each side of the CPU can freely share the 32 registers belonging to that side.

Additionally, each side features a “data cross path” — a single data bus connected to all the registers on the other side, by which the two sets of functional units can access data from the register files on the opposite side. The C6416 DSP core pipelines data-cross-path accesses over multiple clock cycles. This allows the same register to be used as a data-cross-path operand by multiple functional units in the same execute packet.

All functional units in the C6416 CPU can access operands via the data cross path. Register access by functional units on the same side of the DSP core as the register file can service all the units in a single clock cycle. Figure 3.4 shows the data path of the C6416 DSP chip.

On the DSP core, a delay clock is introduced whenever an instruction attempts to read a register via a data cross path if that register was updated in the previous clock cycle.

Another key feature of the C6416 DSP core is the load/store architecture, where all instructions operate on registers. The function units .L and .S are described in Table 3.2.

The two .S and .L functional units perform a general set of arithmetic, logical, and branch functions with results available every clock cycle. The arithmetic and logical functions on the C64x CPU include single 32-bit, dual 16-bit, and quad 8-bit operations.

Two sets of data-addressing units (.D1 and .D2) are responsible for all data transfers

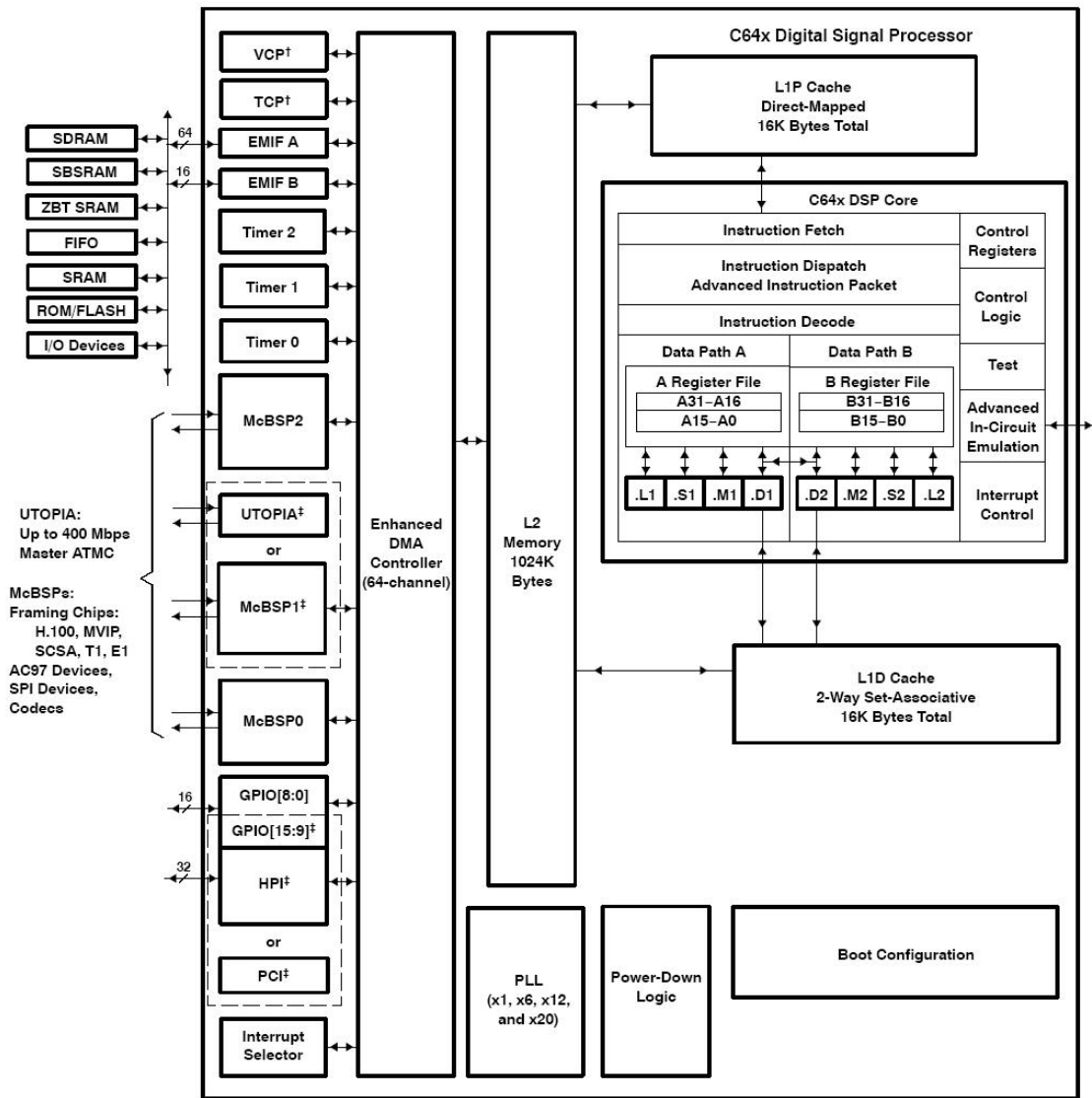
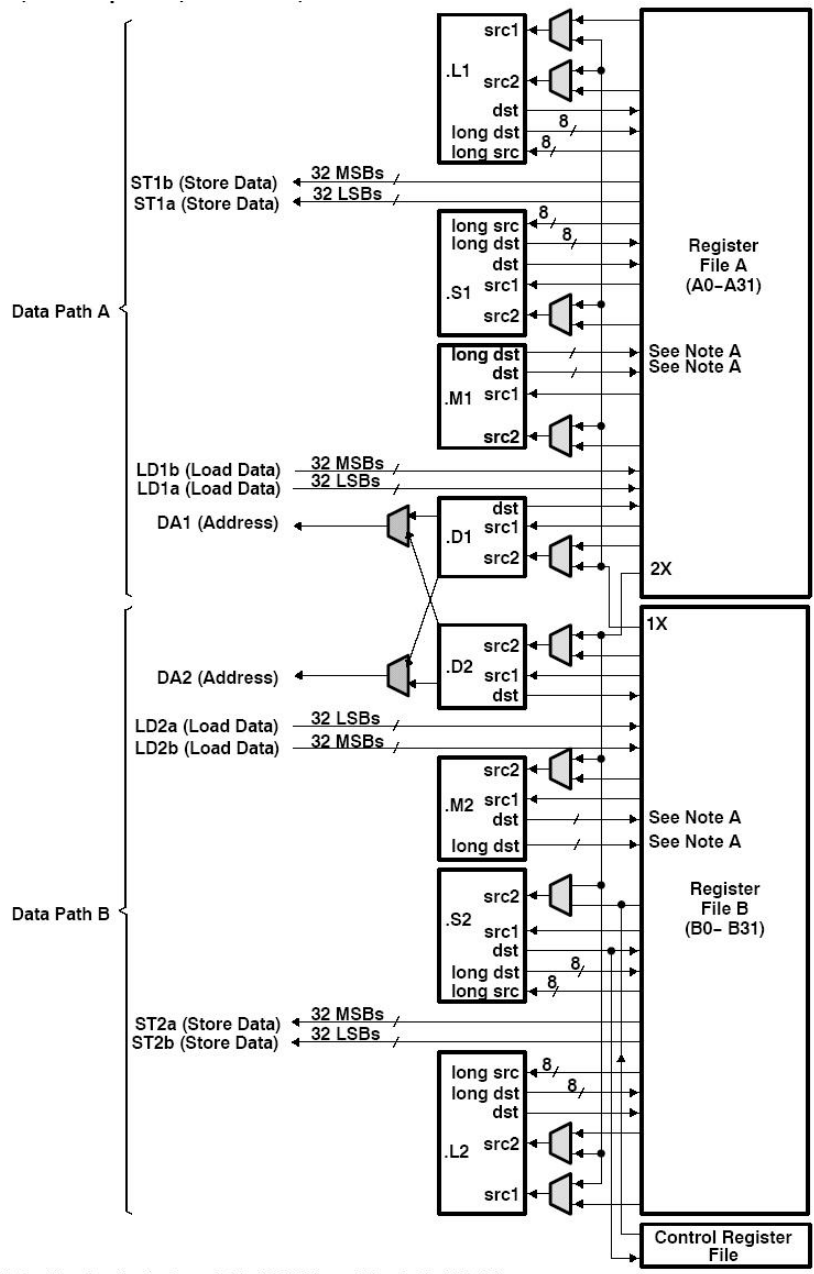


Figure 3.3: Block diagram for C6416 DSP (from [14]).



NOTE A: For the .M functional units, the long dst is 32 MSBs and the dst is 32 LSBs.

Figure 3.4: TMS320C6416 DSP core data paths (from [14]).

Table 3.2: Functional Units (.L, .S) and Operations Performed (from [15])

Function Unit	Fixed-Point Operations
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits Byte shifts Data packing/unpacking 5-bit constant generation Dual 16-bit arithmetic operations Quad 8-bit arithmetic operations Dual 16-bit min/max operations Quad 8-bit min/max operations Quad 8-bit subtract with absolute value
.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

between the register files and the memory. The data address driven by the .D units allows data addresses generated from one register file to be used to load or store data to or from the other register file. The C6416 .D units can load and store bytes (8 bits), half-words (16 bits), and words (32 bits) with a single instruction. And with the new data path extensions, the C6416 .D unit can load and store doublewords (64 bits) with a single instruction. Furthermore, the non-aligned load and store instructions allow the .D units to access words and doublewords on any byte boundary. The C6416 DSP core supports a variety of indirect addressing modes using either linear- or circular-addressing with 5- or 15-bit offsets. All instructions are conditional, and most can access any one of the 64 registers. Some registers, however, are singled out to support specific addressing modes or to hold the condition for conditional instructions (if the condition is not automatically true).

The two .M functional units perform all multiplication operations. Each of the C64x .M units can perform two 16x16-bit multiplies or four 8x8-bit multiplies per clock cycle. The .M unit can also perform 16 32-bit multiply operations, dual 16 16-bit multiplies with add/subtract operations, and quad 8 8-bit multiplies with add operations. In addition to standard multiplies, the C64x .M units include bit-count, rotate, Galois field multiplies, and bidirectional variable shift hardware. The function units .M and .D are described in Table 3.3.

The processing flow begins when a 256-bit-wide instruction fetch packet is fetched from a program memory. The 32-bit instructions destined for the individual functional units are “linked” together by “1” bits in the least significant bit (LSB) position of the instructions. The instructions that are “chained” together for simultaneous execution (up to eight in total) compose an execute packet. A 0 in the LSB of an instruction breaks the chain, effectively placing the instructions that follow it in the next execute packet. A C6416 DSP device enhancement now allows execute packets to cross fetch-packet boundaries. In the TMS320C62x/TMS320C67x DSP devices, if an execute packet crosses the

Table 3.3: Functional Units (.M, .D) and Operations Performed (from [15])

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

fetch-packet boundary (256 bits wide), the assembler places it in the next fetch packet, while the remainder of the current fetch packet is padded with NOP instructions.

In the C64x DSP device, the execute boundary restrictions have been removed, thereby, eliminating all of the NOPs added to pad the fetch packet, and thus, decreasing the overall code size. The number of execute packets within a fetch packet can vary from one to eight. Execute packets are dispatched to their respective functional units at the rate of one per clock cycle and the next 256-bit fetch packet is not fetched until all the execute packets from the current fetch packet have been dispatched. After decoding, the instructions simultaneously drive all active functional units for a maximum execution rate of eight instructions every clock cycle. While most results are stored in 32-bit registers, they can be subsequently moved to memory as bytes, half-words, words, or doublewords. All load and store instructions are byte-, half-word-, word-, or doubleword-addressable.

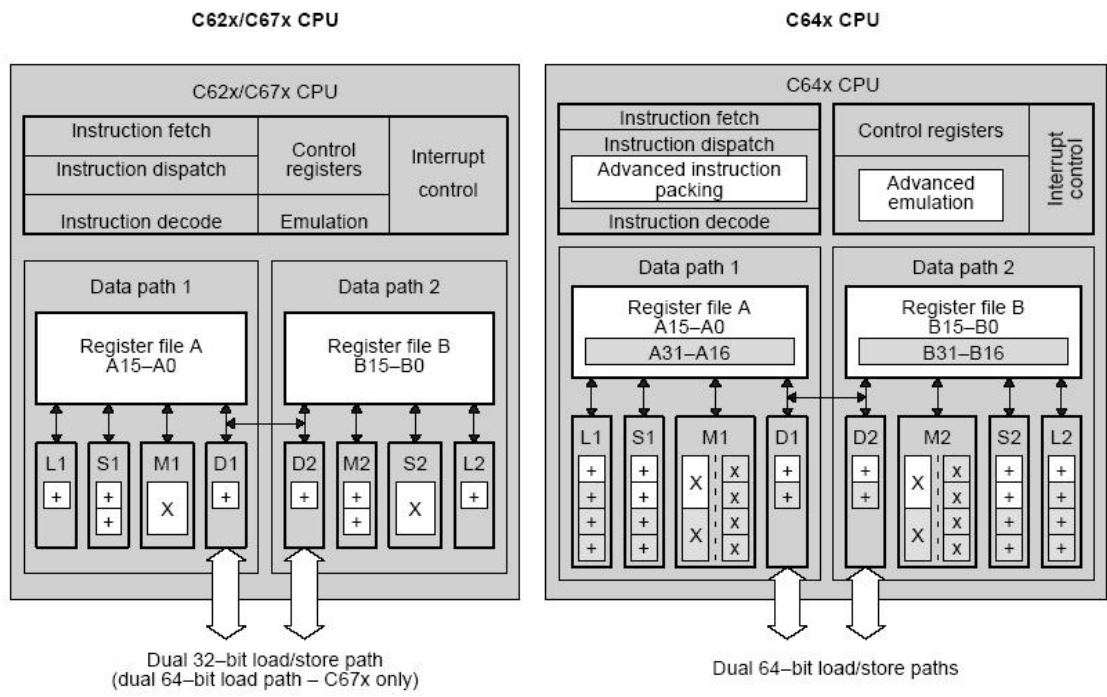


Figure 3.5: Block diagram for C62x and C64x DSP core (from [15]).

Figure 3.5 compares the difference between the C62x DSP core and the C64x DSP core. By doubling the registers in the register file and doubling the width of the data path as well as utilizing advanced instruction packing, the C6000 compiler can improve performance with even fewer restrictions placed upon it by the architecture. These additions and others make the C64x an even better compiler target than the original C62x architecture, while reducing code size by up to 25%.

3.3 Data Transmission Mechanism [15]

Many applications of the Matador family baseboards involve communication with the host CPU in some manner. All applications at a minimum must be reset and downloaded from the host, even if they are isolated from the host after that. Other applications need to interact with a host program during the lifetime of the program. This may vary from a small amount of information to acquiring large amounts of data.

Some examples:

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

These different requirements require different levels of support to efficiently accomplish. The simplest method supported is a mapping of Standard C++ I/O to the Uniterminal applet that allows console-type I/O on the host. This allows simple data input and control and the sending of text strings to the user. The next level of support is given by the Packetized Message Interface. This allows more complicated medium rate transfer of commands and information between the host and target. It requires more software support on the host than the Standard I/O does. For full rate data transfers the hardware supports the creation of data streaming to the host, for the maximum ability to move data between the target and host. On Quixote baseboard, a second type of busmaster communication between target and host is available for use, the CPU Busmaster interface.

The primary CPU busmaster interface is based on the streaming model, where logically data is an stream between the source and destination. The model is more efficient because the signaling between the two parties in the transfer can be kept to a minimum and transfers can be buffered for maximum throughput. In addition, the Busmaster streaming interface is fully handshake, so that no data loss can occur in the process of streaming.

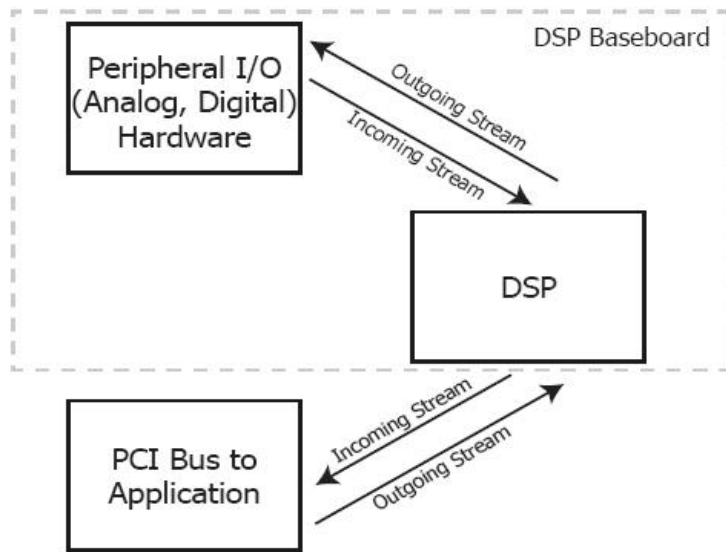


Figure 3.6: Block diagram of DSP streaming mode (from [11]).

For example, if the application cannot process blocks fast enough, the buffers will fill, then the busmaster region will fill, then busmastering will stop until the application resumes processing. When the busmaster stops, the DSP will no longer be able to add data to the PCI interface FIFO.

The DSP Streaming interface is bi-directional. Two streams can run simultaneously, one running from the analog peripherals through the DSP into the application. This is called the “Incoming Stream”. The other stream runs out to the analog peripherals. This is the “Outgoing Stream”. In both cases, the DSP needs to act as a mediator, since there is no direct access to analog peripherals from the host. Figure 3.6 shows the block diagram of the DSP streaming mode.

DSP Streaming is initiated and started on the Host, using the Caliente component. On the target, the DSP interface uses a pair of DSP/BIOS Device Drivers, PciIn (on the Outgoing Stream) and PciOut (on the Incoming Stream), provided in the Pismo peripheral libraries for the DSP. They use burst-mode and are capable of copying blocks of data between target SDRAM and host bus-master memory via the PCI interface at instantaneous rates up to 264 MB/sec.

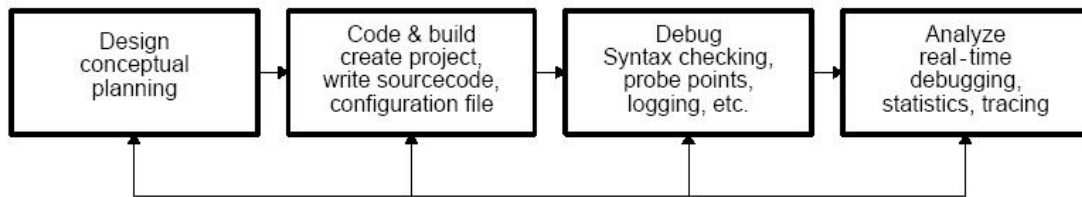


Figure 3.7: Simplified code composer studio development flow (from [17]).

In addition to the busmaster streaming interface, the DSP and the host also have a lower bandwidth communication link called packetized message interface for sending commands or side information between the host PC and the target DSP.

3.4 Code Composer Studio Introduction [16], [17]

TI's Code Composer Studio (CCS) is a useful GUI tool to develop DSP codes. The CCS contains simple components: concept/design, code/build, debug, analyze, and extends the basic code generation tools with a set of debugging and real-time analysis capabilities. The phases of the development cycle are shown in Figure 3.7.

We briefly describe some of its features related our implementation. The details can be found in [16] and [17].

1. Compiles your C code to generate the Common Object File Format (COFF) output file.
2. Choose Run, Halt, Animate, or Run Free to start or stop to execution your program.
3. When the DSP halts, check the memory sections.
4. Probes the PC file stream into or from the target memory locations.
5. Counts the instruction cycles from the profile.

We can divide the software development into three steps.

Step 1: Write the C program like standard ANSI C code. Then use the debugger to profile the C code to identify the inefficient areas in the code.

Step 2: Use the optimization techniques and intrinsic function to improve the performance. Refine the C code procedures such as data type modifiers, compiler options, intrinsics, and so on.

Step 3: Find the most time-critical areas and use the linear assembly code to replace the C code. We can use the assembly optimizer to optimize the code.

In our work, we only focus on step 1 and step 2. Details for the optimization methods are shown in the next chapter.



Chapter 4

DSP Implementation

In the earlier chapters, the backgrounds of uplink synchronization scheme and its related function are given. We also described the environment of the DSP implementation. In this chapter, we discuss the DSP implementation of uplink synchronization and its related work on C6416 DSP. First, we describe the procedure of our implementation work. Second, we illustrate some optimization methods using the features of C6416 and applied to our implementation. Third, we discuss the progress in each part of our system with different methods. Because the compiler changes the C program into assembly code, we can see the parallel situation from the assembly code. The profile is for comparison between the original floating-point code and the optimized fixed-point code. Finally at the end of this chapter, we present some experimental results on the speed and the synchronization performance of our implementation.

4.1 Procedure of the Implementation Work

Traditional development flows in the DSP industry have involved validating a C model for correctness on a host PC or UNIX workstation and then painstakingly porting that C code to hand coded DSP assembly language. 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. Figure 4.1 shows the phases in the 3-step software development flow.

4.2 Optimization Method

Speeding up the execution time of the OFDMA framing/deframing structure, the SRRC filter, and the uplink synchronization scheme is the main task of our implementation. In this section, we introduce the supported by the special features of C64x DSP. The experimental results are discussed in the next section.

4.2.1 Configuring the Setting of Compiler Options

As we mentioned in section 3.4, the Code Composer Studio (CCS) is a useful GUI tool for us to develop DSP codes. CCS compiles the C code and assembles it into the Common Object File Format (COFF) file format. Compiler options control the operation of both the compiler and the programs it runs. Proper configuration of the compiler options helps the compiler to generate efficient assembly codes. The compiler tools include a shell program (c16x), which you use to compile, assembly optimize, assemble, and link program in a single step. The options described in Table 4.1 are obsolete or intended for debugging, and could potentially decrease performance and increase code size. Avoid using these options with performance critical code.

The options in Table 4.2 can improve performance but require certain characteristics to be true.

Details for total compiler options can be found in [18]. The compiler option we usually use is `-o3`, which represents the highest level of optimization available. In addition to the optimization described in Table 4.2, `-o3` can perform other code size reducing optimization like: eliminating unused assignments, eliminating local and global common subunused assignments, and removing functions that are never called.

In addition, we can specify program-level optimization by using the `-pm` option with the `-o3` option. With program-level optimization, all of the source files are compiled

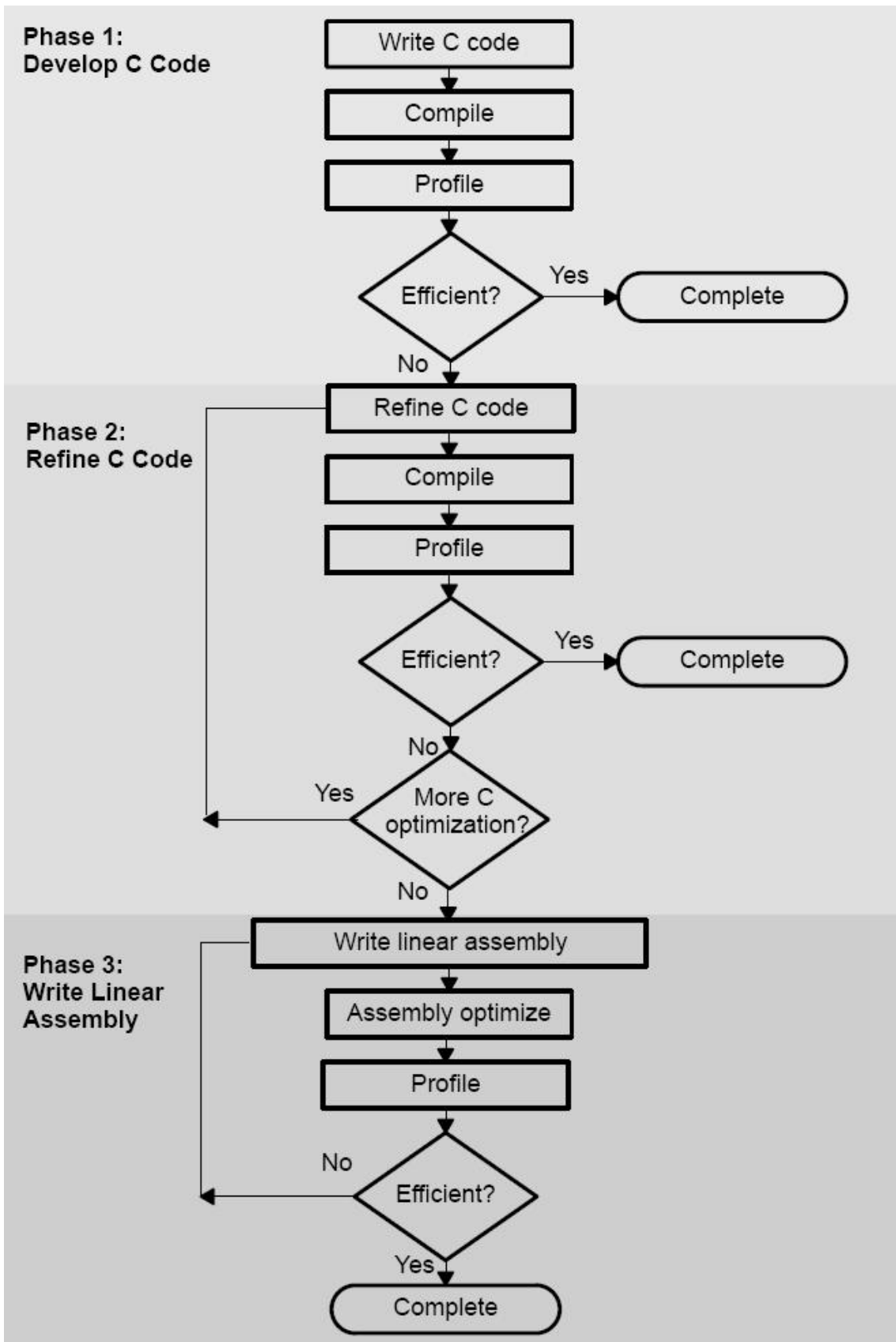


Figure 4.1: Code development flow of C6000 (from [19]).

Table 4.1: Compiler Options to Avoid on Performance Critical Code (from [19])

Option	Description
<code>-g/-s/-ss/-mg</code>	These options limit the amount of optimization across C statements leading to larger code size and slower execution.
<code>-mu</code>	Disables software pipelining for debugging. Use <code>-ms2/-ms3</code> instead to reduce code size which will disable software pipelining among other code size optimizations.
<code>-o1/-o0</code>	Always use <code>-o2/-o3</code> to maximize compiler analysis and optimization. Use code size flags (<code>-msn</code>) to tradeoff between performance and code size.
<code>-mz</code>	Obsolete. On pre-3.00 tools, this option may have improved your code, but with 3.00+ compilers, this option will decrease performance and increase code size.

into one intermediate file giving the compiler complete program view during compilation. This creates significant advantage for determining pointer locations passed into a function. Once the compiler determines two pointers do not access the same memory location, substantial improvements can be made in software pipelined loops. Because the compiler has access to the entire program, it performs several additional optimizations 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.

Also, using the `-pm` option can lead to better schedules for our loops. If the number of iterations of a loop is determined by a value passed into the function, and the compiler can

determine what that value is from the caller, then the compiler will have more information about the minimum trip count of the loop leading to a better resulting schedule.

4.2.2 Using Intrinsic [19]

The C6000 compiler provides intrinsics, special functions that map directly to C64x instructions, to optimize our C code quickly. All instructions that are not easily expressed in C code are supported as intrinsics. Intrinsics are specified with a leading underscore (-) and are accessed by calling them as we call a function. The table of TMS320C6000 C/C++ compiler intrinsics can be found in [19].

4.2.3 Software Pipelining

Pipeline is used to parallelize instruction execution. The C64x pipeline has several features that improve performance. Figure 4.2 shows all the phases in each stage of the C64x pipeline in sequential order, from left to right [13]. As shown in Figure 4.2, the C64x has 11 phases, and the phases are grouped into 3 pipeline stages: program fetch, instruction decode and execution. In the execution stage, most of the C64x instructions are done in one phase. However, the load instruction needs five execution phases, the store instruction needs three execution phases, the multiplication needs four execution phases, and the branch needs six execution phases. If the sequential instructions need the result of these kinds of multi-cycle instructions, there is a delay before the result is written to the register file and available. Thus the NOP instruction is added to the program by the compiler to represent one cycle delay. So 4, 2, 3, 5 NOPs are added following the load, store, multiplication and branch instructions respectively.

Software pipelining is a technique which can be used to schedule instruction from a loop so that multiple iterations of the loop execution in parallel. It is a great way to improve performance. The concept of software pipelining consists of implementing parallel instructions, filling delay slots with useful instructions, loop unrolling and maximizing functional units usage. When we use the -o2 or -o3 compiler options, the compiler at-

Table 4.2: Compiler Options for Performance (from [19])

Option	Description
-mh<n>§ -mhh	Allows speculative execution. The appropriate amount of padding must be available in data memory to insure correct execution. This is normally not a problem but must be adhered to.
-mi<n>§ -mii	Describes the interrupt threshold to the compiler. If you know that NO interrupts will occur in your code, the compiler can avoid enabling and disabling interrupts before and after software pipelined loops for a code size and performance improvement. In addition, there is potential for performance improvement where interrupt registers may be utilized in high register pressure loops. (See the <i>TMS320C6000 Programmer's Guide</i> (SPRU198))
-mt§	Enables the compiler to use assumptions that allow it to be more aggressive with certain optimizations. When used on linear assembly files, it acts like a <code>.no_mdep</code> directive that has been defined for those linear assembly files. (See the <i>TMS320C6000 Programmer's Guide</i> (SPRU198))
-o3†	Represents the highest level of optimization available. Various loop optimizations are performed, such as software pipelining, unrolling, and SIMD. Various file level characteristics are also used to improve performance.
-op2§	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler. This improves variable analysis and allowed assumptions.
-pm‡	Combines source files to perform program-level optimization.

† Although -o3 is preferable, at a minimum use the -o option.

‡ Use the -pm option for as much of your program as possible.

§ These options imply assertions about your application.

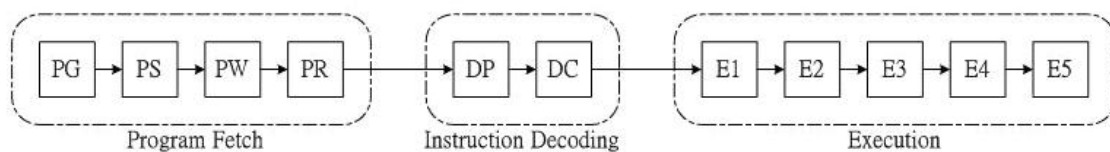


Figure 4.2: C64x fixed-point pipeline phases.

tempts to software pipeline code with the information that it gathers from the program. If the compiler can gather the more information from the program, the result schedule can be better. We may help the optimization work if the compiler by providing some information to the compiler as described below.

Loop unrolling

Loop unrolling expands small loops so that all iterations of the loop appear. It can increase the number of instructions available to execute in parallel. The compiler may automatically unroll the loop or be suggested using the *UNROLL* pragma. The syntax of the *UNROLL* pragma is:

$$\#pragma\ UNROLL(n)$$

If possible, the compiler unrolls the loop so there are n copies of the original loop. But under the conditions listed below, the compiler will not do software pipelining [19]:

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

4.2.4 Data Type Modification

The TMS320C6416 is a fixed-point DSP, so floating-point operations on C6416 DSP are inefficient. This is the main reason we rewrite the original floating-point C code to fixed-point version. We should use the 16-bit data type for multiplication inputs whenever

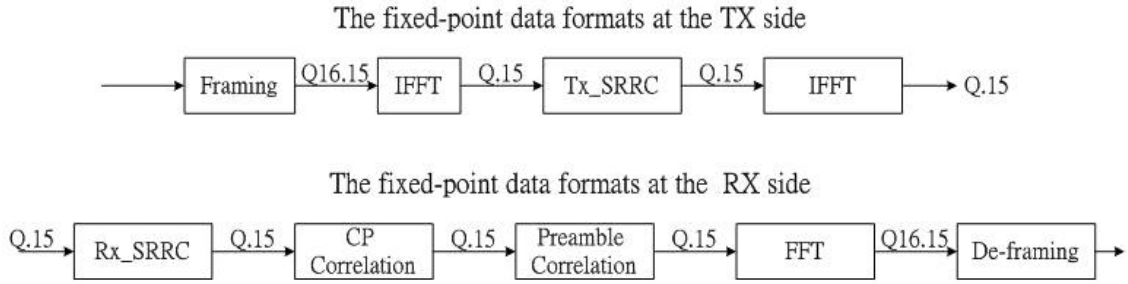


Figure 4.3: The fixed-point data formats at the TX side and the RX side.

possible because this data type can provide the most efficient use of the 16-bit multiplier in C64x DSP. Figure 4.3 shows the data formats at the TX side and the RX side.

In the original floating-point version, Tx_SRRC, Rx_SRRC and SYNC function need lots of 32-bit by 32-bit floating-point multiply operations. In fixed-point version, we use 16-bit by 16-bit fixed-point multiply operations to instead. In UL, the ranges of data values before IFFT and after FFT are $[1, -1]$. Also, the data values after IFFT and before FFT are less than 1. Therefore, we set the input/output data formats for Tx_SRRC, Rx_SRRC and SYNC as Q.15, which places the sign bit in the leftmost and the remainder 15 bits are fraction component. Compared with the other 16-bit data type, Q.15 can support the best precision for the data which is less than 1.

We use the IFFT/FFT function from TI C64x DSP library, which supports two types of IFFT/FFT. The former is 32-bit input/output data type; the latter is 16-bit input/output data type. The main reason we choose 32-bit input/output data type is that IFFT/FFT data input must be scaled by the length of IFFT/FFT to prevent overflow. According to IEEE 802.16a, length of IFFT/FFT is 2^{11} . If we choose 16-bit data type before IFFT, only 4 bits can be used to represent the fixed-point value. In our implementation, the data formats before IFFT is Q16.15. Q16.15 places the sign bit in the leftmost, followed by 16 bits integer and 15 bits fraction component. Compared with the other 32-bit data type, Q16.15 can be easily transformed to the 16-bit Q.15 data type.

In order to evaluate the precision of fixed-point format, we compare the uplink syn-

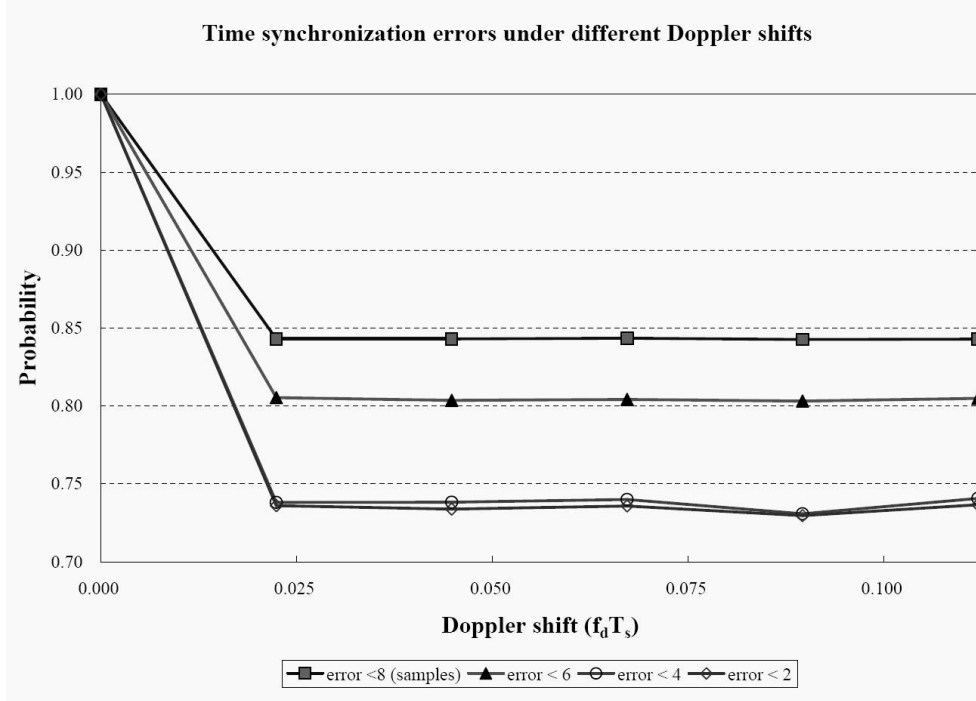


Figure 4.4: Error distribution under different maximum Doppler shifts using time domain approach in fixed-point version.

chronization performance between floating-point system and fixed-point system. Figure 4.4 shows the symbol time synchronization errors of the first coming signal under different Doppler spreads using fixed-point version.

Compared with Figure ??, the uplink synchronization performance of the fixed-point version is very close to that of the floating-point version. Therefore, the Q.15 format fixed-point number is precise enough to the synchronization process.

4.3 Framing/Deframing Structure

4.3.1 Framing

In the original floating-point code, we implement the pseudo random binary sequence (PRBS) generator to do pilot/preamble modulation in framing structure. The polynomial for the PRBS generator is $X^{11} + X^9 + 1$, as Figure 2.7 shows. However, the initialization vector of the PRBS is fixed. In the revised version, we only need to compute the PRBS

```

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

```

Figure 4.5: C code for PRBS generator.

w_k for the first symbol and use the same value for the other symbol. Figures 4.5 and 4.6 show the C code and the compiler's feedback for PRBS generator.

We also do some data type modification to reduce the complexity and to fit the data type of the other program. The two versions of the C code are shown in Figure 4.7 and the compiler's feedback are shown in Figure 4.8.

However, it also has disqualified loop in our fixed-point version, which shown in Figure 4.9. This is because we cannot remove the call function fread() in the for loop.

Table 4.3 shows a comparison of the performance for the floating-point version and the fixed-point version. Both versions use -o3 compiler option to do file-level software pipelining.

4.3.2 Deframing

Compared with framing structure, deframing structure is used after the 2048 FFT block. We need to implement the PRBS generator to do pilot/preamble modulation in framing

```

; *-----*
; *   SOFTWARE PIPELINE INFORMATION
; *   Loop source line           : 141
; *   Loop opening brace source line : 142
; *   Loop closing brace source line : 154
; *   Loop Unroll Multiple       : 2x
; *   Known Minimum Trip Count   : 106
; *   Known Maximum Trip Count   : 106
; *   Known Max Trip Count Factor : 106
; *   Loop Carried Dependency Bound(^) : 11
; *   Unpartitioned Resource Bound : 11
; *   Partitioned Resource Bound(*) : 13
; *   Resource Partition:
; *
; *           A-side   B-side
; *   .L units           0       0
; *   .S units           9      13*
; *   .D units           1       1
; *   .M units           0       0
; *   .X cross paths     4       3
; *   .T address paths   1       1
; *   Long read paths    0       0
; *   Long write paths   0       0
; *   Logical ops (.LS)   0       0       (.L or .S unit)
; *   Addition ops (.LSD) 14      24       (.L or .S or .D unit)
; *   Bound(.L .S .LS)   5       7
; *   Bound(.L .S .D .LS .LSD) 8      13*
; *   Searching for software pipeline schedule at ...
; *       ii = 13 Did not find schedule
; *       ii = 14 Schedule found with 2 iterations in parallel done
; *   Epilog not removed
; *   Collapsed epilog stages : 0
; *   Collapsed prolog stages : 1
; *   Minimum required memory pad : 0 bytes
; *
; *   Minimum safe trip count : 1 (after unrolling)
; *-----*

```

Figure 4.6: Compiler’s feedback for PRBS generator loop.

Table 4.3: Breakdown of Clock Cycles for Framing()

Number of Frames =1 Frame size=4*OFDMA Symbols	Floating-point version	Fixed-point version
Code size (bytes)	1944	2144
Number of execution	4	4
Max. cycles	161128	161902
Min. cycles	102416	11434
Avg. cycles	146253	124089
Total cycles	585013	496357


```

/* before modification */
// UL preamble
for(n=0; n<Nsubcarrier; n++){
    carrier_n_s = Nsubchannels*n + (perbase[(n+ps*s)%Nsubchannels] +
    IDcell*(unsigned short)ceil(((float)(n+1)/(float)Nsubchannels)))%Nsubchannels);

    if( ((wk[carrier_n_s/8]<<(carrier_n_s%8))&(0x80))==0x80 )
        frameout[2*carrier_n_s]= FIXED_DOUBLE_CONST(-1);
    else
        frameout[2*carrier_n_s]= FIXED_DOUBLE_CONST(1);
        frameout[2*carrier_n_s+1]=0;
}
/* after modification */
// UL preamble
for(n=0; n<Nsubcarrier; n++){
    k = (n<32) ? 1:2; //ceil(((float)(n+1)/(float)Nsubchannels));
    carrier_n_s=Nsubchannels*n+(perbase[(n+ps*s)%Nsubchannels]+
    IDcell*k)%Nsubchannels);

    if( ((wk[carrier_n_s/8]<<(carrier_n_s%8))&(0x80))==0x80 )
        frameout[2*carrier_n_s]= FIXED_DOUBLE_CONST(-1);
    else
        frameout[2*carrier_n_s]= FIXED_DOUBLE_CONST(1);
        frameout[2*carrier_n_s+1]=0;
}

```

Figure 4.7: Two versions of C programs for framing.

structure. However, we need to do in deframing structure is to remove the pilot/preamble data. That is to say, we do not need to implement the PRBS generator in deframing structure. That is why the complexity for the framing structure is much more than the deframing structure. We also do some data type modification to fit the data type of the other related code.

Like framing(), we also have a disqualified loop in our fixed-point version, which shown in Figure 4.10. This is because we cannot remove the call function fwrite() in the for loop.

Table 4.4 shows a comparison of the performance for the floating-point version and the fixed-point version.

4.4 IFFT/FFT Structure

In OFDMA system, the modulation/demodulation block can be done efficiently using IFFT/FFT algorithm. According to standard IEEE 802.16a, length of IFFT/FFT (N) is 2048. In the original program, it handles the 32-bit floating-point data type input, output

```

/* before modification */
;-----*
;*  SOFTWARE PIPELINE INFORMATION
;*    Disqualified loop: bad loop structure
;-----*
/* after modification */
;-----*
;*  SOFTWARE PIPELINE INFORMATION
;*    Loop source line           : 76
;*    Loop opening brace source line : 76
;*    Loop closing brace source line : 93
;*    Known Minimum Trip Count      : 53
;*    Known Maximum Trip Count      : 53
;*    Known Max Trip Count Factor   : 53
;*    Loop Carried Dependency Bound(^) : 23
;*    Unpartitioned Resource Bound   : 9
;*    Partitioned Resource Bound(*)  : 9
;*    Resource Partition:
;*
;*          A-side   B-side
;*    .L units           0     1
;*    .S units           8     9*
;*    .D units           4     2
;*    .M units           0     3
;*    .X cross paths     4     3
;*    .T address paths   4     1
;*    Long read paths    0     0
;*    Long write paths   0     0
;*    Logical ops (.LS)   0     0     (.L or .S unit)
;*    Addition ops (.LSD) 11    11     (.L or .S or .D unit)
;*    Bound(.L .S .LS)   4     5
;*    Bound(.L .S .D .LS .LSD) 8     8
;*    Searching for software pipeline schedule at ...
;*      ii = 23 Did not find schedule
;*      ii = 24 Schedule found with 2 iterations in parallel done
;*    Epilog not removed
;*    Collapsed epilog stages : 0
;*    Prolog not removed
;*    Collapsed prolog stages : 0
;*    Minimum required memory pad : 0 bytes
;*    Minimum safe trip count : 2
;-----*

```

Figure 4.8: Compiler's feedback for framing loop before and after optimization.

```


//non_preamble carrier allocation
for(n=0;n<Nsubcarrier;n++){
    carrier_n_s = Nsubchannels*n+(perbase[(n+ps*s)%(Nsubchannels)]+
    IDcell*(unsigned short)ceil(((float)(n+1)/(float)Nsubchannels))%(Nsubchannels));

    if(n==26 || n==L || n==(L+13) || n==(L+27) || n==(L+40)){
        if( ((wk[carrier_n_s/8]<<(carrier_n_s%8))&(0x80))==0x80 )
            frameout[2*carrier_n_s]= FIXED_DOUBLE_CONST(-1.33333333);
        else
            frameout[2*carrier_n_s]= FIXED_DOUBLE_CONST(1.33333333);

        frameout[2*carrier_n_s+1]=0;
    }
    else{
        fread(readin,sizeof(float),2,in);
        frameout[2*carrier_n_s]=readin[0];
        frameout[2*carrier_n_s+1]=readin[1];
    }
}

```

Figure 4.9: A part of C code for framing.



```

for(n=0;n<Nsubcarrier;n++){
    if(n!=26 && n!=L && n!=(L+13) && n!=(L+27) && n!=(L+40)) //pilots: no handling
    {
        k = (n<32) ? 1:2; //ceil(((float)(n+1)/(float)Nsubchannels));
        carrier_n_s=Nsubchannels*n+(perbase[(n+ps*s)%(Nsubchannels)]+
        IDcell*k)%(Nsubchannels);
        readout[0]=framein[2*carrier_n_s];
        readout[1]=framein[2*carrier_n_s+1];
        fwrite(readout,sizeof(FIXED),2,out);
    }
}

```

Figure 4.10: A part of C code for deframing.

Table 4.4: Breakdown of Clock Cycles for Deframing()

Number of Frames =1 Frame size=4*OFDMA Symbols	Floating-point version	Fixed-point version
Code size (bytes)	1992	1992
Number of execution	4	4
Max. cycles	3010	3010
Min. cycles	93	93
Avg. cycles	2112	2112
Total cycles	8451	8451

Table 4.5: IFFT/FFT Function

Functions	Description
DSP_fft32x32	Extended precision, mixed radix FFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 32 bits.
DSP_fft32x32s	Extended precision, mixed radix FFT, digit reversal, out of place, with scaling and rounding. Input and output: 32 bits, Twiddle factor: 32 bits.
DSP_ifft32x32	Extended precision, mixed radix IFFT, digit reversal, out of place, with scaling and rounding. Input and output: 32 bits, Twiddle factor: 32 bits.

and twiddle factor. In order to reduce the computation complexity due to lots of the floating-point multiply operations, we use the IFFT/FFT function from TI C64x DSP library (DSPLIB) to instead.

DSPLIB is an optimized DSP Function Library for C programmers using TMS320C64x devices. It includes many C-callable, assembly-optimized, general-purpose signal processing routines. These routines are typically used in computation-intensive real-time applications where optimal execution speed is critical. By using these routines, we can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language.

TI DSPLIB can support several types of FFT function, which are given in [20]. The original floating-point FFT code uses 32-bit data type input, output and twiddle factors. In order to maintain the precision, we consider the following IFFT/FFT function shown in Table 4.5.

Complex forward mixed radix 32×32 -bit FFT with rounding (DSP_fft32x32) computes an extended precision complex forward mixed radix FFT with rounding and digit reversal. Input data $x[]$, output data $y[]$ and coefficients $w[]$ are 32-bit. The output is returned in the separate array $y[]$ in normal order. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance

in the presence of cache. The FFT coefficients (twiddle factors) are generated using the program “tw_fft32x32”. No scaling is done with the routine; thus the input data must be scaled by $2^{\log_2 N}$ to completely prevent overflow. The routine uses $\log_4 N - 1$ stages of Cooley Tukey radix-4 DIF FFT and performs either a radix-2 or radix-4 DIF FFT on the last stage depending on N . If N is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform. In our work, we have 5 stages of radix-4 transform and 1 stage radix-2 transform.

Complex forward mixed radix 32- \times 32-bit FFT with scaling (DSP_fft32x32s) computes an extended precision complex forward mixed radix FFT with scaling, rounding and digit reversal. DSP_fft32x32s and DSP_fft32x32 are very similar. The only difference is that for DSP_fft32x32s, scaling by 2 takes place at each radix-4 stage except for the last one. A radix-4 stage can add a maximum of 2 bits, which would require scaling by 4 to completely prevent overflow. Thus, the input data must be scaled by $2^{\log_2 N - \text{ceil}[\log_4 N - 1]}$.

Complex inverse mixed radix 32- \times 32-bit FFT with rounding (DSP_ifft32x32) computes an extended precision complex inverse mixed radix FFT with rounding and digit reversal. In reality we can re-use DSP_fft32x32 to perform IFFT, by first conjugating the input, performing the FFT, conjugating again. This allows DSP_fft32x32 to perform the IFFT as well. However if the double conjugation needs to be avoided then this routine uses the same twiddle factors as the FFT and performs an IFFT. The change in the sign of the twiddle factors is adjusted for in the routine. Hence this routine uses the same twiddle factors as the DSP_fft32x32 routine.

Table 4.6 shows a comparison of the two FFT functions for $N = 2048$. In order to achieve better precision, we adopt DSP_ifft32x32 and DSP_fft32x32 to do IFFT/FFT.

As Table 2.3 shows, it needs 19203 real multiplications and 64259 real additions for radix-4 DIF IFFT/FFT theoretically. Practically, the time DSP_ifft32x32/DSP_fft32x32 needed is 28811 clock cycles. We list the complexity and performance of IFFT/FFT in Table 4.7.

Table 4.6: Comparison of Different IFFT/FFT

Function	DSP_fft32x32	DSP_fft32x32s	DSP_ifft32x32
Code Size (Bytes)	932	932	932
Clock Cycles	28811	28811	28811
Scaled Bits	11	6	11
Architecture	5 stages of radix-4 transform and 1 stage radix-2 transform		

Table 4.7: Complexity and Performance of IFFT/FFT Implementation

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

Figure 4.11 shows a part of hand assembly code for DSP_fft32x32.

The reason for DSP_ifft32x32/DSP_fft32x32 only need fewer cycles is that it use some intrinsics to reduce complexity. The 32 by 32 multiplies are done with a 1.5 bit loss in accuracy. This comes about because the contribution of the low 16 bits to the 32 bit result is not computed. In addition the contribution of the low * high term is shifted by 16 as opposed to 15, for a loss of 0.5 bits after rounding. The real part of complex multiply is given by

$$(X + jY) \times (C + jS) = \text{_mpyhir}(si10, yt1_0) + \text{_mpyhir}(co10, xt1_0) + (\text{_dotprsu2}(yt1_0xt1_0, si10co10) \ll 1)$$

where the functions `_mpyhir` and `_dotprsu2` are shown in Table 4.8.

Tables 4.9 and 4.10 show a comparison of the IFFT/FFT performance for the floating-point version and the fixed-point version.

```

[!A_pro]STDW .D2T2 B_y_l2_1:B_y_l2_0, *B_x_[B_l2] ;[25,2]
|| SUB .L1 A_p2c, A_p3c, A_y_l1_1 ;[25,2]y[11+1]=co20*yt0-
|| ADDAH .D1 A_y_l1_0, A_p23r, A_y_l1_0 ;[25,2] si20*xt0)>>15
|| ADD .L2X B_p0r, A_p1r, B_y_h2_0 ;[25,2]y[h2] = (si10*yt1+
|| MPYHIR .M2 B_co30, B_yt2, B_p4c ;[15,3] co10*xt1)>>15
|| MPYHIR .M1X A_si10, B_xt1, A_p1c ;[15,3]
|| PACK2 .S2 B_si10, B_co10, B_si10co10 ;[15,3] ()>>16
|| SUB .S1 A_xh0, A_xh20, A_xt0 ;[15,3] xt0=xh0-xh20
|| ADDAH .D2 B_y_h2_0, B_p01r, B_y_h2_0 ;[26,2]
[!B_pro2]STDW .D1T1 A_y_h1_1:A_y_h1_0, *A_x_1[0] ;[16,3]
|| MPYHIR .M2 B_si30, B_xt2, B_p5c ;[16,3]
|| MPYHIR .M1 A_co20, A_yt0, A_p2c ;[16,3]
|| PACK2 .S1 A_si20, A_co20, A_si20co20 ;[16,3] ()>>16
|| PACK2 .L2 B_co30, B_si30, B_co30si30 ;[16,3] ()>>16
|| SUB .L1X B_fft_jmp, A_j, A_ifj ;[ 6,4] ifj = (j - fft_jmp)
|| MV .S2X A_j, B_j ;[ 6,4]
|| BDEC .S1 LOOP_Y, A_i ;[37,1]
|| MPYHIR .M2 B_co30, B_xt2, B_p4r ;[17,3]
|| MPYHIR .M1 A_si20, A_yt0, A_p3r ;[17,3]
|| PACKH2 .S2 B_yt2, B_xt2, B_yt2xt2 ;[17,3]
|| LDDW .D2T1 *B_w1[B_j], A_co20:A_si20 ;[ 7,4]
|| LDDW .D1T2 *A_w0[A_j], B_co10:B_si10 ;[ 7,4]
|| SUB .L2X B_xp1, A_x11p1, B_x11 ;[ 7,4] x11=x[1]-x[11p1]
|| ADD .L1X B_xp0, A_x11p0, A_xh0 ;[ 7,4] xh0=x[0]+x[11]
[!A_pro]STDW .D2T2 B_y_h2_1:B_y_h2_0, *B_x_[B_h2] ;[28,2]
|| ADDAH .D1 A_y_l1_1, A_p23c, A_y_l1_1 ;[28,2]
|| DOTPRSU2 .M2 B_yt2xt2, B_si30co30, B_p45r ;[18,3]
|| MPYHIR .M1 A_co20, A_xt0, A_p2r ;[18,3]
|| PACKH2 .L1 A_yt0, A_xt0, A_yt0xt0 ;[18,3]
|| PACK2 .S2 B_co10, B_si10, B_co10si10 ;[18,3] ()>>16
|| SUB .L2X B_xp0, A_x11p0, B_x10 ;[ 8,4] x10=x[0]-x[11]
|| ADD .S1X B_xp1, A_x11p1, A_xh1 ;[ 8,4] xh1=x[1]+x[11p1]

```

Figure 4.11: A part of assembly code for DSP_fft32x32.

Table 4.8: Used Compiler Intrinsic in DSP_ifft32x32/DSP_fft32x32

C Compiler Intrinsic	Description
int_mpyhir (int src1, int src2);	Produces a signed 16 by 32 multiply.
int_dotprsu2 (int src1, uint src2);	The product of the first signed pair of 16-bit values is added to the product of the unsigned second pair of 16-bit values in src1 and src2. 2^{15} is added and the result is sign shifted right by 16.

Table 4.9: Breakdown of Clock Cycles for IFFT()

Number of Frames =1 Frame size=4*OFDMA Symbols	Floating-point version	Fixed-point version
Code size (bytes)	632	936
Number of execution	4	4
Max. cycles	21278850	35710
Min. cycles	20850744	35710
Avg. cycles	21164023	35710
Total cycles	84656093	142840



Table 4.10: Breakdown of Clock Cycles for FFT()

Number of Frames =1 Frame size=4*OFDMA Symbols	Floating-point version	Fixed-point version
Code size (bytes)	356	276
Number of execution	4	4
Max. cycles	22151436	32247
Min. cycles	22151436	32247
Avg. cycles	22151436	32247
Total cycles	88605744	128988

Table 4.11: Breakdown of Clock Cycles for TX_SRRC()

Number of Frames =1 Frame size=4*OFDMA Symbols	Floating-point version	Fixed-point version
Code size (bytes)	1864	1624
Number of execution	4	4
Max. cycles	48790975	6199459
Min. cycles	48772795	6199459
Avg. cycles	48782144	6199459
Total cycles	195128576	24797836

4.5 Transmission Filtering

4.5.1 Oversampling and SRRC Filter in the Transmitter

In order to provide the ability to simulate path delays at non-integer sample times, an interpolator is added to the transmitter to yield 4-times oversampled transmitter output. In our system, we adopt the 57-taps SRRC filter with $\alpha = 0.155$. For the same reason, we shorten the data type from 32-bit floating-point to 16-bit fixed-point in the revised program.

When we use `-O3` compiler option, the compiler can do software pipelining automatically. However, `Tx_SRRC()` still has the disqualified loop in the call function `mul_sum()`, which shown in Figure 4.12.

This is because the call function `mul_sum()` has serious data dependency problem. `mul_sum()` is frequently used in `Tx_SRRC()`. This is the main reason why `Tx_SRRC()` cannot achieve real-time processing.

Table 4.11 shows a comparison of the performance for the floating-point and the fixed-point version. The fixed-point version is 7.87 times faster than the floating-point version.

4.5.2 Downsampling and SRRC Filter in the Receiver

Unlike `Tx_SRRC()`, `Rx_SRRC()` has the qualified loop in the call function `mul_sum()`, which shown in Figure 4.13.

```

void mul_sum( FIXED *input, int j, FIXED *output, char tail)
{
    char m,n;
    FIXED_DOUBLE TempOut0=0,TempOut1=0,TempOut2=0,TempOut3=0;
    for(m=mul_per_point;m>=(mul_per_point-j);m--)
    {
        TempOut0=TempOut0+input[m]*LPF_coefficient_0[m];
        if( (n=(m-1)) >=0 )
        {
            if(!tail)
            {
                TempOut1=TempOut1+input[m]*LPF_coefficient_1[n];
                TempOut2=TempOut2+input[m]*LPF_coefficient_2[n];
                TempOut3=TempOut3+input[m]*LPF_coefficient_3[n];
            }
            else
            {
                TempOut1=TempOut1+input[n]*LPF_coefficient_1[n];
                TempOut2=TempOut2+input[n]*LPF_coefficient_2[n];
                TempOut3=TempOut3+input[n]*LPF_coefficient_3[n];
            }
        }
    }
    output[0]=TempOut0>>15;
    output[1]=TempOut1>>15;
    output[2]=TempOut2>>15;
    output[3]=TempOut3>>15;
}

;-----*
;*  SOFTWARE PIPELINE INFORMATION
;*  Disqualified loop: bad loop structure
;-----*

```

Figure 4.12: C code for mul_sum() in Tx_SRRC().

```

void mul_sum( FIXED *input, FIXED *output){
    char m;
    FIXED_DOUBLE AccBuf=0;
    for(m=0;m<57;m++) AccBuf=AccBuf+(FIXED_DOUBLE) (input[m]*LPF_coefficient[m]);
    *output = AccBuf >> 15;
}
;-----*
;*  SOFTWARE PIPELINE INFORMATION
;*  Loop source line           : 65
;*  Loop opening brace source line : 66
;*  Loop closing brace source line : 66
;*  Loop Unroll Multiple       : 4x
;*  Known Minimum Trip Count   : 14
;*  Known Maximum Trip Count   : 14
;*  Known Max Trip Count Factor : 14
;*  Loop Carried Dependency Bound(^) : 0
;*  Unpartitioned Resource Bound : 2
;*  Partitioned Resource Bound(*) : 2
;*  Resource Partition:
;*
;*          A-side   B-side
;*  .L units           0       0
;*  .S units           0       1
;*  .D units           2*      0
;*  .M units           2*      0
;*  .X cross paths     0       0
;*  .T address paths   2*      1
;*  Long read paths    0       0
;*  Long write paths   0       0
;*  Logical ops (.LS)  0       0      (.L or .S unit)
;*  Addition ops (.LSD) 2       0      (.L or .S or .D unit)
;*  Bound(.L .S .LS)   0       1
;*  Bound(.L .S .D .LS .LSD) 2*    1
;*  Searching for software pipeline schedule at ...
;*  ii = 2  Schedule found with 6 iterations in parallel done
;*  Collapsed epilog stages : 5
;*  Prolog not entirely removed
;*  Collapsed prolog stages : 3
;*  Minimum required memory pad : 40 bytes
;*  Minimum safe trip count : 1 (after unrolling)
;-----*

```

Figure 4.13: C code and compiler's feedback for mul_sum() loop.

```

for(j=0;j<29;j++)
{
    fread(SRRC_temp,sizeof(FIXED),2,in);
    SRRC_buffer_real[28-j]=SRRC_temp[0];
    SRRC_buffer_imag[28-j]=SRRC_temp[1];
}
}
-----*
;*
;*  SOFTWARE PIPELINE INFORMATION
;*    Disqualified loop: loop contains a call
;*-----*

```

Figure 4.14: C code and compiler's feedback for Rx_SRRC() loop.

Table 4.12: Breakdown of Clock Cycles for RX_SRRC()

Number of Frames =1 Frame size=4*OFDMA Symbols	Floating-point version	Fixed-point version
Code size (bytes)	508	564
Number of execution	9216	9216
Max. cycles	23985	11991
Min. cycles	16866	2301
Avg. cycles	21868	2302
Total cycles	302305027	21215706

However, Rx_SRRC() also has the disqualified loop, which shown in Figure 4.14. This is because the for loop contains the call function fread().

Table 4.12 shows a comparison of the performance for the floating-point and the fixed-point version. The fixed-point version is 14.25 times faster than the floating-point version.

4.6 Uplink Synchronization Using Time Domain Approach

The main operations in the uplink synchronization are CP correlation and preamble correlation. In this section, the preamble correlation is used by time domain approach.

4.6.1 CP_Correlation

CP_correlation() is the function to do the unlink synchronization stage I. In the original program, we need to use 32-bit by 32-bit floating-point multiply operations. In TI C6416

```

/* before modification */
for(i=0;i<CP_downsampling_samples;i++)
{
    *CP_real=*CP_real+sync_buffer_1_real[i]*sync_buffer_1_real[i+2048]+
        sync_buffer_1_imag[i]*sync_buffer_1_imag[i+2048];
    *CP_imag=*CP_imag+sync_buffer_1_real[i]*sync_buffer_1_imag[i+2048]-
        sync_buffer_1_imag[i]*sync_buffer_1_real[i+2048];
}

```

Figure 4.15: C code in CP_correlation() before optimization.

DSP, the .M unit deals with 16-bit by 16-bit fixed-point multiply operations. In the revised program, we shorten the data type from 32-bit floating-point to 16-bit fixed-point.

Software pipelining is also used in this function to improve the performance. In addition to use `-o3` compiler option to do software pipelining, we also do loop unrolling by hand. The two versions of the C code are shown in Figures 4.15 and 4.16. The clock cycles of the two versions are 5474 and 1291. That is, the modified version is 4.24 times faster than the original version.

After loop unrolling, some kinds of the original code can be viewed as calculating $a[i] * a[j] + a[i + 1] * a[j + 1]$ where $a[i]$ is a 16-bit value. In this condition, the compiler use the C6000 instruction DOTP2 to replace the two 16-bit by 16-bit fixed-point multiply operations and one 16-bit by 16-bit fixed-point addition operation. Details for the instruction DOTP2 can be found in [13]. This is the main reason why we can reduce the complexity. Figures 4.17 and 4.18 show the compiler's feedback for the loop in the CP_correlation function.

A comparison of the performance for the floating-point and the fixed-point version is shown in Table 4.13. The fixed-point version is 13.39 times faster than the floating-point version.

4.6.2 Preamble_correlation

Preamble_correlation() is the function to do the unlink synchronization stage II by using time domain approach. Like CP_correlation(), Preamble_correlation() also needs to com-

```

/* after modification */
for(i=0;i<CP_downsampling_samples;i=i+8)
{
    *CP_real=*CP_real+sync_buffer_1_real[i] *sync_buffer_1_real[i+2048]+
    sync_buffer_1_imag[i] *sync_buffer_1_imag[i+2048]+
    sync_buffer_1_real[i+1]*sync_buffer_1_real[i+2049]+
    sync_buffer_1_imag[i+1]*sync_buffer_1_imag[i+2049]+
    sync_buffer_1_real[i+2]*sync_buffer_1_real[i+2050]+
    sync_buffer_1_imag[i+2]*sync_buffer_1_imag[i+2050]+
    sync_buffer_1_real[i+3]*sync_buffer_1_real[i+2051]+
    sync_buffer_1_imag[i+3]*sync_buffer_1_imag[i+2051]+
    sync_buffer_1_real[i+4]*sync_buffer_1_real[i+2052]+
    sync_buffer_1_imag[i+4]*sync_buffer_1_imag[i+2052]+
    sync_buffer_1_real[i+5]*sync_buffer_1_real[i+2053]+
    sync_buffer_1_imag[i+5]*sync_buffer_1_imag[i+2053]+
    sync_buffer_1_real[i+6]*sync_buffer_1_real[i+2054]+
    sync_buffer_1_imag[i+6]*sync_buffer_1_imag[i+2054]+
    sync_buffer_1_real[i+7]*sync_buffer_1_real[i+2055]+
    sync_buffer_1_imag[i+7]*sync_buffer_1_imag[i+2055];
    *CP_imag=*CP_imag+sync_buffer_1_real[i] *sync_buffer_1_imag[i+2048]-
    sync_buffer_1_imag[i] *sync_buffer_1_real[i+2048]+
    sync_buffer_1_real[i+1]*sync_buffer_1_imag[i+2049]-
    sync_buffer_1_imag[i+1]*sync_buffer_1_real[i+2049]+
    sync_buffer_1_real[i+2]*sync_buffer_1_imag[i+2050]-
    sync_buffer_1_imag[i+2]*sync_buffer_1_real[i+2050]+
    sync_buffer_1_real[i+3]*sync_buffer_1_imag[i+2051]-
    sync_buffer_1_imag[i+3]*sync_buffer_1_real[i+2051]+
    sync_buffer_1_real[i+4]*sync_buffer_1_imag[i+2052]-
    sync_buffer_1_imag[i+4]*sync_buffer_1_real[i+2052]+
    sync_buffer_1_real[i+5]*sync_buffer_1_imag[i+2053]-
    sync_buffer_1_imag[i+5]*sync_buffer_1_real[i+2053]+
    sync_buffer_1_real[i+6]*sync_buffer_1_imag[i+2054]-
    sync_buffer_1_imag[i+6]*sync_buffer_1_real[i+2054]+
    sync_buffer_1_real[i+7]*sync_buffer_1_imag[i+2055]-
    sync_buffer_1_imag[i+7]*sync_buffer_1_real[i+2055];
}

```

Figure 4.16: C code in CP_correlation() after optimization.

Table 4.13: Breakdown of Clock Cycles for CP_correlation()

Number of Frames =1 Frame size=4*OFDMA Symbols	Floating-point version	Fixed-point version
Code size (bytes)	860	756
Number of execution	512	512
Max. cycles	56247	1291
Min. cycles	660	55
Avg. cycles	768	57
Total cycles	393507	29396

```

; *-----*
; * SOFTWARE PIPELINE INFORMATION
; *
; * Loop source line : 98
; * Loop opening brace source line : 99
; * Loop closing brace source line : 104
; * Loop Unroll Multiple : 4x
; * Known Minimum Trip Count : 64
; * Known Maximum Trip Count : 64
; * Known Max Trip Count Factor : 64
; * Loop Carried Dependency Bound(^) : 76
; * Unpartitioned Resource Bound : 12
; * Partitioned Resource Bound(*) : 16
; * Resource Partition:
; *
; *          A-side   B-side
; * .L units           0       0
; * .S units           1       0
; * .D units          12      12
; * .M units           4       4
; * .X cross paths     5       5
; * .T address paths  16*    16*
; * Long read paths    0       0
; * Long write paths   0       0
; * Logical ops (.LS)  1       2   (.L or .S unit)
; * Addition ops (.LSD) 8       6   (.L or .S or .D unit)
; * Bound(.L .S .LS)   1       1
; * Bound(.L .S .D .LS .LSD) 8       7
; *
; * Searching for software pipeline schedule at ...
; *     ii = 76 Did not find schedule
; *     ii = 77 Did not find schedule
; *     ii = 79 Did not find schedule
; * Disqualified loop: did not find schedule
; *-----*

```

Figure 4.17: Compiler's feedback for CP_correlation() loop before optimization.

```

; *-----*
; * SOFTWARE PIPELINE INFORMATION
; *
; *   Loop source line           : 98
; *   Loop opening brace source line : 99
; *   Loop closing brace source line : 132
; *   Known Minimum Trip Count     : 32
; *   Known Maximum Trip Count     : 32
; *   Known Max Trip Count Factor   : 32
; *   Loop Carried Dependency Bound(^) : 38
; *   Unpartitioned Resource Bound  : 8
; *   Partitioned Resource Bound(*)  : 16
; *   Resource Partition:
; *
; *           A-side   B-side
; *   .L units           0       0
; *   .S units           1       0
; *   .D units           4       8
; *   .M units           9       7
; *   .X cross paths     6      16*
; *   .T address paths   9       11
; *   Long read paths    0       0
; *   Long write paths   0       0
; *   Logical ops (.LS)  0       0      (.L or .S unit)
; *   Addition ops (.LSD) 1       29     (.L or .S or .D unit)
; *   Bound(.L .S .LS)   1       0
; *   Bound(.L .S .D .LS .LSD) 2     13
; *
; *   Searching for software pipeline schedule at ...
; *       ii = 38 Schedule found with 2 iterations in parallel done
; *
; *   Epilog not removed
; *   Collapsed epilog stages : 0
; *   Collapsed prolog stages : 1
; *   Minimum required memory pad : 0 bytes
; *   Minimum safe trip count : 1
; *-----*

```

Figure 4.18: Compiler's feedback for CP_correlation() loop after optimization.


```

/* before modification */
for (user=0;user<2;user++)
{
    real=0;
    imag=0;
    for (i=0;i<2048;i++)
    {
        if (user==0)
            complex_mul(&ref1[2*i],&ref1[2*i+1], &sync_buffer_1_real[2047-i],
                &sync_buffer_1_imag[2047-i], &temp_real,&temp_imag);
        else //user==1
            complex_mul(&ref2[2*i],&ref2[2*i+1], &sync_buffer_1_real[2047-i],
                &sync_buffer_1_imag[2047-i], &temp_real,&temp_imag);
        real=real+temp_real;
        imag=imag+temp_imag;
    }
    temp=(real>>15)*(real>>15)+(imag>>15)*(imag>>15);
    if (temp>peak[user])
    {
        peak[user]=temp;
        location[user]=count;
    }
}
void complex_mul(FIXED *in1_real, FIXED *in1_imag, FIXED *in2_real, FIXED *in2_imag,
    FIXED_DOUBLE *out_real, FIXED_DOUBLE *out_imag)
{
    *out_real=(FIXED_DOUBLE) (FIXED_MUL(*in1_real,(*in2_real))+FIXED_MUL(*in1_imag,(*in2_imag)));
    *out_imag=(FIXED_DOUBLE) (FIXED_MUL(*in1_imag,(*in2_real))-FIXED_MUL(*in1_real,(*in2_imag)));
}

```

Figure 4.19: C code in Preamble_correlation() before optimization.

pute a lot of 32-bit by 32-bit floating-point multiply operations to get the symbol start time accuracy. In order to reduce the complexity, we need to shorten the data type from 32-bit floating-point to 16-bit fixed-point in order to satisfy the characteristics of C6416 DSP.

In the original program, it needs to call `complex_mul()` function in for loop. We rewrite this part by doing complex multiplications directly to decrease the number of NOPs and achieve better pipelining. Figures 4.19 and 4.20 show the two versions of the code.

The clock cycles of the two versions are 258148 and 8327. That is, the modified version is 31 times faster than the original version. Figure 4.21 shows the compiler's feedback for the loop in the `Preamble_correlation` function. When we remove the call function `complex_mul()` in the for loop, software pipelining can be done better. That is the main reason why we can reduce the complexity.

Table 4.14 shows a comparison of the performance for the floating-point and the fixed-point version. The fixed-point version is 379.53 times faster than the floating-point ver-

```

/* after modification */
for (user=0;user<2;user++)
{
    real=0;
    imag=0;
    for(i=0;i<2048;i++)
    {
        if(user==0)
        {
            real=real+ ref1[2*i]*sync_buffer_1_real[2047-i]
            +ref1[2*i+1]*sync_buffer_1_imag[2047-i];
            imag=imag+ref1[2*i+1]*sync_buffer_1_real[2047-i]
            -ref1[2*i]*sync_buffer_1_imag[2047-i];
        }
        else //user==1
        {
            real=real+ ref2[2*i]*sync_buffer_1_real[2047-i]
            +ref2[2*i+1]*sync_buffer_1_imag[2047-i];
            imag=imag+ref2[2*i+1]*sync_buffer_1_real[2047-i]
            -ref2[2*i]*sync_buffer_1_imag[2047-i];
        }
    }
    temp=(real>>15)*(real>>15)+(imag>>15)*(imag>>15);
    if(temp>peak[user])
    {
        peak[user]=temp;
        location[user]=count;
    }
}

```

Figure 4.20: C code in Preamble_correlation() after optimization.

```

/* before modification */
;-----*
;*   SOFTWARE PIPELINE INFORMATION
;*   Disqualified loop: bad loop structure
;-----*
/* after modification */
;-----*
;*   SOFTWARE PIPELINE INFORMATION
;*   Loop source line           : 197
;*   Loop opening brace source line : 198
;*   Loop closing brace source line : 226
;*   Loop Unroll Multiple       : 4x
;*   Known Minimum Trip Count    : 512
;*   Known Maximum Trip Count    : 512
;*   Known Max Trip Count Factor : 512
;*   Loop Carried Dependency Bound(^) : 2
;*   Unpartitioned Resource Bound : 8
;*   Partitioned Resource Bound(*) : 8
;*   Resource Partition:
;*
;*           A-side   B-side
;*   .L units           0       0
;*   .S units           1       0
;*   .D units           2       2
;*   .M units           8*      8*
;*   .X cross paths     4       4
;*   .T address paths   4       4
;*   Long read paths    0       0
;*   Long write paths   0       0
;*   Logical ops (.LS)  0       0       (.L or .S unit)
;*   Addition ops (.LSD) 8       8       (.L or .S or .D unit)
;*   Bound(.L .S .LS)   1       0
;*   Bound(.L .S .D .LS .LSD) 4       4
;*   Searching for software pipeline schedule at ...
;*       ii = 8  Schedule found with 3 iterations in parallel done
;*   Epilog not entirely removed
;*   Collapsed epilog stages : 1
;*   Prolog not removed
;*   Collapsed prolog stages : 0
;*   Minimum required memory pad : 16 bytes
;*   Minimum safe trip count : 2 (after unrolling)
;-----*

```

Figure 4.21: Compiler's feedback for Preamble_correlation() loop before and after optimization.

Table 4.14: Breakdown of Clock Cycles for Preamble_correlation()

Number of Frames =1 Frame size=4*OFDMA Symbols	Floating-point version	Fixed-point version
Code size (bytes)	808	1416
Number of execution	256	256
Max. cycles	3160370	8327
Min. cycles	3160370	8327
Avg. cycles	3160370	8327
Total cycles	809054720	2131712

sion.

4.6.3 Complexity Analysis

The DSP chip has 2 units to perform multiplication and 6 units for addition. When we analyze the complexity, we focus on the multiplications and the additions in our program. The data amount we consider is 4 OFDMA symbols, equal to a frame. Each OFDMA symbol has 2304 samples. We only do uplink synchronization scheme on the first symbol of one frame.

Figure 2.10 shows the structure of this time offset estimator using CP correlation information. For the first 256 samples, the real multiplications we need is

$$256 \times \underbrace{4}_{\text{complex multiplication}} = 1024.$$

The real addition we need is

$$256 \times \underbrace{2}_{\text{from complex multiplication}} + 255 \times \underbrace{2}_{\text{complex addition}} = 1022.$$

For the other samples, the multiplications we need is

$$2 \times \underbrace{4}_{\text{complex multiplication}} + \underbrace{2}_{\text{modulus}} = 10.$$

The real additions we need is

$$2 \times \underbrace{2}_{\text{from complex multiplication}} + \underbrace{1}_{\text{from modulus}} = 5.$$

Table 4.15: Complexity and Performance of CP Correlation Implementation

Number of Frames =1	Needed Number of Clock Cycles	Equivalent Number of Clock Cycles	Performance
CP Correlation Case 1	682	1291	52.85%
CP Correlation Case 2	6	55	10.61%

The total real multiplications per frame is

$$1024 + 10 \times \underbrace{511}_{range} = 6134.$$

The total real additions per frame is

$$1022 + 5 \times \underbrace{511}_{range} = 3577.$$

We list the complexity and performance of CP correlation in Table 4.15.

Figure 2.14 shows the structure using preamble correlation information. For each user, the real multiplications we need is:

$$2048 \times \underbrace{4}_{complex\ multiplication} \times \underbrace{128}_{range} = 1048576.$$

The real additions we need is:

$$2048 \times \underbrace{2}_{from\ complex\ multiplication} \times \underbrace{128}_{range} + 2047 \times \underbrace{2}_{complex\ addition} \times \underbrace{128}_{range} = 1048320.$$

In our work, the number of users is 2. The total multiplications per frame is

$$1048576 \times \underbrace{2}_{user\ num} = 2097152.$$

The total additions per frame is

$$1048320 \times \underbrace{2}_{user\ num} = 2096640.$$

The total clock cycles we need is

$$2097152 \times \frac{1}{2} + 2096640 \times \frac{1}{6} = 1398016.$$

We list the complexity and performance of preamble correlation in Table 4.16.

Table 4.16: Complexity and Performance of Preamble Correlation Implementation

Number of Frames =1	Needed Number of Clock Cycles	Equivalent Number of Clock Cycles	Performance
Preamble Correlation	1398016	2131712	65.58%

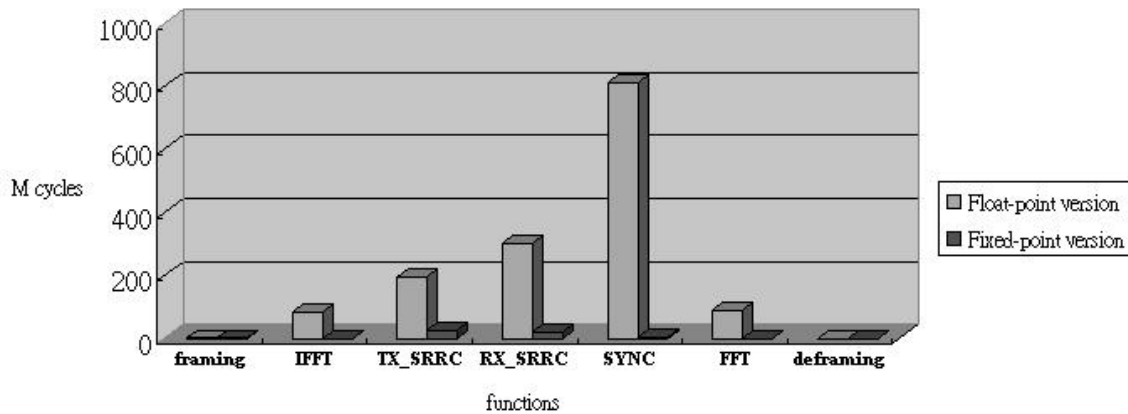


Figure 4.22: Comparison between floating-point version and fixed-point version.

4.7 Conclusion in Optimization

Figure 4.22 shows the comparison between floating-point version and fixed-point version. The clock cycle for IFFT/FFT is reduced from 84.6 and 88.6 Mcycles to 0.14 and 0.12 Mcycles, which is 99.83% and 99.85% reduction. The clock cycle for Tx_SRRC/Rx_SRRC is reduced from 195.1 and 302.3 Mcycles to 24.7 and 21.2 Mcycles, which is 87.29% and 92.98% reduction. The clock cycle for SYNC is reduced from 809.4 Mcycles to 2.1 Mcycles, which is 99.73% reduction.

Framing/deframing and IFFT/FFT can achieve real-time computation, but SYNC and Tx_SRRC/Rx_SRRC cannot for frame size=4×OFDMA symbols. The major reason SYNC cannot achieve real-time for frame size =4×OFDMA symbols is that the DSP can only support 0.97 Mcycles but the SYNC function needs 2.1 Mcycles theoretically and 4.3Mcycles practically. The SYNC function can achieve real-time computation for frame size=19×OFDMA symbols. However, Tx_SRRC/Rx_SRRC cannot achieve real-

time computation by using larger frame size. We need to rewrite Tx_SRRC/Rx_SRRC function to do real-time processing.



Chapter 5

Conclusion and Future Work

5.1 Conclusion

We considered implementation of TDD OFDMA uplink synchronization scheme on TI's C6416 digital signal processor. The implementation is based on the code from [5], which is floating-point version. We rewrote the original code to fixed-point version. In order to verify the accuracy of the fixed-point uplink synchronization scheme, the framing/deframing structure, Tx/Rx SRRC filter and IFFT/FFT have been also implemented.

For DSP implementation, we use some optimization methods to reduce computation complexity. We use the software pipeline for faster execution. In our work, we try to make use of the eight functional units of C6416 DSP and increase the parallel instructions as much as we can. We use the TI DSPLIB function `DSP_ifft32x32` and `DSP_fft32x32` from TI's DSPLIB to replace the original floating-point IFFT/FFT function. Overall, the clock cycle for IFFT/FFT is reduced from 84.6 and 88.6 Mcycles to 0.14 and 0.12 Mcycles, which is 99.83% and 99.85% reduction. The clock cycle for Tx_SRRC/Rx_SRRC is reduced from 195.1 and 302.3 Mcycles to 24.7 and 21.2 Mcycles, which is 87.29% and 92.98% reduction. The clock cycle for SYNC is reduced from 809.4 Mcycles to 2.1 Mcycles, which is 99.73% reduction. However, due to the malfunction of the Quixote board, we are not able to actually transmit data between the DSP board and the host PC. We are forced to simulate on the TI DSP simulator, which is built in the TI code composer studio. Therefore, the simulation profile shown in this thesis may not be the exact profile

obtained from the DSP emulator operated on Quixote board. But we believe that the optimizations we have done based on the DSP simulator are also valid on Quixote board, and the improving rate should also be similar.

We introduced the TDD OFDMA uplink synchronization schemes in chapter 2. In uplink, two methods were presented to do time synchronization. One is using the correlation in the frequency domain and the other is in the time domain. The correlation in the time domain has better performance when the channel is not slow fading. The time synchronization errors are in some degree correlated to the channel model. Thus the guard interval should be at least larger than two times of the delay spread. The DSP system was introduced in chapter 3. We briefly described the structure of the Quixote DSP board, the C6416 DSP chips, the transmission mechanism and the Code Composer Studio. We implemented our program into the DSP. The process and performance were recorded in chapter 4.

5.2 Potential Future Work

Due to the board hardware defect and/or system software bug, we are unable to run and test our implementation on the DSP baseboard yet. The DSP implementation should be executed on the DSP baseboard, and the streaming interface is needed to connect to the Host PC in real time execution. The Host PC reads in the source data from the file in the memory, and then it transfers the data to DSP through the streaming interface. After DSP has processed data, it transfers data back to the Host PC.

Bibliography

- [1] S. Kaiser and K. Fazel, "A spread-spectrum multicarrier multiple-access system for mobile communications," *Proc. 1st Int. Workshop on Multicarrier Spread Spectrum*, pp. 45–56, Apr. 1997.
- [2] 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. 1900–914, Nov. 1999.
- [3] H. Sari and G. Karam, "Orthogonal frequency-division multiple access and its application to CATV networks," *Eur. Trans. Telecommun.*, vol. 9, pp. 507–516, Dec. 1998.
- [4] IEEE Std 802.16a-2003, *IEEE Standard for Local and Metropolitan Area Networks — Part 16: Air Interface for Fixed Broadband Wireless Access Systems — Amendment 2: Medium Access Control Modifications and Additional Physical Layer Specifications for 2–11GHz*. New York: IEEE, April 1, 2003.
- [5] 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.

- [6] O. Edfors, M. Sandell, J. J. van de Beek, D. Landstrom, and F. Sjoberg, "An introduction to orthogonal frequency-division multiplexing," <http://courses.ece.uiuc.edu/ece459/spring02/ofdm tutorial.pdf>.
- [7] M. Morelli, "Timing and Frequency Synchronization for the Uplink of an OFDMA System," *IEEE Trans. Commun.*, vol. 52, pp. 296–306, Feb. 2004.
- [8] 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.
- [9] C. K. Chang, "Investigation and design of FFT core for OFDM communication systems," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2002.
- [10] 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.
- [11] Innovative Integration, *Quixote User's Manual*, Dec. 2003.
- [12] Innovative Integration, *Quixote Data Sheet*, <http://www.innovative-dsp.com/support/datasheets/quixote.pdf>.
- [13] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, literature no. SPRU189F, Oct. 2000.
- [14] Texas Instruments, *TMS320C6414T, TMS320C6415T, TMS320C6416T Fixed-Point Digital Signal Processors*, literature no. SPRS226A, Mar. 2004.
- [15] Texas Instruments, *TMS320C64x Technical Overview*, literature no. SPRU395B, Jan. 2001.
- [16] Texas Instruments, *Code Composer User's Guide*, literature no. SPRU296, Feb. 1999.

- [17] Texas Instruments, *TMS320C6000 Code Composer Studio Getting Started Guide*, literature no. SPRU509D, Aug. 2003.
- [18] Texas Instruments, *TMS320C6000 Optimizing Compiler User Guide*, literature no. SPRU187K, Oct. 2002.
- [19] Texas Instruments, *TMS320C6000 Programmer's Guide*, literature no. SPRU198G, Aug. 2002.
- [20] Texas Instruments, *TMS320C64x DSP Library Programmer's Reference*, literature no. SPRU565B, Oct. 2003.



作者簡歷

林筱晴，民國六十九年十二月出生於桃園縣。民國九十一年六月畢業於國立交通大學電子工程學系，並於同年九月進入國立交通大學電子研究所就讀，從事通訊系統方面相關研究。民國九十三年六月取得碩士學位，碩士論文題目為『IEEE 802.16a 分時雙工正交分頻多重進接之上行同步技術研討與在數位訊號處理器上的實現』。研究範圍與興趣包括：通訊系統、信號處理及通道編碼。

