電機資訊學院 電子與光電學程

碩 士 論 文

在多數位訊號處理器系統上進行高效率無線通道模擬之研討

Research in Efficient Wireless Channel Simulation on Multi-DSP Platform

研 究 生 ： 李 建 興
指導教授 ： 林 大 衛 博士

中華民國 九十三 年 七 月

在多數位訊號處理器系統上進行高效率無線通道模擬之研討

Research in Efficient Wireless Channel Simulation on Multi-DSP Platform

研 究 生: 李建興　　Student : Chien-Hsing Lee
指導教授: 林大衛 博士　　Advisor : Dr. David W. Lin

國 立 交 通 大 學

電機資訊學院 電子與光電學程

碩 士 論 文

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

中華民國 九十三 年 七 月

在多數位訊號處理器系統上進行高效率無線通道模擬之研討

研究生：李建興　　　　　　　　指導教授：林大衛 博士

國立交通大學電機資訊學院 電子與光電學程(研究所)碩士班

摘 要

數位訊號處理器是個可編程以完成達到不同功能性的有用工具。我們想要改進一個已實現於數位訊號處理器上的無線通道模擬器。其工作平台係使用桌上型電腦為主控中心，加插一塊 Innovative Integration 公司的 Quatro6x DSP 板，該 DSP 板共裝置四顆德州儀器公司出品的 TMS320C6x 數位訊號處理器。此模擬系統主要是利用其中三顆加以實現，分別為：一顆依據 3GPP WCDMA 上行傳輸之規格的調變器，一顆幾種不同通道的模擬器，及一顆接收濾波器。然而，有三個問題尚待釐清解決。第一，這三顆 DSP 程式在這個工作平臺上執行需按特定順序，難以同時啟動。第二，各訊號處理元件均可達到即時速度，但卻在其連接後的多處理器系統上變慢。第三，由於只有浮點格式的基本數學函數庫可用，通道係數尚未完全使用定點方式產生。在本篇論文中，我們改進這三個問題。首先，我們將工作平台更新到特定版本組合的一個新平台，解決了啟動的問題。然後，我們提出一個應用雙緩衝(double buffering)技巧的管線化(pipelining)架構，改善了速度的問題。此外，我們運用 CORDIC 演算法，以定點算術運算四個基本數學函數，避免了浮點的問題。總之，我們找出在多數位訊號處理器平台上順利進行高效率無線通道模擬的一些方法，然而，WCDMA 的詳盡探討並非本文所涉及。

Research in Efficient Wireless Channel Simulation on Multi-DSP Platform

Student: Chien-Hsing Lee Advisor: Dr. David W. Lin

Degree Program of Electronics Engineering Computer Science
National Chiao Tung University

Abstract

The DSP is a useful tool that is programmable to achieve different functionalities. We want to improve a previously developed DSP-based wireless channel simulator. The existing simulator includes a 3GPP WCDMA modulator, several kinds of channel simulator and a matched filter. A desktop PC acts as the controller, and an Innovative Integration's Quatro6x DSP-embedded card is employed in the system. Four Texas Instruments' TMS320C6x DSP chips are placed on the board. Three chips used to accomplish the system, one for modulator, one for channel, and one for the matched filter. However, three problems remained to be addressed. Firstly, synchronized execution of the three programs was not smooth on the platform. Secondly, real-time performance degraded on the connected multiprocessors DSP system. Thirdly, fixed-point generation of channel coefficients was not available yet for lack of fixed-point library. These three main topics are presented in this thesis. To begin, we do migration to a specified new platform for first problem solution. Then, we propose a pipelining structure applying double buffering scheme for second problem solution. Furthermore, we apply CORDIC algorithm to evaluate elementary functions in fixed-point for third problem solution. In a word, we seek out several methods to run an efficient wireless channel simulator smoothly on a multiprocessor platform, but a close study of WCDMA is not our concern.

誌謝

這篇論文的完成，首先，我要感謝我的指導教授林大衛博士這些日子以來的耐心教導。每當我的研究學習遇到瓶頸時，老師總是不厭其煩地指點迷津，在此致上最衷心的謝意與敬意。此外，也要感謝口試委員們於百忙之中撥冗，並於口試時給予寶貴的意見與建議。

我要感謝許多夥伴們，有了你們的包容與幫忙，我方能兼顧學業與工作。實驗室內充足的資源，使我的研究學習沒有後顧之憂。感謝諸位學長姊、同學與學弟妹們，在彼此互相的討論砥礪下，使我獲益良多。同時，也要感謝朋友與同事們，適時地關懷督促與伸出援手。另外，還要感謝 Innovative Integration、德州儀器、與盛暘科技等公司，適切地技術支援與答詢。

當然，我要獻上最深摯的感謝給我的父母與家人。你們在背後默默的支持與鼓勵，是我成長茁壯的陽光與土壤，更是我前進的動力，才使得我得以堅持到最後，順利完成心中的願望。

一路走來，點點滴滴，要感謝的如滿天星斗難以計數。感謝所有陪我走過、關心我的人，也感謝上蒼保佑。謹以這篇論文，與你們一同分享這份喜悅。

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Current generations of telecommunications infrastructure require real-time performance. To meet the demands of next generation equipment, such as third generation (3G) cellular mobile radio communication system, designers must seek methods to improve hardware and software efficiency. This thesis is intended as an investigation of the methods.

With the advent of applications such as 3G wireless system implementation that require intensive real-time computation, distributed processing systems consisting of several interconnected DSPs are a useful prototyping tool [38]. It comes within the scope of this thesis to run an efficient 3G wireless channel simulator smoothly on a multiprocessor platform, but a close study of 3G topic is not necessary for our present concern.

Fig. 1.1 gives a simple block diagram of the overall WCDMA system. A large number of studies has been made in a team project reported in [7], [11], [23], [24], and [32]. Four DSP boards have been used in the work, three fixed-point ones called Quatro62 and one floating-point one called Quatro67. The Quatro6x is made by Innovative Integration (II). It houses four Texas Instruments (TI)'s TMS320C6x01 processors in a symmetric multiprocessing relationship with interprocessor communication links. Basically, the implemented individual functional blocks on the DSP processors can reach real-time computation separately. However, the interprocessor data transmission was done in a block-based, event-driven fashion. Therefore, a more efficient implementation is desirable.

The components shaded in Fig. 1.1 belong to an implemented wireless channel simu-

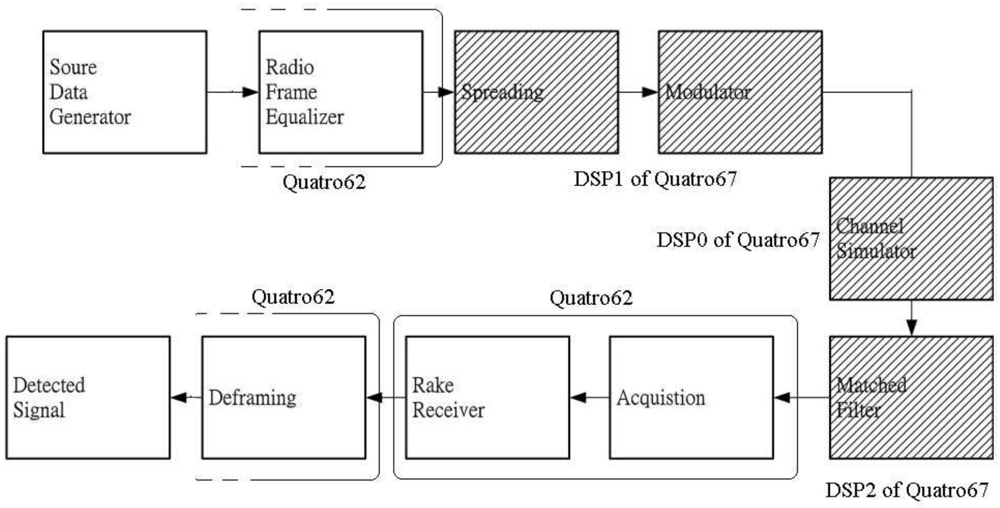


Fig. 1.1: Block diagram of WCDMA system for one-way transmission (from [32]).

lator on Quatro67 platform. Although a great deal of effort has been made in the implementation by Tsai [32], several issues remained to be addressed. Firstly, synchronizing execution of multiprocessor programs was not smooth on the platform. The individual DSP programs needed to be executed in a particular order, that is, they could not be loaded to the DSPs simultaneously. Secondly, real-time performance degraded on the connected multiprocessor DSP system. The actual run time increased unexpectedly after we connected the individual DSP programs together on the Quatro6x DSP board. It was an issue also faced by other project team members mentioned previously. Thirdly, fixed-point generation of channel coefficients was not available yet. The generation of channel coefficients still depended on floating-point mathematic functions. The three issues are addressed in this thesis.

This thesis is organized as follows. Chapter 2 reviews briefly the implemented wireless channel simulator. In chapter 3, we introduce the DSP environment and deal with synchronizing execution problem. Chapter 4 identifies efficient multiprocessing, including pipelining-512 structure, real speed observation, and double-buffering skill. We discuss elementary function evaluation in fixed-point arithmetic, such as CORDIC, in chapter 5. Finally, chapter 6 contains the conclusion and potential future work.

Chapter 2

Overview of An Existing DSP-Based Wireless Channel Simulator

2.1 Introduction to the Existing Simulator [32]

A DSP-based wireless channel simulator has been implemented by Tsai [32]. This implemented system employs a collaborative computing structure, which is composed of a desktop PC and a DSP-embedded plug-in card. There are four DSP chips on the DSP-embedded card, called DSP0 to DSP3. Three chips are used to accomplish the system. DSP0 acts as the channel simulator. The function of DSP1 includes spreading, scrambling coding and pulse-shaping filtering. The matched filter is in DSP2. Fig. 2.1 shows a block diagram of the modulator, channel simulator and receiver filter. The modulator is used to generate the transmitted signal. The data after framing operation (not shown in the figure) is input into the system and spread according to the 3GPP standard [3], which includes channelization coding, scrambling coding, and the pulse-shaping filtering. Then the data are passed through the channel, such as static and fading channels. Besides the multipath effect, which is defined in the 3GPP standard [1], multi-user interference also is considered. Finally, the noisy, distorted signal is received through the matched filter.

In the existing simulator [32], DSP processors are used to implement 3GPP WCDMA transmission signal processing and simulate the wireless channel for real-time experiments. However, we want to improve the existing simulator for several remained issues.

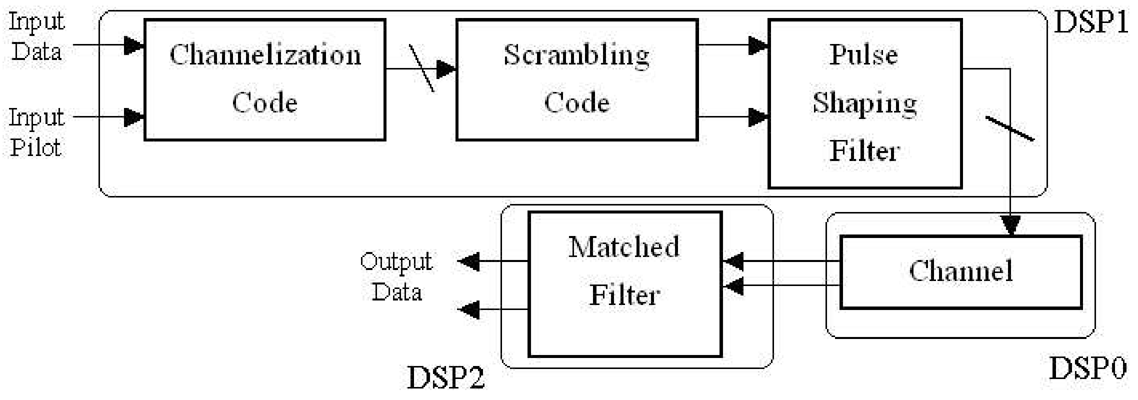


Fig. 2.1: Block diagram of the implemented system (from [32]).

2.2 WCDMA Uplink Transmission Scheme [32]

To begin, we first introduce the transport channels. The time durations are defined by start and stop instants, measured in integer multiples of chips. A radio frame is a processing duration which consists of 15 slots. The length of a radio frame corresponds to 38400 chips. A slot is a duration which consists of fields containing bits. The length of a slot corresponds to 2560 chips.

2.2.1 Spreading Modulator

The spreading modulator performs two operations. The first, called channelization, transforms every source data symbol into a number of chips, thus increasing the bandwidth of the signal. The second operation, called scrambling, distinguishes different users in the receiver.

Channelization codes

With the channelization, data symbols on I- and Q-branches are independently multiplied with an orthogonal variable spreading factor (OVSF) code. The cross-correlation between orthogonal codes is zero for synchronous transmission.

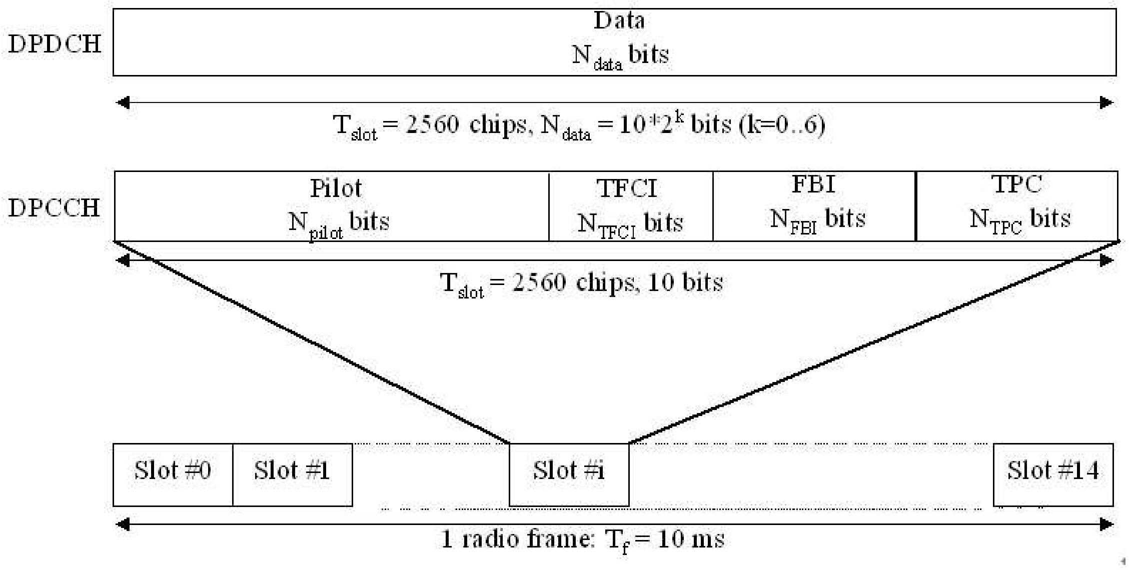


Fig. 2.2: Frame structure of uplink DPDCH/DPCCH (from [2]).

Scramble codes

After the channelization operation, the I- and Q-branch signals are multiplied by the complex-valued scrambling code. The code can be either a short or a long code. The required number of codes depends on the expected traffic load and spectrum efficiency. There are 2^{24} different uplink scrambling codes with different initial values for the generating registers. The long scrambling sequences $C_{long,1,n}$ and $C_{long,2,n}$ are constructed from sum of 38400 chips segments of two binary m-sequence generated by means of two generator polynomial of degree 25. Fig. 2.3 shows the configuration of uplink long scrambling sequence generators.

Control of SNR

To define the power level of input data, we have to compute the signal energy in the overall system and find how to adjust the power to achieve different SNR. The SNR at matched filter output for DPDCH is:

$$SNR_d \triangleq \frac{E_d}{\sigma_v^2} = \frac{4A^2 \cdot SF^2 \cdot \beta_d^2}{2 \cdot SF \cdot \sigma^2} = \frac{2A^2 \cdot SF \cdot \beta_d^2}{\sigma^2},$$

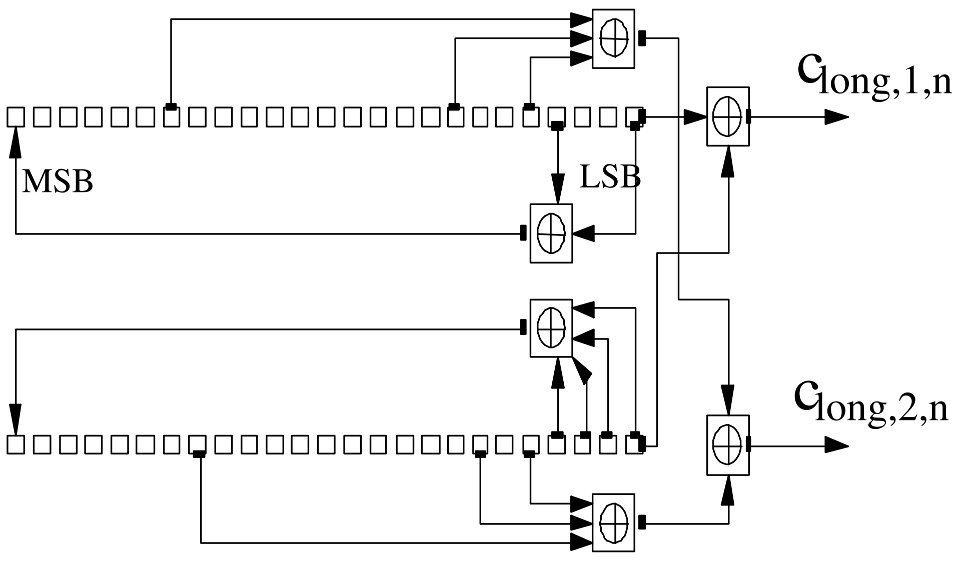


Fig. 2.3: Configuration of long scrambling sequence generator (from [3]).

where A is the amplitude of the transmitted signal, σ^2 is the noise variance on each quadrature branch at the input to the matched filter. For DPCCH, the SNR is:

$$SNR_c \triangleq \frac{E_c}{\sigma_v^2} \frac{4A^2 \cdot 256^2 \cdot \beta_d^2}{2 \cdot 256 \cdot \sigma^2} = \frac{2A^2 \cdot 256 \cdot \beta_c^2}{\sigma^2}.$$

Hence we can adjust the amplitude factor A of the receiver input signal to achieve any desired SNR.

2.2.2 Pulse Shaping Filter

Due to the requirement of bandwidth-limited transmission, the output chip stream from the spreading modulator is filtered using a pulse shaping filter (PSF). The 3GPP WCDMA employs root-raised-cosine (RRC) pulse shaping with roll-off $\alpha = 0.22$ [1]. It is found [32] that a 33-taps RRC filter can comply with the spectrum emission mask [1] to beyond 8 MHz. Fig. 2.4 is the comparison. Assume that the subsequent analog filtering can effectively suppress the signal power above 8 MHz.

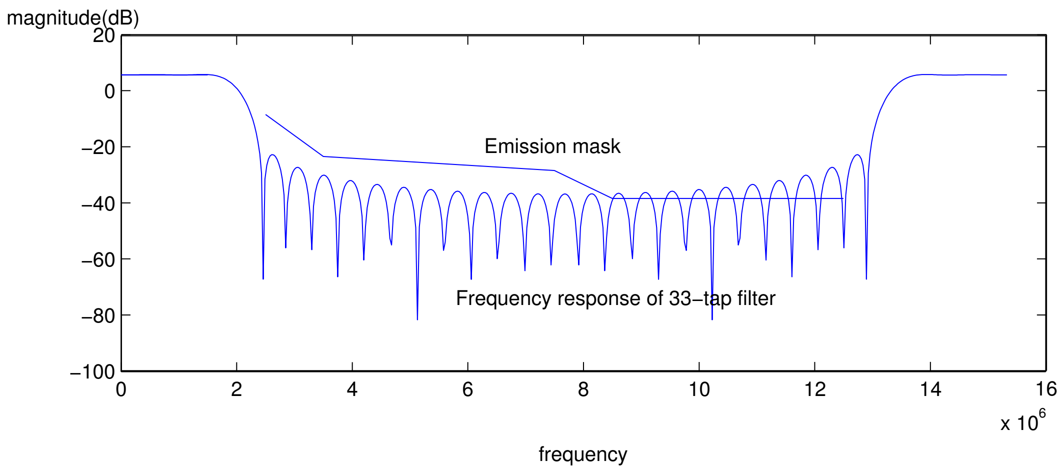


Fig. 2.4: Frequency response of 33-tap RRC filter (4 times oversampled) vs. the emission mask (from [32]).

At transmitter — the polyphase technique

In the transmitter, for reducing unnecessary computations, Tsai [32] consider a more efficient implementation to oversample the data and pass them through the PSF. Since only every Lth sample of the input data is nonzero, a better implementation, shown in Fig. 2.6, would involve applying filter coefficients only to input values that are nonzero [26].

To illustrate the advantage of Fig. 2.6 compared to Fig. 2.5, we note that if $H(z)$ is a filter of length N, then we then need NL multiplications and $(NL-1)$ additions per unit time originally. On the other hand, we only need $L(N/L)$ multiplications and $L(N/L-1)$ additions per unit time for the set of polyphase filters, plus $(L-1)$ additions to obtain an output datum. Thus we can obtain significant saving in computation. In DSP implementation, the rearrangement reduces the required run time and makes the system complete the signal processing in the limited amount of time [26], [32].

At receiver — the matched filter

At receiver side, a matched filter is designed to provide the maximum signal-to-noise power ratio at its output in AWGN. Thus the amount of computation is quite large. A

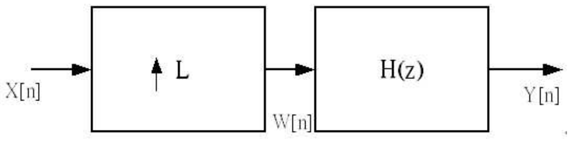


Fig. 2.5: Interpolated filtering system (from [26]).

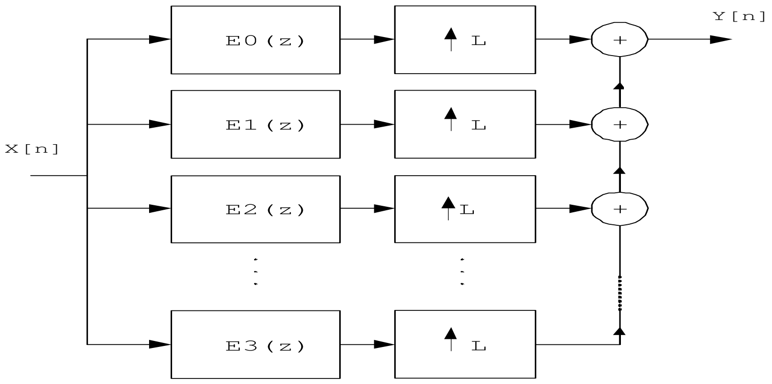


Fig. 2.6: Implementation of interpolated filter after applying the polyphase decomposition (from [26]).

9-tap RRC filter is chosen [32] to replace the original 33-tap one. We need to perform $15.36\text{M}\cdot 9\cdot 2 = 276.84\text{M}$ multiplications in one TI's TMS320C6701 chip, which is within the real-time computation ability of the chip.

2.2.3 Channel Model

A communication channel transmits the information-bearing signal to the destination. The signal is subject to multipath fading and addition of noise, which produce random variations in the signal. The block diagram of channel simulator is illustrated in Fig. 2.7, where only a single user is considered. The interference from other users is added to the result at the output of single user channel. In our simulation, Tsai [32] implement two kinds of channels, static and fading.

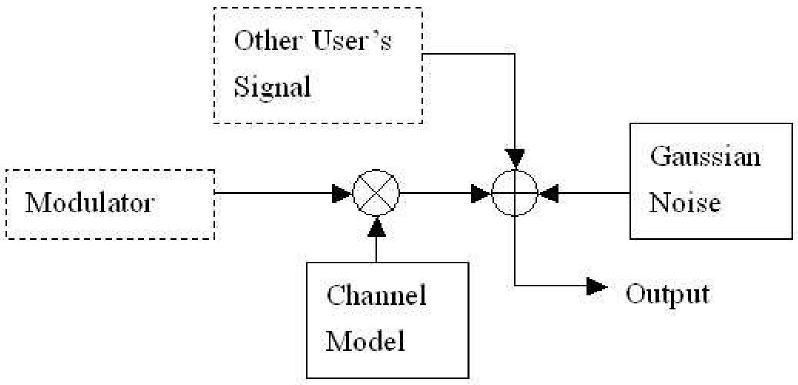


Fig. 2.7: Block diagram of channel model (from [32]).

Static channel

In this situation, multipaths exist, but the channel coefficients do not change during the transmission period. The channel coefficient of each path is a complex value given by $\alpha e^{j\theta}$, where α is the power level of the path and can be computed from the definition of SNR. $e^{j\theta} = \cos\theta + j\sin\theta$ is the phase of the path. In addition to static multipath propagation, white Gaussian noise is added.

Fading channel

Fading refers to rapid fluctuation of the amplitude of the channel gain over a short period. The power of each multipath is time varying, resulting from moving mobile or surrounding objects. To approximate Rayleigh fading, Jakes [13] suggests using phases $\theta_{n,j} = \beta_n + 2\pi(j-1)/(N_0+1)$, where $j = 1$ to N_0 is the waveform index. The model becomes

$$T(t) = \sqrt{\frac{2}{N_0}} \sum [\cos\beta_n + j\sin\beta_n] \cos(w_n t + \theta_n)$$

where the Doppler frequency w_m is given by $w_m = 2\pi\frac{v}{\lambda}$, with v being the velocity of the mobile and λ being the wavelength of the carrier. In our case, $f = 2$ GHz and $\lambda = 3 \times 10^8/2 \times 10^9 = \frac{3}{20}$ m, $w_n = w_m \cos\left(\frac{2\pi n}{N_0}\right)$, $n = 1, 2, \cdots, N_0$, and $\beta_n = \frac{\pi n}{N_0}$. Further discussion will be presented in chapter 5.

Table 2.1: Propagation Conditions for Multipath (Fading) Environments [1]

Case 1, speed 3km/h		Case 2,speed 3km/h		Case 3, 120 km/h		Case 4,speed 3 km/h	
Realative Delay [ns]	Average power [dB]	Realative Delay [ns]	Average power [dB]	Realative Delay [ns]	Average power [dB]	Realative Delay [ns]	Average power [dB]
0	0	0	0	0	0	0	0
976	-10	976	0	260	-3	976	0
		20000	0	521	-6		
				781	-9		

Multipath propagation

Some multipath propagation conditions are defined in [1]. Table 2.1 shows the propagation conditions that are used for the performance measurements in multipath environment. A chip in our program is $1/3.84 = 260$ ns. After oversampling by 4 times, each symbol is $260/4 = 65$ ns. The conversion of the delay listed in Table 2.1 is computed by

$$delay\_symbol = delay/65.$$

For 976 ns delay, we implement 15 points shift in our program. Except the defined multipath propagation listed in the Table 2.1, Tsai [32] also simulate other kinds of propagation channels including the birth-death and moving propagation conditions [1].

Gaussian noise

The signal transmitted through channel is added white Gaussian noise. Tsai [32] generate the noise by warping two uniform distribution random sequences. The operation is done on the PC and the noise is stored in files. After using Matlab functions to compare the power spectral density between the amount of the noise we store and use repeatedly and the real situation, we decide to store 298801 complex-valued noise data in files. Further discussion will be presented in chapter 5.

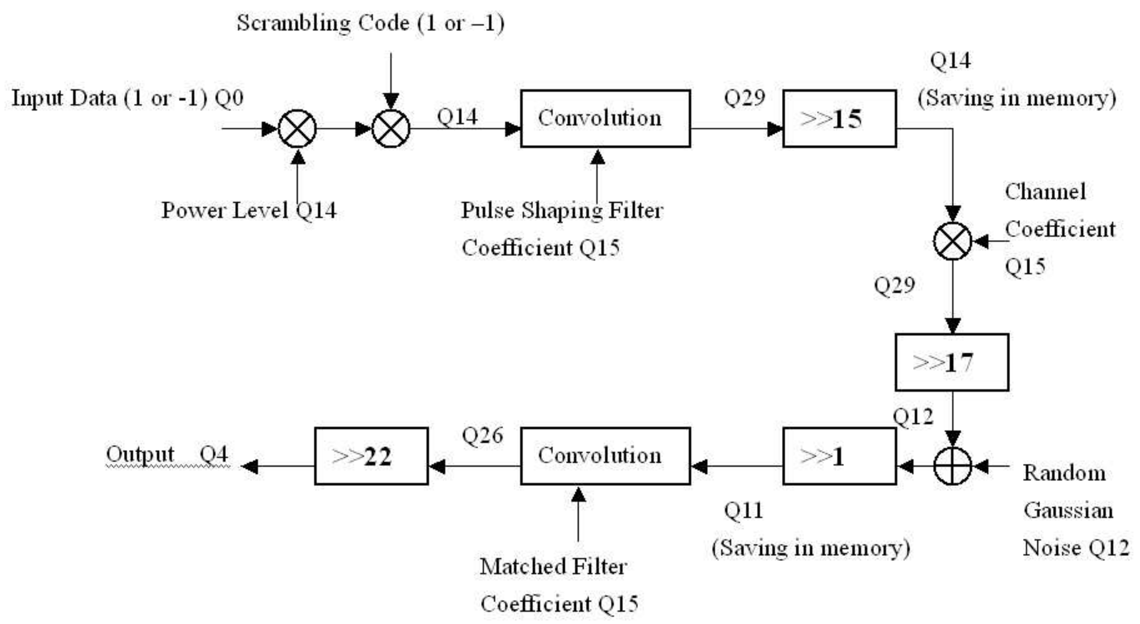


Fig. 2.8: The scaling situation of each step (from [32]).

2.2.4 Fixed-Point Simulation

For improving the speed and saving the memory, we have to simulate with fixed-point numbers instead of floating-point numbers. The mechanism we want is that the data saved in the internal memory in DSP board should be 16-bit integers. For maintaining the precision, the method is to shift each data to 16-bit integer. After addition or multiplication operations, Tsai [32] put them in a longer temporary register to avoid overflow. Tsai pay attention to the addition of each step during the simulation and make sure the scale on the two data are the same. The scale of the data in each step is illustrated in Fig. 2.8. The example in Fig. 2.8 is for static channel with $SNR = 10$ dB. When we save the data into memory, we have to measure the maximum value and downshift them to 16-bit integers. Further discussion will be presented in chapter 5.

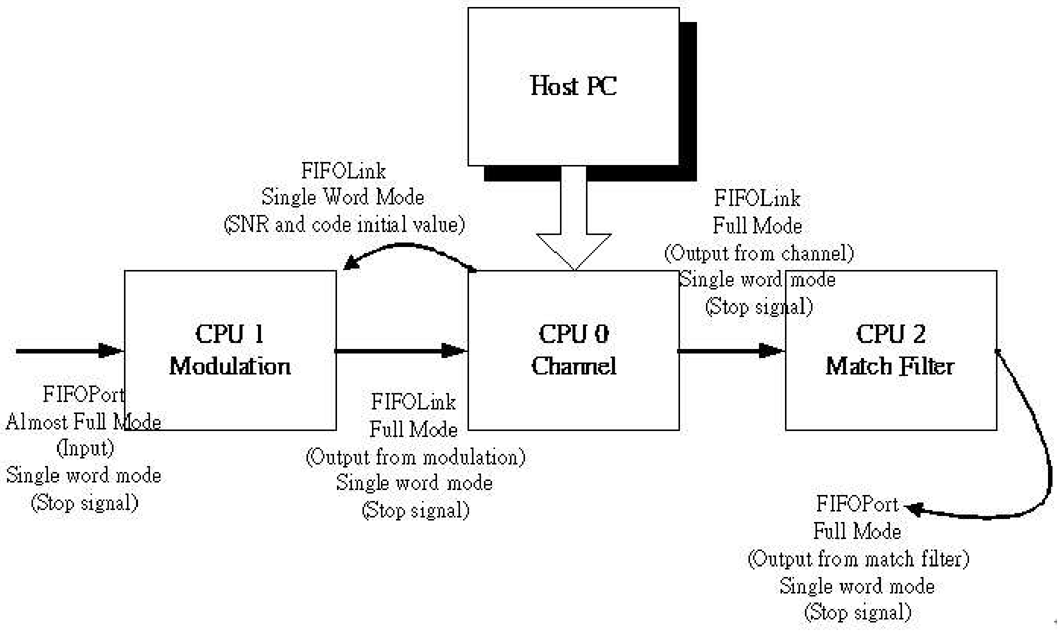


Fig. 2.9: Transmission mechanism in the existing simulator (from [32]).

2.3 DSP Implementation of the Existing Simulator [32]

A DSP-based simulator has been implemented by Tsai [32]. In this existing simulator, one floating-point Quatro67 DSP board is used to implement to the modulator, the wireless channel simulator and the receiver. This section shows briefly their key record.

2.3.1 Transmission Mechanism

Fig. 2.9 shows the transmission mechanism of the existing simulator [32]. The transmission between two chips or two boards is performed by a flexible FIFO-based interprocessor communications network. And if we want to process the data saved in external memory, we use DMA controller to move data rapidly. Further discussion will be presented in the following chapter.

Table 2.2: Memory Arrangement [32]

	CPU1 (Modulation)	CPU0 (Channel)	CPU2 (Matched Filter)
Internal Memory	56.19 Kbytes	63.46 Kbytes	43.28 Kbytes
External Memory	150 Kbytes (Scramble codes)	1167 Kbytes (Noise)	

Table 2.3: Profiles of Scrambling Operation for Different Versions of the Code [32]

Modified Version	Total Cycle	Computation Time (per frame)	Memory Usage
If-else statement	2210565	1.326×10^{-2} s	9.375 Kbytes
Direct multiplication and addition	117974	7.708×10^{-4} s	150 Kbytes

2.3.2 Memory Arrangement

The existing simulator [32] processes a slot of information each time. In each slot, after oversampling 4 times, the amount of data are 10240 chips of complex data. For the data saved as 16-bit integer, the memory size is 40.96 Kbytes. Table 2.2 shows the memory arrangement for each block in each CPU.

2.3.3 Optimization and Profile

Tsai [32] use software pipelining, loop unrolling, speculative execution replacement and loop partition to optimize the performance [43], [39]. The profiles of different transceiver functions are given in Tables 2.3, 2.4 and 2.5, respectively. The measurement tool used here is the profile, which is provided by TI's TMS320C6x Code Composer Studio (CCS).

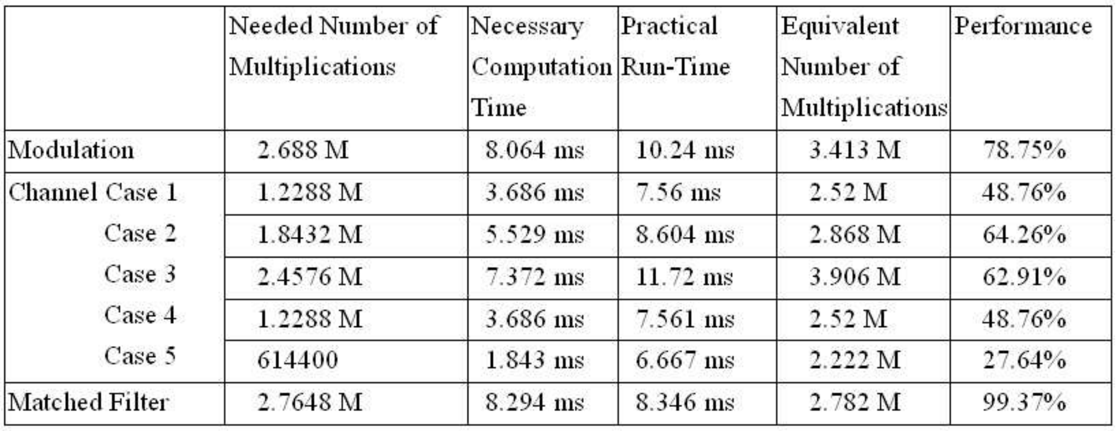
Table 2.4: Profiles of Pulse Shaping Filter Operation for Different Versions of the Code [32]

Modified Version	Total Cycle	Computation Time (per frame)
Original	10763220	6.458×10^{-2} s
Loop Unrolling	15727890	9.437×10^{-2} s
Loop Partition	1539691	9.238×10^{-3} s

Table 2.5: Profiles of Matched Filter Operation for Different Versions of the Code [32]

Modified Version	Total Cycle	Computation Time (per frame)
33-tap Filter	19340310	1.16×10^{-1} s
9-tap Filter	3032278	1.819×10^{-2} s
9-tap Filter (After data declaration)	1391295	8.348×10^{-3} s

Table 2.6: Complexity and Performance of Implementation [32]



	Needed Number of Multiplications	Necessary Computation Time	Practical Run-Time	Equivalent Number of Multiplications	Performance
Modulation	2.688 M	8.064 ms	10.24 ms	3.413 M	78.75%
Channel Case 1	1.2288 M	3.686 ms	7.56 ms	2.52 M	48.76%
Case 2	1.8432 M	5.529 ms	8.604 ms	2.868 M	64.26%
Case 3	2.4576 M	7.372 ms	11.72 ms	3.906 M	62.91%
Case 4	1.2288 M	3.686 ms	7.561 ms	2.52 M	48.76%
Case 5	614400	1.843 ms	6.667 ms	2.222 M	27.64%
Matched Filter	2.7648 M	8.294 ms	8.346 ms	2.782 M	99.37%

2.3.4 Complexity and Performance

When we analyze the complexity, we focus on the multiplications in our program. The amount of data we consider is 38400 chips, equal to a frame. It should be completed in 10 ms. The complexity and final performance are given in Table 2.6. The percentage figures listed in the table reflect the achievement from the effort spent in optimization by Tsai [32]. Two demo systems were constructed: the demo subsystem (shown in Fig. 2.10) and the multi-user system (not shown here).

2.4 Problems with the Existing Simulator

In summary, Tsai [32] implements a 3GPP WCDMA modulator, several kinds of channel simulators, and a matched filter on the Quatro67 multi-DSP card. For single user transmission under static channel with multipath propagation, the processing speed of

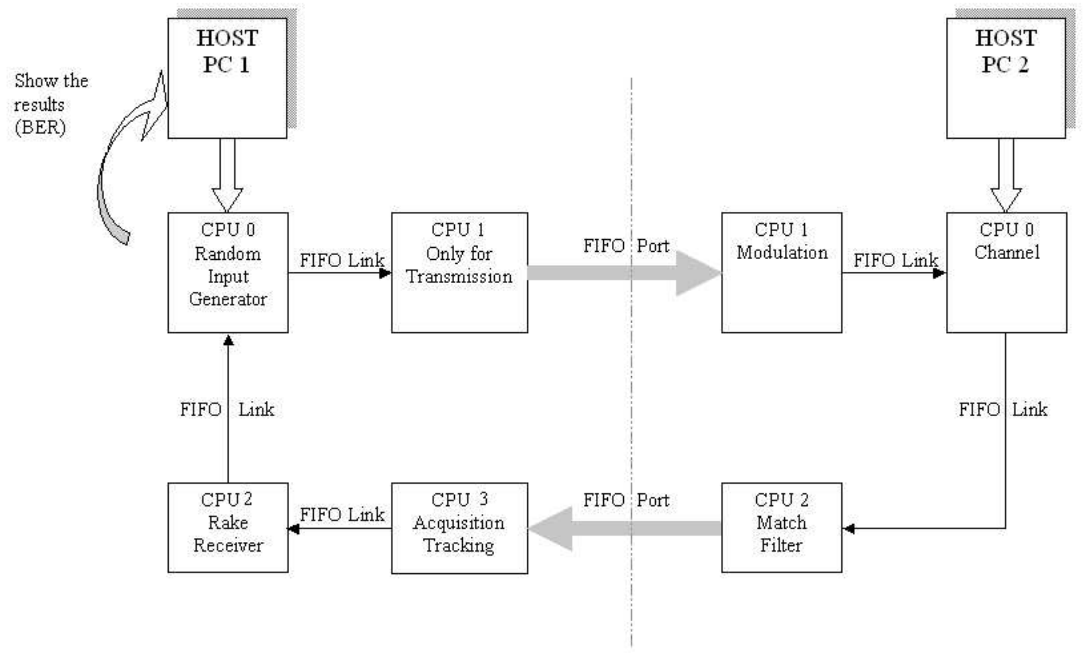


Fig. 2.10: Block diagram of the demo subsystem (from [32]).

the modulator and channel simulator can achieve the needed 3.84 Mchips per second. However, the existing simulator suffers from three problems. They will be improved in following three chapters respectively.

2.4.1 Synchronizing Execution Problem

In the existing simulator [32], three DSP chips are used to implement the modulator, the channel, and the receiver respectively. However, the three programs need to be executed in a particular order. That is, the system fails to download them simultaneously. We study the synchronizing problem in chapter 3.

2.4.2 Real-Time Multiprocessing Problem

A real-time DSP-based modulator, wireless channel simulator and receiver filter has been implemented [32]. For example, three individual parts channel, modulation, and matched filter can run at real-time in about 7.5, 10.2, and 8.3 ms per frame, respectively. We connect three main parts and download them to the Quatro6x DSP board. However, the actual run time is about 20 ms per frame [32]. But in 3GPP standard, each frame has length 10 ms. We study the real-time multiprocessing problem in chapter 4.

2.4.3 Fixed-Point Generation Problem

We use the Quatro67 DSP board to simulate the channel model. The TMS320C67x DSPs on the board do floating-point number operations. With the restriction of the DSP, the performance of the fading channel is not so good because the programs have to call special mathematic functions to generate the channel coefficients. There are several methods to change the generation of the channel coefficients without using special functions which need branching [32]. These methods are fixed-point methods. We study the problem in chapter 5.

Chapter 3

The Quatro6x Multiprocessor Platform

3.1 Overview

Fig. 3.1 shows the DSP board we use. It is an Innovative Integration (II)'s Quatro6x DSP board which houses four Texas Instruments (TI)'s TMS320C6x DSP chips. A host PC and several development tools work together with the board to provide a complete design environment. The development tools are TI's Code Composer Studio integrated development environment, JTAG emulator and II's Zuma toolset for the Quatro6x. In the existing simulator [32], three DSP chips are used to implement the modulator, the channel, and the receiver respectively. However, the three DSP programs need to execute in a certain specified order. Otherwise, troubles arise. Is it due to the platform or our application? Therefore, in this chapter, after introduction of the DSP environment, we are concerned with the interprocessor interaction mechanism, and the problem of synchronizing execution in the multiprocessor environment.

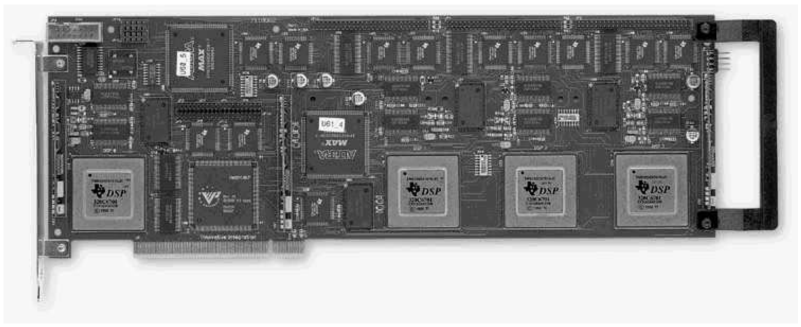


Fig. 3.1: Innovative Integration's Quatro6x DSP board (from [18]).

3.2 Introduction to the Multi-DSP Board

The DSP board we use is Innovative Integration (II)'s Quatro6x DSP board. It houses four Texas Instruments (TI)'s TMS320C6x01 DSP chips. The four chips may be fixed-point TMS320C6201 DSPs or floating-point TMS320C6701 DSPs. In the following, Quatro62 and Quatro67 refer to Quatro6x (or Q6x) that houses TMS320C6201 and TMS320C6701 DSPs, respectively, and Quatro6x (or Q6x) refer to either case. This section introduces the DSP chip and the DSP board.

3.2.1 TMS320C6x DSP Chip

Much description given in this subsection is taken from [41] and [42]). TI's TMS320C6701 is a 167 MHz floating-point DSP, and TMS320C6201 is a 200 MHz fixed-point DSP. Fig. 3.2 give their block diagrams. The TMS320C6x ('C6x) DSP processor consists of three main parts: the core, memory, and peripherals.

DSP core

The DSP core has two paths A and B in which processing occurs. Each data path has a register file containing sixteen 32-bit registers. Each path has four functional units to perform multiplication (.M), addition (.L), branching (.S) and load/store (.D). The functional units of each data path have a data bus to connect with the registers on the opposite side of the DSP core so that the units can exchange data.

Internal memory

The 'C6x has 64 Kbytes internal program memory and 64 Kbytes internal data memory. The program memory is 256 bits wide, having one fetch packet per line. The program memory can be configured as a program cache or a directed program memory. The 64 Kbytes of data memory of the 'C6x is organized into two blocks of 32 Kbytes: the TMS320C6701 have eight banks per block, and the TMS320C6201 have four banks per block.

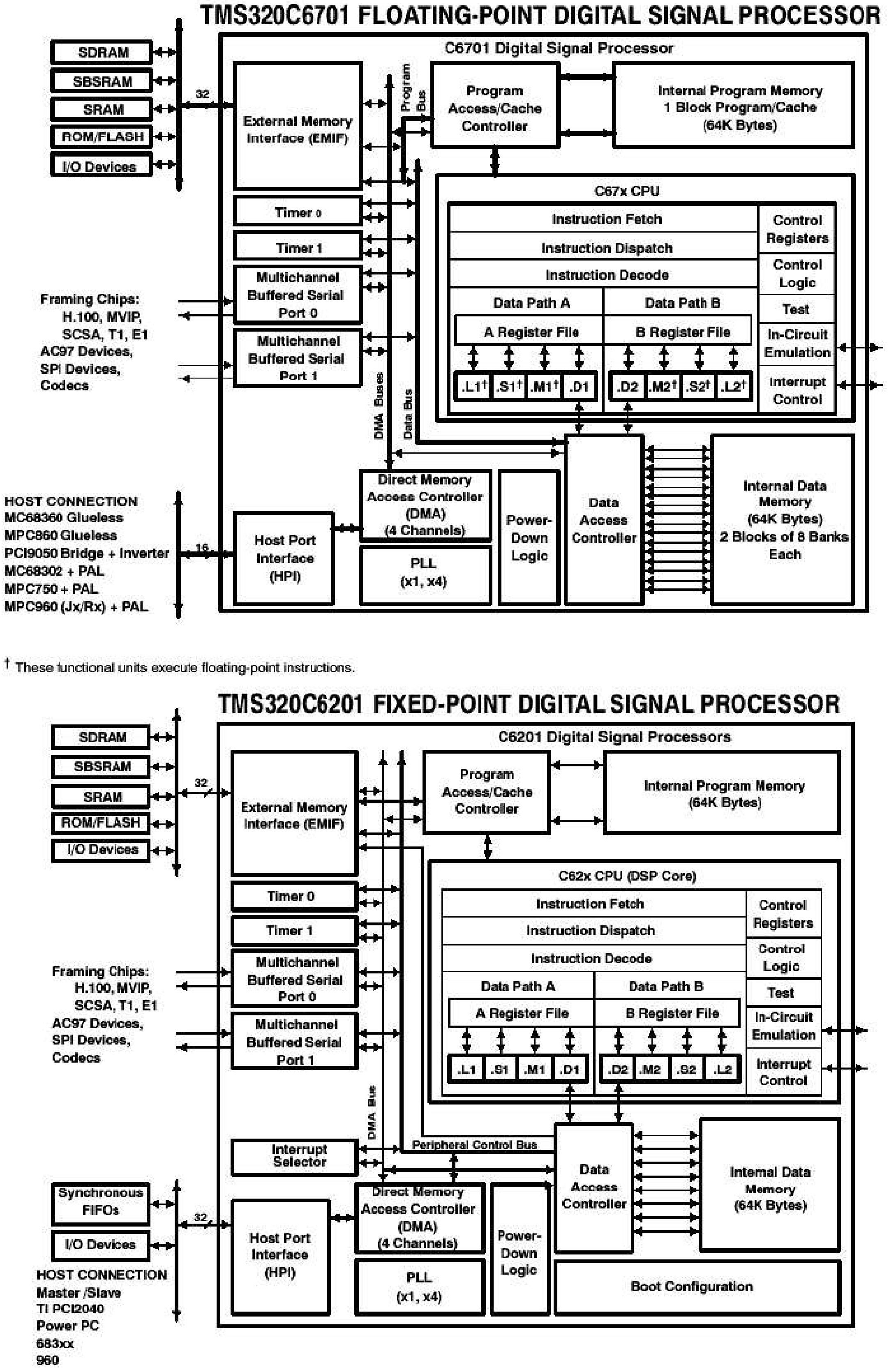

TMS320C6701 FLOATING-POINT DIGITAL SIGNAL PROCESSOR
C6701 Digital Signal Processor
SDRAM
SBSRAM
SRAM
ROM/FLASH
I/O Devices
32
External Memory Interface (EMIF)
Program Bus
Program Access/Cache Controller
Internal Program Memory 1 Block Program/Cache (64K Bytes)
Timer 0
Timer 1
Multichannel Buffered Serial Port 0
Multichannel Buffered Serial Port 1
Framing Chips: H.100, MVIP, SCSA, T1, E1 AC97 Devices, SPI Devices, Codecs
C67x CPU
Instruction Fetch
Instruction Dispatch
Instruction Decode
Data Path A
Data Path B
A Register File
B Register File
.L1† .S1† .M1† .D1
.D2 .M2† .S2† .L2†
Control Registers
Control Logic
Test
In-Circuit Emulation
Interrupt Control
DMA Buses
Data Bus
Direct Memory Access Controller (DMA) (4 Channels)
Power-Down Logic
Data Access Controller
Internal Data Memory (64K Bytes) 2 Blocks of 8 Banks Each
HOST CONNECTION MC68360 Glueless MPC860 Glueless PCI9050 Bridge + Inverter MC68302 + PAL MPC750 + PAL MPC960 (Jx/Rx) + PAL
16
Host Port Interface (HPI)
PLL (x1, x4)
† These functional units execute floating-point instructions.
TMS320C6201 FIXED-POINT DIGITAL SIGNAL PROCESSOR
C6201 Digital Signal Processors
SDRAM
SBSRAM
SRAM
ROM/FLASH
I/O Devices
32
External Memory Interface (EMIF)
Program Access/Cache Controller
Internal Program Memory (64K Bytes)
Timer 0
Timer 1
Multichannel Buffered Serial Port 0
Multichannel Buffered Serial Port 1
Framing Chips: H.100, MVIP, SCSA, T1, E1 AC97 Devices, SPI Devices, Codecs
C62x CPU (DSP Core)
Instruction Fetch
Instruction Dispatch
Instruction Decode
Data Path A
Data Path B
A Register File
B Register File
.L1 .S1 .M1 .D1
.D2 .M2 .S2 .L2
Control Registers
Control Logic
Test
In-Circuit Emulation
Interrupt Control
DMA Bus
Peripheral Control Bus
Interrupt Selector
Data Access Controller
Internal Data Memory (64K Bytes)
Synchronous FIFOs
I/O Devices
32
Host Port Interface (HPI)
Direct Memory Access Controller (DMA) (4 Channels)
PLL (x1, x4)
Power-Down Logic
Boot Configuration
HOST CONNECTION Master /Slave TI PCI2040 Power PC 683xx 960

Fig. 3.2: Block diagram of TMS320C6x01 DSP chip (from [33]).

Peripherals

The 'C6x chip contains several peripherals for communication with off-chip memory, coprocessors, host processors, and serial devices. Peripherals include a direct memory access (DMA) controller, power-down logic, external memory interface (EMIF), serial ports, expansion bus or host port, and timers. EMIF provides the interface for the DSP core to connect with several external devices, allowing additional data and program memory space. The DMA controller transfers data between regions in the memory map without passing through the DSP core. The DMA allows the movement of data at the internal memory, internal peripheral, or the external devices occurs in the background of the DSP core operation.

3.2.2 Quatro6x DSP Board

Much description given in this subsection is taken from [19] and [7]. Fig. 3.3 shows a block diagram of the Quatro6x board. Four DSP chips and memories are shown with connection to peripherals and other interfaces. The Quatro6x is a PCI bus compatible DSP board housing four Texas Instruments (TI)'s TMS320C6x ('C6x) DSP chips in a symmetric multiprocessing relationship with interprocessor communication links. The four chips are called DSP0 to DSP3 (CPU0 to CPU3 or DSP-A to DSP-D) anticlockwise. The Quatro6x's features include the following:

Six interprocessor FIFOLinks

The Quatro6x implements a high speed, flexible FIFO-based interprocessor communication network called FIFOLink. The FIFOLink network allows any on-board processor to transmit to and receive from any other processor on the card via a high-speed 32-bit wide FIFO buffered interface. Each of the six available links implements a 512 $\times$ 32 bidirectional buffer, and the maximum transfer rate reaches 160 Mbytes/sec on a 200 MHz Quatro6x board. Further discussion of FIFOLinks will be presented in the following

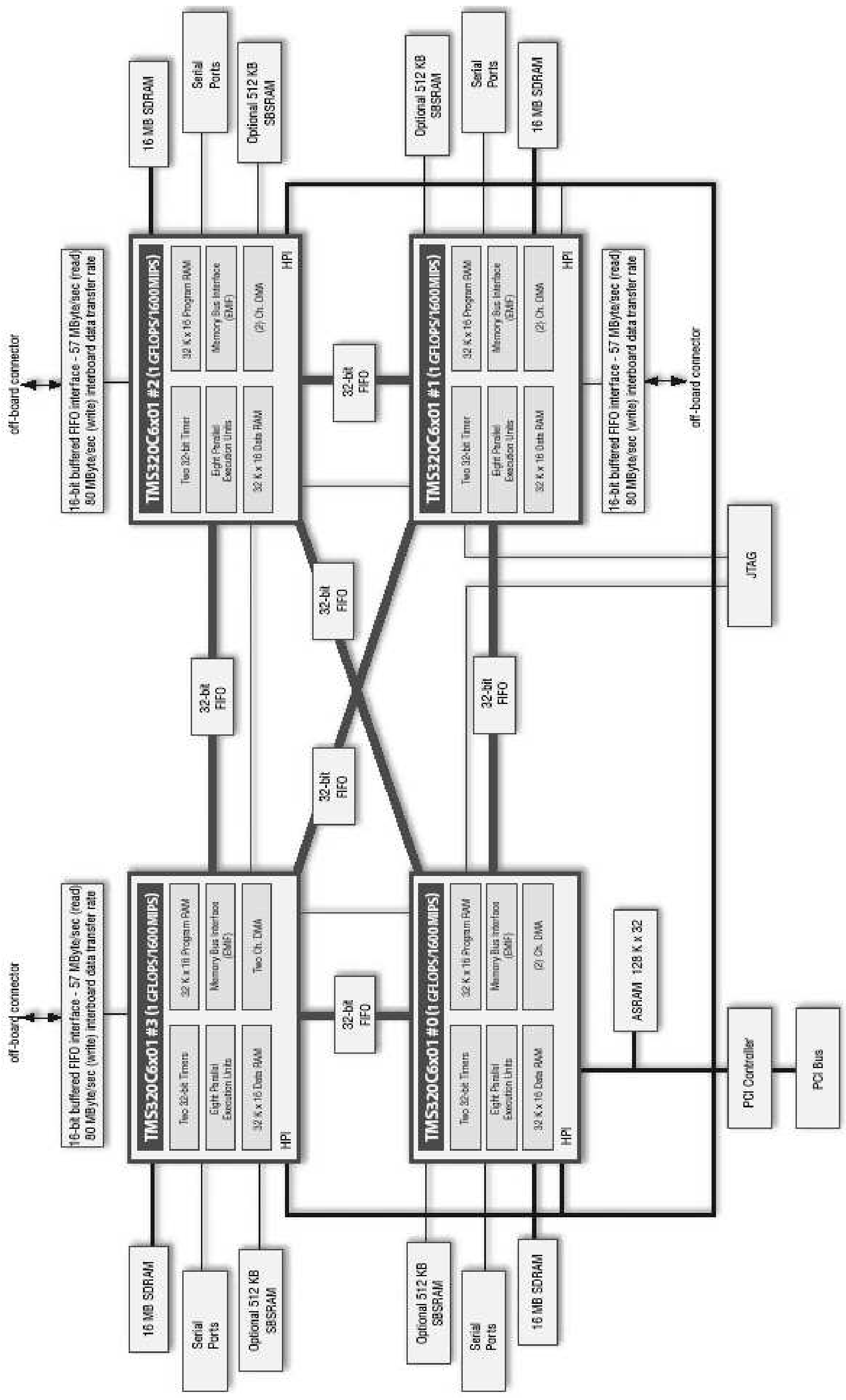

off-board connector
16-bit buffered FIFO interface - 57 MByte/sec (read) 80 MByte/sec (write) interboard data transfer rate
TMS320C6x01 #2 (1 GFLOPS/1600 MIPS)
Two 32-bit Timer
Eight Parallel Execution Units
32 K x 16 Data RAM
32 K x 16 Program RAM
Memory Bus Interface (EMIF)
(2) Ch. DMA
HPI
16 MB SDRAM
Serial Ports
Optional 512 KB SBSRAM
32-bit FIFO
TMS320C6x01 #1 (1 GFLOPS/1600 MIPS)
JTAG
TMS320C6x01 #3 (1 GFLOPS/1600 MIPS)
Two 32-bit Timers
Two Ch. DMA
TMS320C6x01 #0 (1 GFLOPS/1600 MIPS)
ASRAM 128 K x 32
PCI Controller
PCI Bus

Fig. 3.3: Innovative Integration's Quatro6x block diagram (from [18]).

chapter.

Three interboard FIFOPorts

The FIFOPort feature provides a means for interboard communications. It provide three bidirectional buffered 16-bit interfaces which allow external hardware or other II's DSP boards to communicate with the Quatro6x. A 512 x 16 FIFO is provided per FIFOPort, and up to 80 Mbytes/sec writing rate and 57 Mbytes/sec reading rate can be reached. Only DSP1, 2, and 3 support FIFOPorts, but DSP0 (first processor) has not. More details about applying FIFOPort can be found in [11].

PCI interface with ASRAM buffer memory

The Quatro6x provides a standard 32-bit PCI bus interface for communication between the PC and the DSP board. Only first processor (DSP0) can communicate directly with the host PC through PCI. The ASRAM (asynchronous SRAM) accessible by the PCI bus interface device. The 128K $\times$ 32 ASRAM is used as a buffer for busmaster and slave data movement on the PCI bus. Further understanding of PCI can be gained from [7].

External SBSRAM and SDRAM memory pools

Optional SBSRAM (synchronous burst SRAM) and 16 Mbytes SDRAM (synchronous DRAM) memories provide large areas to store data or program. The SBSRAM and SDRAM are not accessible by the PCI interface and are private to their associated processor. The processors allow 8-, 16-, and 32-bit wide data movement to and from off-chip SBSRAM and SDRAM memory.

3.3 Introduction to the Development Tools

Much description given in this section is taken from [19]. The development tools of the Quatro6x are Code Composer Studio integrated development environment, JTAG emulator and Zuma toolset for the Quatro6x.

3.3.1 Code Composer Studio

TI's Code Composer Studio (CCS) is an integrated development environment to provide editing, compiling, downloading and low level debugging.

When used in conjunction with II's JTAG emulator, CCS allows to access specific DSP registers and functions. The PCI-style JTAG debugger is a separate card connecting with Quatro6x DSP board via cable. Using the JTAG-based, hardware-assisted C/Assembler source debugger, typical application programs will consist of one or more C (.c), header (.h), and assembly language (.asm) source files, as needed. Additionally, target program generation requires use of a linker command file (.cmd) which specifies the memory map (.map) for the target and optionally includes commands defining the libraries to be linked into the final application.

The linker command file serves three main objectives. The first objective is to describe to the linker the memory map of the system to be used, and this is specified by "MEMORY{...}". The second objective is to tell the linker how to bind each section of the program to a specific section as defined by the "MEMORY{...}" area, which is specified in the "SECTION{...}". The third objective is to supply the linker with the input and output files, and options of the linker.

3.3.2 Zuma Software Toolset

The Zuma toolset is a comprehensive collection of tools and libraries used to develop application programs for several series of II's DSP boards, which includes:

1. DSP Peripheral Library – supporting on-board peripheral and DSP functions.
2. Dynamic Link Library (DLL) – for host PC software application development.
3. Host Support Applets – for terminal emulation and automatic program download.
4. Sample Applications – showing Host PC as well as target DSP coding techniques.

UniTerminal and MPO are II's support applets for terminal emulation and automatic program download.

UniTerminal

Each of the development packages is supplied with a terminal emulator application called "UniTerminal," which can be used either stand-alone or in conjunction with CCS. If we invoke the UniTerminal utility and it is successfully started, then UniTerminal will display "Status: Active. DSP DLL Loaded OK" at the bottom of its client window.

The terminal emulator UniTermimal provides a C language-compatible, standard I/O terminal emulation facility for interacting with the stdio library running on an Innovative Integration target DSP processor. The DSP program execution will be halted automatically at the first stdio library call if the terminal emulator is not executing when the DSP application is run, since standard I/O uses hardware handshaking.

The UniTerminal supports downloading of either COFF (Common Object File Format) files (.OUT) or multiprocessor out (.MPO) files. The .MPO file provides a means of downloading separate .OUT files to multiple processors simultaneously, which greatly simplifies the task of synchronizing execution in a multiprocessor environment.

MPO

The MPO editor provides a means of editing the special configuration files used on the Quatro6x to allow downloading of multiple COFF object files simultaneously. The UniTerminal applets understand the MPO file format and are able to consume .MPO files as well as .OUT files as download arguments. Attempting to download an MPO file from within UniTerminal will cause new code to be loaded onto and executed by all processors. This is in contrast to the downloading a standard COFF .OUT file, which simply downloads and executes code on DSP0 (first processor) only.

TEST example program

One may refer to the target board directory for example programs provided with II's development package for examples of the use of the Quatro6x. These programs are provided as models for custom user software, and II highly recommends that the user examine these examples before beginning a first development effort for the target DSP. Full source code is provided for user inspection and reuse in modified or customer application.

TEST and TEST2 are board level hardware test programs, capable of exercising the major peripherals on the Quatro6x to double-check proper hardware functionality. As such, it contains routines for exercising each of the peripherals on the Quatro6x, including:

1. FIFOLinks,
2. FIFOPorts,
3. Internal timers,
4. Sync serial ports,
5. Busmastering, and
6. Interprocessor interrupts.

The programs aim to be encompassing in that they try to test as much of the board-level functionality as possible. The code included for TEST and TEST2 are broken down into functional pieces which are called separately for each subsystem to be tested, it is possible to factor out individual tests for use in other programs.

For example, the Quatro6x implements an interprocessor interrupt generation architecture which allows any one processor to notify any other processor of an event or condition via an interrupt. The 'C6x has two 32-bit general-purpose timers that is used to time events, count events, generate pulses, interrupt the CPU and send synchronization events

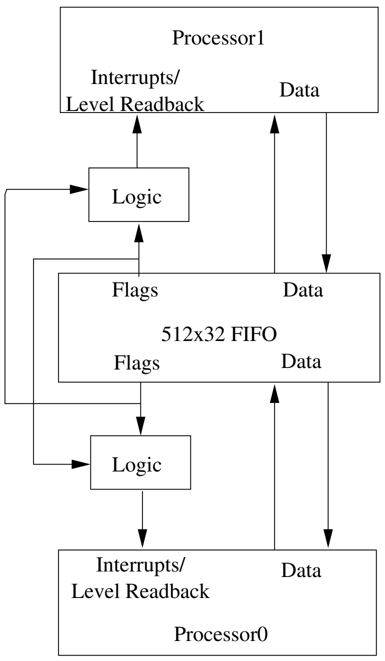


Fig. 3.4: FIFOLink block diagram (from [19]).

to the DMA controller. If we want to apply interprocessor interrupt based on internal timer, it is one of ways to factor out them from TEST and TEST2.

3.4 Interprocessor Interaction

Much description given in this section is taken from [19], [11], and [32]. On the Quatro6x DSP board, each of the four DSP processors has FIFOLink connected to another onboard processor. The FIFOLinks are compatible with the DMA controller for high-performance interprocessor data flow. Both FIFOPort and FIFOLink have several modes that can be used: single word, full words by DMA and almost full mode. In this section, we introduce the way to use FIFOLink and DMA functions for communication between two processors.

3.4.1 FIFOLink Functions

Fig. 3.4 shows the details of a single FIFOLink interface connection and its attendant control and status signals. Each FIFOLink includes a 512-element $\times$ 32-bit bidirectional buffer with full level and interrupt control on data transmission and reception. In this subsection, we describe some important functions used in FIFOLink. Before using FIFO (both the FIFOLink and the FIFOPort), there are some important things to do:

- Include the header file *"periph.h."*
- Declare the variables used by FIFO as global variables.

FIFOLink reset

The receive FIFO may be cleared and its condition reset at any time by using the function:

$$reset\_fifo\_link(cpu),\ cpu \in 0,\ 1,\ 2,\ 3.$$

FIFOLink status

The current fullness of a given link may be determined by reading the status port. The low-order six bits of the status port shows the status of *Full*, *Empty*, and *Almost Full* from each device. The FIFO status is defined in Table 3.1. We can use the following function to get the status:

$$get\_fifo\_link\_status(cpu),\ cpu \in 0,\ 1,\ 2,\ 3.$$

FIFOLink data transfer functions using CPU

Data may be moved between memory blocks and each of the FIFOLinks using the functions listed in Table 3.2. These routines are coded as inline functions for speed. The address of FIFOLinks is defined as Periph->FLink[fifo_link(cpu)]. For example, when we want to transmit a single word "a" to other CPUs through FIFOLink buffer, the used instruction as follows,

Periph->FLink[fifo_link(cpu)]=a; // cpu ∈ 0, 1, 2, 3.

It's same as fifo_link_split(cpu,a). If we check FIFOLink status before the transfer, that is,

while(!(get_fifo_link_status(cpu)&Tx_FIFO_EMPTY);

Periph->FLink[fifo_link(cpu)]=a;

Table 3.1: FIFO Status Definition [19]

C #define	Bits #	Condition
Rx_FIFO_FULL	0	Receive FIFO contains 512 elements
Rx_FIFO_EMPTY	1	Receive FIFO contains 0 elements
Rx_FIFO_AF	2	Receive FIFO contains more elements than pro-programmed threshold
Tx_FIFO_FULL	3	Transmit FIFO contains 512 elements
Tx_FIFO_EMPTY	4	Transmit FIFO contains 0 elements
Tx_FIFO_AF	5	Transmit FIFO contains more elements than pro-programmed threshold

Table 3.2: FIFOLink Data Transfer Functions Using CPU [19]

C Function	Description
fifo_link_spit(cpu,a)	Write a single word to the transmit FIFO using CPU without handshaking
fifo_link_emit(cpu,a)	Write a single word to the transmit FIFO using CPU with handshaking
fifo_link_eat(cpu)	Read a single word from the receive FIFO using CPU without handshaking
fifo_link_key(cpu)	Read a single word from the receive FIFO using CPU with handshaking
fill_fifo_link()	Write up to 512 elements from a memory buffer into the transmit FIFO using CPU
bleed_fifo_link()	Read up to 512 elements from receive FIFO into memory buffer using CPU

This is single word transfer with handshaking, and same as fifo_link_emit(cpu,a). Similar examples can be found in II recommended example program, TEST and TEST2.

3.4.2 DMA Transfer Functions

The DMA controller transfers data between regions in the memory map without intervention by the CPU. It has four independent programmable channels, allowing four different contexts for DMA operation. The DMA channels may be used to transfer data between any of the FIFOLinks and a memory buffer using the inline functions in Table 3.3. The call sequences are:

dma_mem_to_port(int channel, int src, int* dest, int count, int block),*

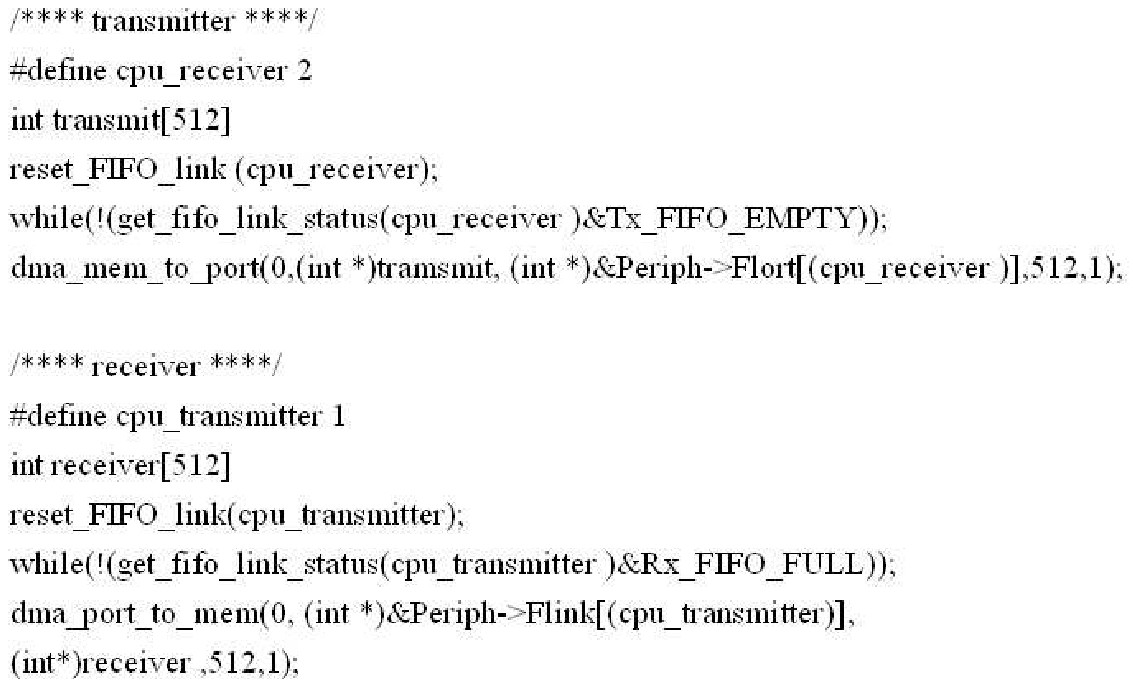
dma_port_to_mem(int channel, int src, int* dest, int count, int block),*

dma_copy_mem(int channel, int src, int* dest, int count, int block).*

For instance, the function dma_copy_mem(int channel, int* src, int* dest, int count, int block) copies “count” words of memory from the source buffer “src” to the destination

Table 3.3: DMA Data Transfer Functions [19]

C Function	**Description**
dma_port_to_mem()	Read up to 65536 words from a FIFO at indicated address into a memory buffer using specified DMA channel
dma_mem_to_port()	Write up to 65536 words from a memory buffer into a FIFO using specified DMA channel
dma_copy_mem()	Copies up to 65536 words between internal memory and external memory using specified DMA channel channel



```
/**** transmitter ****/
#define cpu_receiver 2
int transmit[512]
reset_FIFO_link (cpu_receiver);
while(!(get_fifo_link_status(cpu_receiver )&Tx_FIFO_EMPTY));
dma_mem_to_port(0,(int *)tramsmit, (int *)&Periph->Flort[(cpu_receiver )],512,1);

/**** receiver ****/
#define cpu_transmitter 1
int receiver[512]
reset_FIFO_link(cpu_transmitter);
while(!(get_fifo_link_status(cpu_transmitter )&Rx_FIFO_FULL));
dma_port_to_mem(0, (int *)&Periph->Flink[(cpu_transmitter)],
(int*)receiver ,512,1);
```

Fig. 3.5: Code for using DMA through FIFOLink (from [32]).

buffer “dest.” This function utilizes the specified DMA channel to perform the move. If “block” is true, the function waits until the move is completed before processing; otherwise execution continues immediately after the DMA operation starts. We give an example of using DMA through FIFOLink buffer with full level in Fig. 3.5. More details will be discussed in chapter 4.

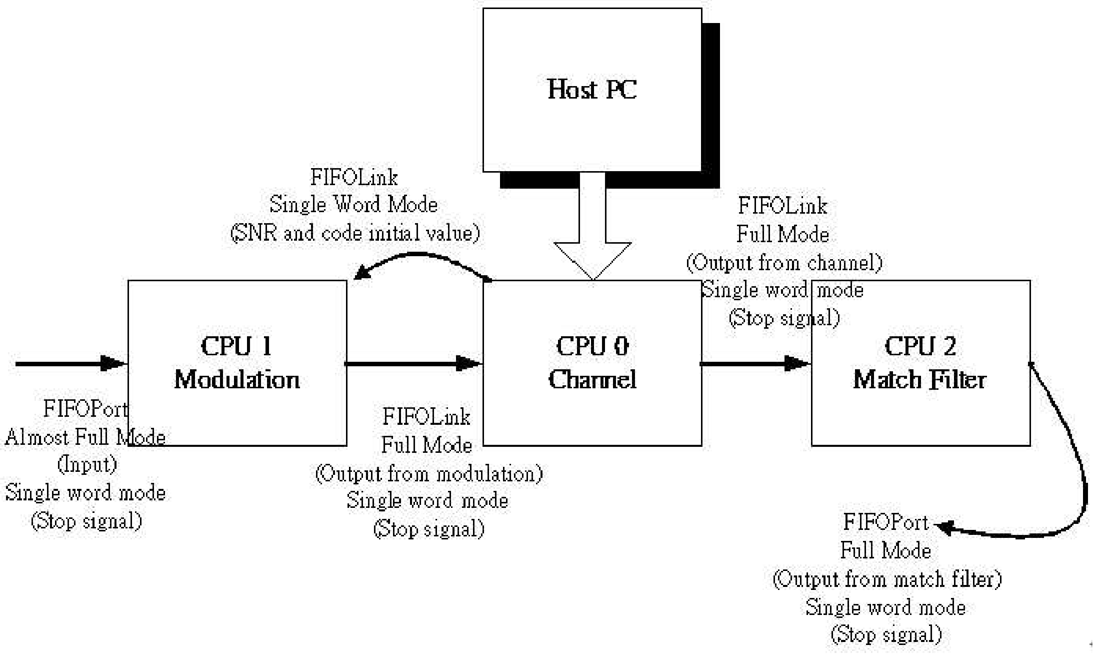


Fig. 3.6: Transmission mechanism in the existing simulator (from [32]).

3.4.3 Transmission Mechanism [32]

Fig. 3.6 shows the transmission mechanism of the existing simulator [32]. When we initialize the DSP, the SNR, initial value of scrambling codes and channel case will be set in the controller PC and downloaded to CPU0. We use FIFOLink single mode transmission to transmit the information to the modulator in CPU1. The input to modulator from the last board is received using FIFOPort almost full mode. We pass the output to the next processor after finishing one slot. The FIFOLink can transmit 512×32 bits of integer per time in full condition. During the process, a stop signal will be transmitted through all processors. The stop command is given by the first processor or the host PC. We use single word mode to pass the signal in FIFO.

Table 3.4: Unlucky Style

	noise.h & channel.c	data.h & modulation.c	CCS	MPO

1	short noisereal[298801]={...};
	short noiseim[298801]={...};
	j=(slot*10240+...)%298801;
		unsigned short input[330]={...};
		index=count*10;
		j=(input[330+count]...;
			Can
			Run
			Order
				Can
				Not
				Begin

2	short noisereal[11]={...};
	short noiseim[11]={...};
	j=(slot*10240+...)%11;
		unsigned short input[330]={...};
		index=count*10;
		j=(input[330+count]...;
			Can
			Run
				Can
				Not
				Finish

3.5 Coding Style

In this section, for multiprocessor programs synchronizing execution smoothly in CCS and MPO, we compare two coding styles, which is sensitive to array declaration.

3.5.1 Unlucky Style

In the existing simulator, we found an unlucky coding style as shown in Table 3.4. This unlucky style is to declare array of many elements with assigned value. It exists in two header (.h) files, one is noise.h and another is data.h. As introduced previously, the existing simulator stores 298801 complex-valued noise data in the noise.h file. In addition, there are 330 elements of data stored in data.h for modulation code. However, it is unlucky to run MPO smoothly or CCS without order.

3.5.2 Lucky Style

As shown in Table 3.5, we found a lucky style, which is to declare array of many elements without assigned value , or array of less elements with assigned value. It is lucky to run MPO smoothly. However, it maybe changes largely the implementation of the existing simulator. In next section, we find out a solution without changing coding.

Table 3.5: Lucky Style

	noise.h & channel.c	data.h & modulation.c	CCS	MPO

3	short noisereal[11]={...};
	short noiseim[11]={...};
	j=(slot*10240+...)%11;
		unsigned short input[33]={...};
		index=count*1;
		j=(input[3+count]...;
			Can Run	Can Run

4	short noisereal[298801];
	short noiseim[298801];
	j=(slot*10240+...)%298801;
		unsigned short input[330];
		index=count*10;
		j=(input[330+count]...;
			Can Run	Can Run

3.6 Tools Compatibility

Besides the coding style in previous section, we get a solution in this section ending for synchronizing execution. Before this, we are suffering from tools compatibility.

3.6.1 Two Issues in MPO and CCS

There are two issues related to Quatro6x platform. One is for MPO, another for CCS.

Fig. 3.7 shows the issue 1 in MPO. Its .OUT or .MPO file could not run by UniTerminal only, it seems sensitive to 2 factors: one is .cinit to DRAM or SDRAM, another is array size less or more than 500.

Fig. 3.8 shows the issue 2 in CCS. It repeated “TP>> internal error: bad type: TYPE::type_qualified()” during compile phase in newer version environment CCS2.0. However, the internal error exist, but zero error during compiling phase.

We email the two issues to vendors. However, vendor II suggests us migrate platform to new version. Refer to the appendix for details.

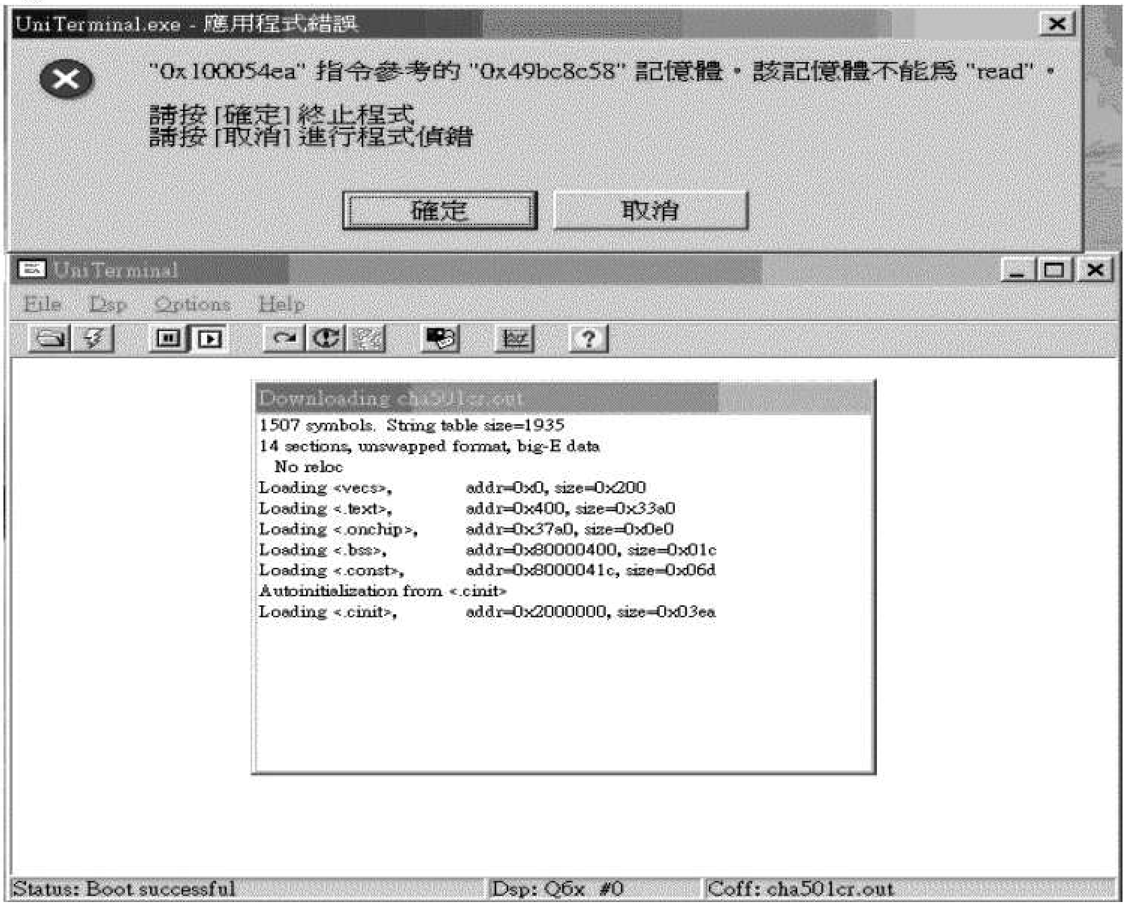
3.6.2 Migration to the New Version

As discussed previously, our application still encountered the two issues on the old version platform. When we tried to migrate to new version CCS2.2, or several different combinations of TI and II toolsets, many additional compile errors or link errors often happened. It also happened for the vendor’s example program of board-level TEST, and two issues related to T.I Q6x DSP

Remark1: one of simple source code to demo issue1:

```
#include <stdio.h>
short a1[501]= { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                 ……
                 100, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                 ……
                 90,91,92,93,94,95,96,97,98,499,500};
void main(){
    printf("\nCPU_A main begin");
    printf("\na1[0]:%d",a1[0]);
    printf("\na1[500]:%d",a1[500]);
    printf("\nCPU_A main end");
}
```

Fig of issue1:



"0x100054ea" 指令參考的 "0x49bc8c58" 記憶體。該記憶體不能為 "read"。

請按 [確定] 終止程式
請按 [取消] 進行程式偵錯

```
1507 symbols.  String table size=1935
14 sections, unswapped format, big-E data
  No reloc
Loading <vecs>,          addr=0x0, size=0x200
Loading <.text>,         addr=0x400, size=0x33a0
Loading <.onchip>,       addr=0x37a0, size=0x0e0
Loading <.bss>,          addr=0x80000400, size=0x01c
Loading <.const>,        addr=0x8000041c, size=0x06d
Autoinitialization from <.cinit>
Loading <.cinit>,        addr=0x2000000, size=0x03ea
```

Status: Boot successful | Dsp: Q6x #0 | Coff: cha501cr.out

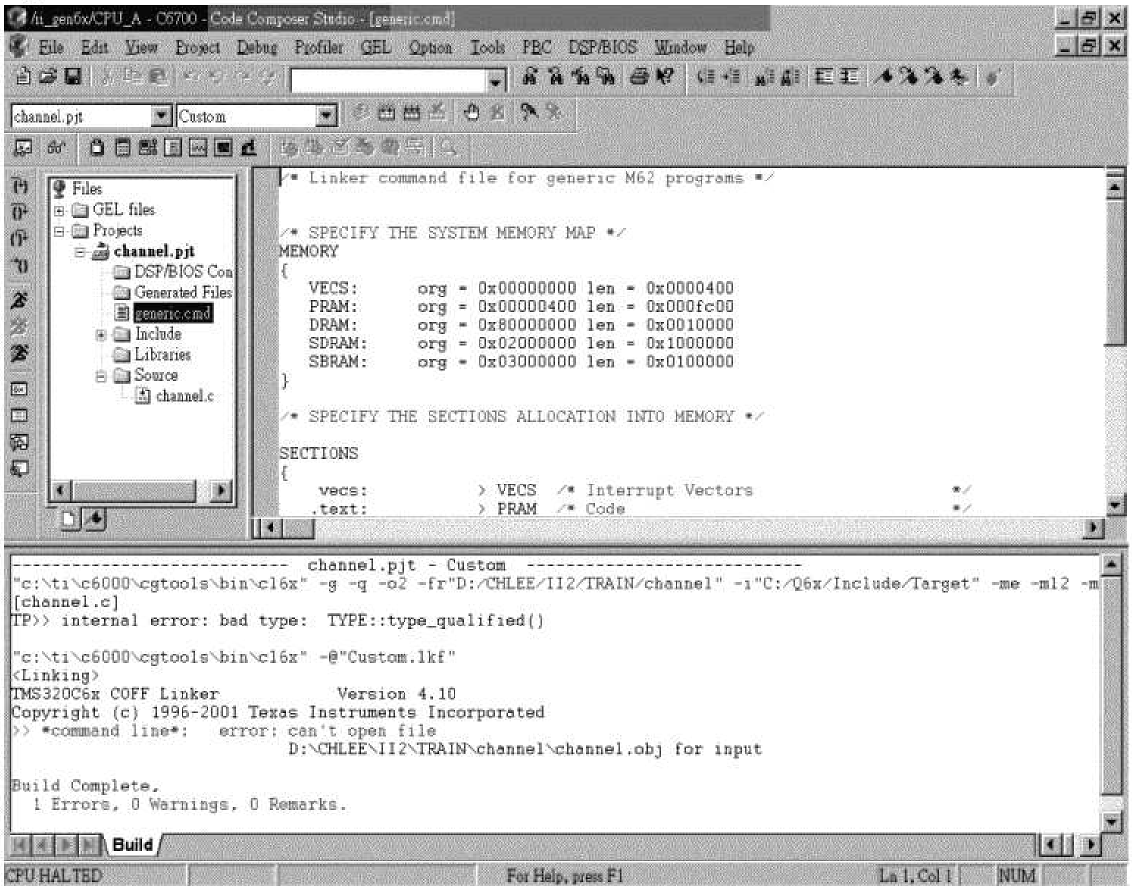
Fig. 3.7: A simple code suffering from issue 1 in MPO.

two issues related to I.I Q6x DSP

Remark2: one of simple source code to demo issue2:

```
#include "periph.h"
short c_multy_real;
void main(){
        short c=0;
        while(!(get_fifo_link_status(2)&Tx_FIFO_EMPTY));
        c_multy_real=c;
}
```

Fig of issue2:



```
/* Linker command file for generic M62 programs */


/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    VECS:      org = 0x00000000 len = 0x0000400
    PRAM:      org = 0x00000400 len = 0x000fc00
    DRAM:      org = 0x80000000 len = 0x0010000
    SDRAM:     org = 0x02000000 len = 0x1000000
    SBRAM:     org = 0x03000000 len = 0x0100000
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
     vecs:            > VECS  /* Interrupt Vectors                    */
    .text:            > PRAM  /* Code                                 */
```

```
----------------------------  channel.pjt - Custom  ----------------------------
"c:\ti\c6000\cgtools\bin\cl6x" -g -q -o2 -fr"D:/CHLEE/II2/TRAIN/channel" -i"C:/Q6x/Include/Target" -me -ml2 -m
[channel.c]
TP>> internal error: bad type:  TYPE::type_qualified()

"c:\ti\c6000\cgtools\bin\cl6x" -@"Custom.lkf"
<Linking>
TMS320C6x COFF Linker           Version 4.10
Copyright (c) 1996-2001 Texas Instruments Incorporated
>> *command line*:   error: can't open file
                            D:\CHLEE\II2\TRAIN\channel\channel.obj for input

Build Complete,
  1 Errors, 0 Warnings, 0 Remarks.
```

Fig. 3.8: A simple code suffering from issue 2 in CCS.

even for a very simple code such as summation from 1 to 100.

The above problem is sensitive to version combinations. After trying various combinations, we got a trouble-free combination without errors as shown above. The combination is II 2.97 installation CD and TI's CCS2.1 with patch from CCS2.0. On this platform, the program can run smoothly through CCS or MPO, that is, our multi-DSP application can run smoothly without a specifying the startup order through CCS, and the task of synchronizing execution through MPO can also run smoothly.

Therefore, we do migration to the specified platform without issues of tool compatibility. The memory amount may be not large enough to allocate both platform and application. Hence one should take care of memory allocation in linker command file to avoid the problem that memory can not be allocated.

Chapter 4

Efficient Multiprocessing on Quatro6x

4.1 Overview

In general, when a system is partitioned to n processors, we hope to get n times the speed compared to using one processor. However, In fact, it is not so simple. For example, the three individual parts (channel, modulation, and matched filter) in the existing implementation can be run in real-time in about 7.5, 10.2, and 8.3 ms per frame, respectively, but the actual run time is about 20 ms per frame after we connect three main parts and download them to Quatro6x DSP board [32]. Why did the run time increase after they are connected? In the early part of this chapter, we profile unexpected the time-consumption to identify the actual performance. Then, in the latter part of the chapter, we point out that how care must be taken to apply double buffering scheme.

4.2 Structure of DSP Partition

Tsai [32] implement the sequential-10240 structure in the existing simulator. In this section, we discuss with several structures of DSP partition, such as parallel structure, sequential-512 and pipelining-512. Then, we devote to implement pipelining-512 structure.

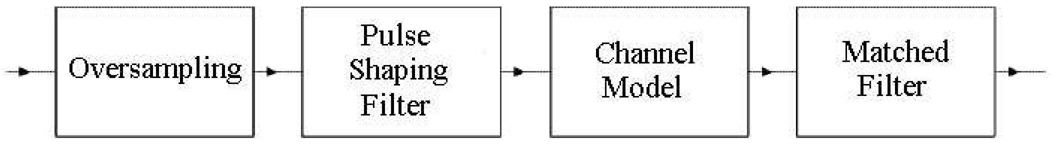


Fig. 4.1: Interpolated fitering system with the cascade of pulse shaping filter, channel, and matched filter.

4.2.1 Parallel Structure

In this subsection, we discuss with parallel structure using polyphase technique. However, we let go this thinking for keep flexibility and lack of enough FIFOPorts.

Polyphase with three functional blocks?

As described in chapter 2, Figures 2.5 and 2.6 depicts 4-polyphase implementation of the pulse shaping filter. Similarly, let us consider four functional blocks as shown in Fig. 4.1. As shown in Fig. 4.2, the question is: can we combine pulse shaping filter with the following channel model and matched filter into one block and apply the polyphase technique? The basic idea is correct, but it loses some flexibility. If we do this, we cannot get the output signal at the output of each functional block. For example, we cannot get the oversampled, transmit-filtered signal since we have combined these three functional blocks together. Particularly, if we combine all filters into one block, then we lose the flexibility of simulating different kinds of channels, especially time-varying channels. Since transmitter and receiver filters belong to the transmitter and the receiver, respectively, they will have to be implemented separately in practical real systems, anyway. Therefore, we keep the separation of these functional blocks.

Quatro6x DSPs parallel interconnection?

In previously existing implementation, we put the pulse shaping filter, the channel model, and the matched filter in three different DSPs and connect them in series. If four DSPs by parallel connection and each DSP included channel model, pulse shaping filter, and

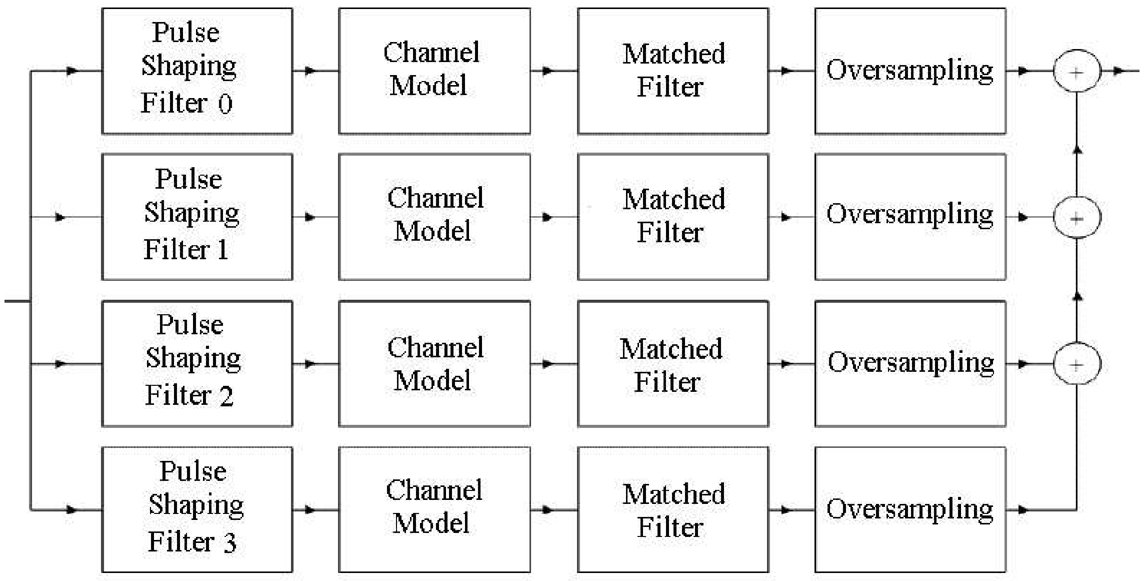


Fig. 4.2: Polyphase technique applied to the cascade of pulse shaping filter, channel, and matched filter.

matched filter. Then, the latency maybe almost vanish because no functional block across interprocessor. However, the scheduling may be a problem. We are not sure whether parallel interconnection will be better or not, maybe need to check the computational complexity to see if it is possible. However, due to only three FIFOPorts exist for four DSPs on Quatro6x board, that is, DSP0 have no FIFOPort, so we let go the parallel connection thinking.

4.2.2 Sequential Structure

In this subsection, we discuss with sequential structure, which use blocking mode DMA.

Blocking mode DMA

As discussed in chapter 3, the DMA functions we use are

dma_mem_to_port(int channel, int src, int* dest, int count, int block),*

dma_port_to_mem(int channel, int src, int* dest, int count, int block),*

dma_copy_mem(int channel, int src, int* dest, int count, int block).*

If “$block$” is true, the function waits until the move is completed before processing, which is called the blocking mode DMA.

Sequential-10240 structure

In the previous implementation, Tsai [32] employ sequential-10240 structure to processes a slot of information each time. It is data amount of 10240 samples per slot. As discussed previously, the FIFOLink buffer size is up to 512 samples. In the sequential-10240 structure, the channel model processor repeat blocking mode DMA 20 times through FIFO to get one slot of input, then compute, and then repeat blocking mode DMA 20 times through FIFO to output the result. However, the latency is big.

Sequential-512 structure

If we process 512 samples each time, then we hope that we can cut the latency down to 1/20. In addition, we also reduce memory usage. In the sequential-512 structure, each time the channel model processor use blocking mode DMA to input 512 samples through FIFO, then compute, and then use blocking mode DMA to output 512 samples through FIFO.

4.2.3 Pipelining Structure

We propose a pipelining structure, which use non-blocking mode DMA and double buffer. Partial idea of the structure is from ’C6x pipeline operation [40] and software pipelining [43], [39].

Terms of pipelining

There are three basic terms common to software pipelining: prolog, loop kernel, and epilog. The first stage, prolog, contains instructions to build the second-stage loop cycle, and the epilog stage contains instructions to finish all loop iterations [8]. In our pipelining-512 structure, we want to pipeline a FIFO-buffered block size of 512 samples. Besides this size is different to the instruction cycle of software pipelining, the terms prolog, loop kernel, and epilog are still available. We give a comparison in Table 4.1.

Non-blocking mode DMA

As discussed previously, the DMA transfer function we use are

dma_mem_to_port(int channel, int src, int* dest, int count, int block),*

dma_port_to_mem(int channel, int src, int* dest, int count, int block),*

dma_copy_mem(int channel, int src, int* dest, int count, int block).*

If "$block$" is false, then program execution continues immediately after the DMA operation starts, which is belonging to the non-blocking mode. There is a while-loop to check the dma_done status inside DMA transfer function. The dma_done command reports DMA operation done ready or not yet, which returns true if DMA channel available. If we use non-blocking mode, this while-loop skips in DMA support library. We could put the while-loop of checking dma_done outside DMA transfer function. Then, we insert the code of a wanted pipelining-512 looping unit between non-blocking call DMA transfer function and while-looping check DMA completion status.

Pipelining-512 structure

As shown in Fig. 4.3 and Table 4.2, there are five operations used for the channel simulator, including

1. dma_copy_mem move in real part noise from external memory without FIFO,
2. dma_copy_mem move in imaginary part noise from external memory without FIFO,
3. $dma\_port\_to\_mem$ move in nth-512 $Fout$ via FIFO from modulator,
4. CPU compute $(n-1)$th-512 $Fout$ according channel model to form $Path$, and
5. $dma\_mem\_to\_port$ move out $(n-2)$th-512 $Path$ via FIFO to matched filter.

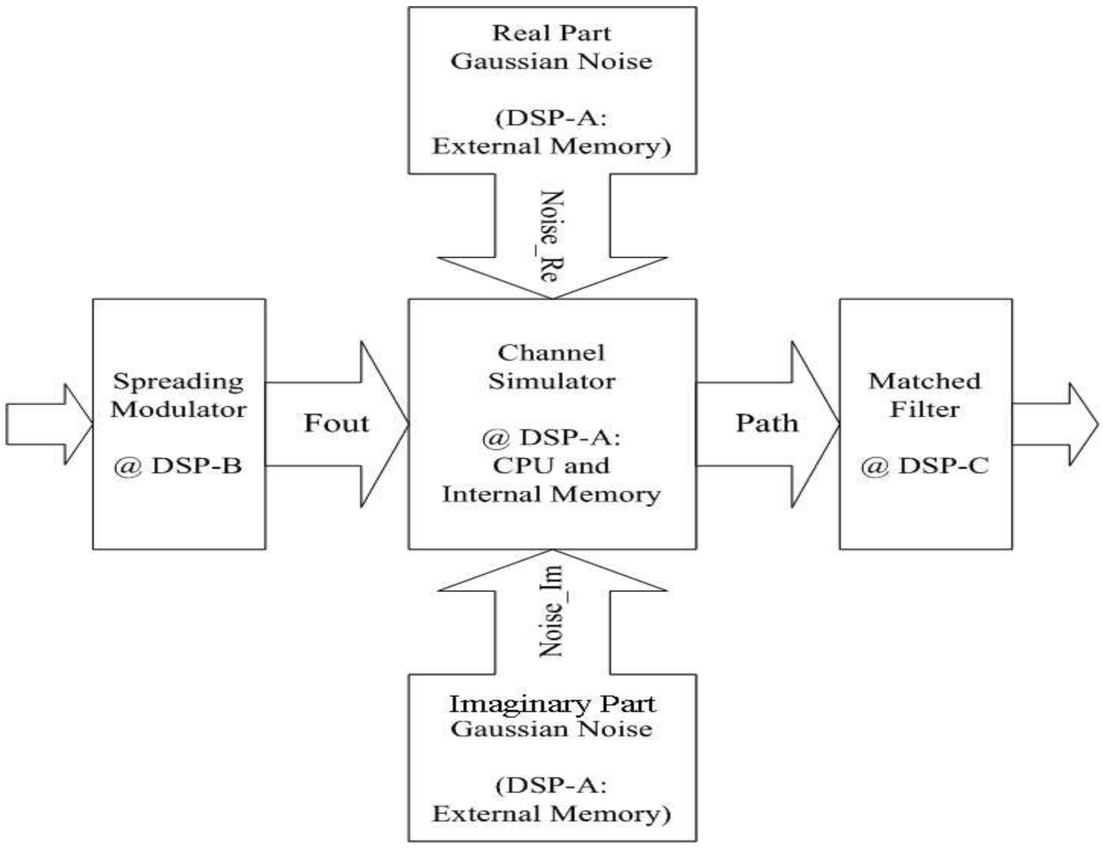


Fig. 4.3: Operations of one computing and four movement used for the channel simulator.

We assign four independent DMA channels for the above four kinds of DMA transfer, respectively. Based on non-blocking DMA, we apply dual buffers for $Fout$ and $Path$, respectively. That is, LF, RF, LP, and RP as shown in Table 4.2. One buffer used for computing while another buffer used for moving data. However, we pay attention to a matter of double buffering in the latter part of this chapter.

Table 4.1: Pipelining Structure, Software Pipelining and 'C6x Pipeline Operation

	Our Pipeline-512 Structure	Software Pipeline and 'C6x Pipeline
Cycle	Block-based 512 samples	Clock-based instructions
Device	Multiprocessors of three DSPs	Eight functional units of CPU core
Stage	Prolog, loop, and epilog	Prolog, loop, and epilog
Phase	Move in, computing, and move out.	Fetch, decode, and execute.

Table 4.2: Scheduling Table of Our Pipelining-512 Structure

Block Cycle	Move In $Fout$ from Modulator	Channel CPU Processing $Fout$ to form $Path$	Move Out $Path$ to Matched Filter
1	1st-512 @LF	No operation	No operation
2	2nd-512 @RF	1st-512 @$LF \to LP$	No operation
3	3rd-512 @LF	2nd-512 @$RF \to RP$	1st-512 @LP
$\cdots$	$\cdots$	$\cdots$	$\cdots$
n	nth-512 @LF	$(n-1)$th-512 @ $RF \to RP$	$(n-2)$th-512 @LP
$\cdots$	$\cdots$	$\cdots$	$\cdots$
10	10th-512 @RF	9th-512 @$LF \to LP$	8th-512 @RP
11	No operation	10th-512 @$RF \to RP$	9th-512 @LP
12	No operation	No operation	10th-512 @RP

4.3 Actual Speed Observation

We insert the $uclock$ commands into the DSP program to observe actual run-time in detail. It is helpful to find out the unexpected time-consumption.

4.3.1 Observation on Single DSP Chip

As shown in Table 4.3, we observe only one individual program on a single processor DSP0, respectively. However, only first processor DSP0 is available to the $uclock$ command. There are many different versions programs belong to the channel, the modulation, the receiver programs, respectively. One of key information exists in the table, that is, the scramble code generation takes around 29 ms per run, and belongs to overhead but not looping part. Zoom in the subroutine; we see that some if-else statements inside the for-loop of many times looping. It may why this routine is so time-consuming.

4.3.2 Observation of Overall Connection Speed

Table 4.4 shows the overall speed after three DSP programs (channel, modulation, and receiver) are connected. There are many different versions programs belong to the different structures, such as sequential and pipelining. We are interested the time-consumption per slot because it is belonging to looping part. In 3GPP standard, a frame of 15 slots

Table 4.3: Observation on Single DSP

ms @ cpu0	Channel	Modulate	scramble	33-tap	9-tap	Remark
TEST_SPEED (ms/frame)	14.255 (14.255)	48.03 (10.685)	29.282 +8.063		8.719 (8.719)	1frame/run case3
DEMO (ms/frame)	16.269 (8.134)	51.379 (11.467)	28.445	237.325 (118.662)	17.458 (8.729)	2frame/run
MULTIUSER (ms/frame)	9250	51.307			17.458	noise by log, cos
SYSTEM (ms/frame)	16.268 (8.134)	249.359 (112.03)	29.3		17.458 (8.729)	2frame/run
SINGE_USER (ms/frame)	16.27 (8.135)	51.398 (11.467)	22.384 +6.073	237.325 (118.662)	17.458 (8.729)	20s/200run with Rake
CHANGING (ms/frame)	15.037 (7.519)	54.19 ($\simeq$ 12.6)	$\simeq$ 29	237.351 (118.676)	17.458 (8.729)	251,92,75 in Table 4.4
Table 2.6	(6.67$\sim$ 11.72)	(10.24)			(8.346)	

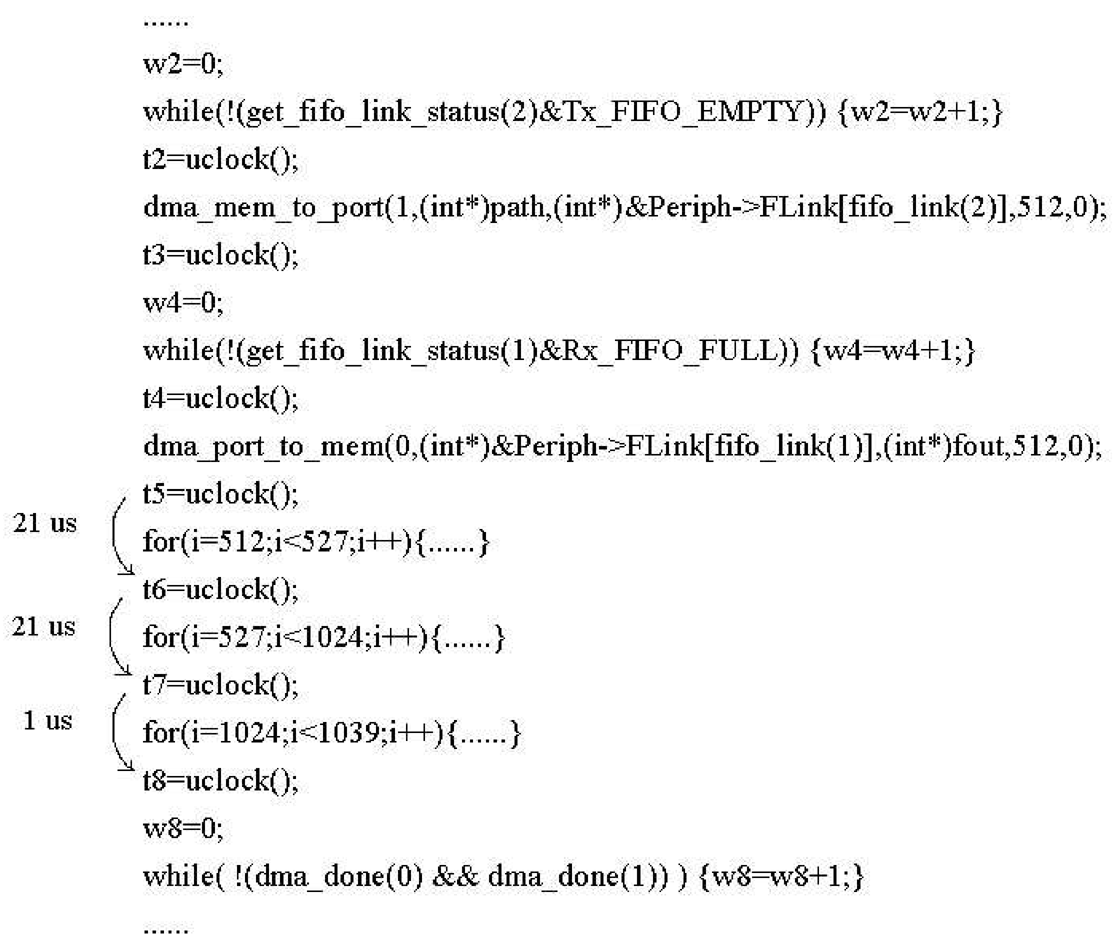
Table 4.4: Observation on Overall Connection Speed

Modulation / Channel	Receiver	Run	Scramble	Slot
pipelining-512	33-tap sequential-512	251 ms	28.9 ms	7421 us
sequential-10240	9-tap sequential-512	110 ms	30.5 ms	2637 us
pipelining-512	9-tap sequential-512	92 ms	29.9 ms	2096 us
pipelining-512	9-tap pipelining-512	75 ms	29.9 ms	1521 us
pipelining-512 (no many *uclock*)	9-tap pipelining-512	64 ms	29.9 ms	1170 us

has 10 ms length. As discussed previously, for FIFO size up to 512 samples, we need move 512 samples 20 times for a slot data. It means that a real-time criterion is about 666 μs per slot, that is, 33 μs per 512 samples. However, the better speed is about 1 ms per slot, which is about 1.5 times the criterion. We zoom in actual speed in the following subsections, respectively.

4.3.3 Observation by Changing Segments of Program

This subsection presents observations by changing segments of the channel program.



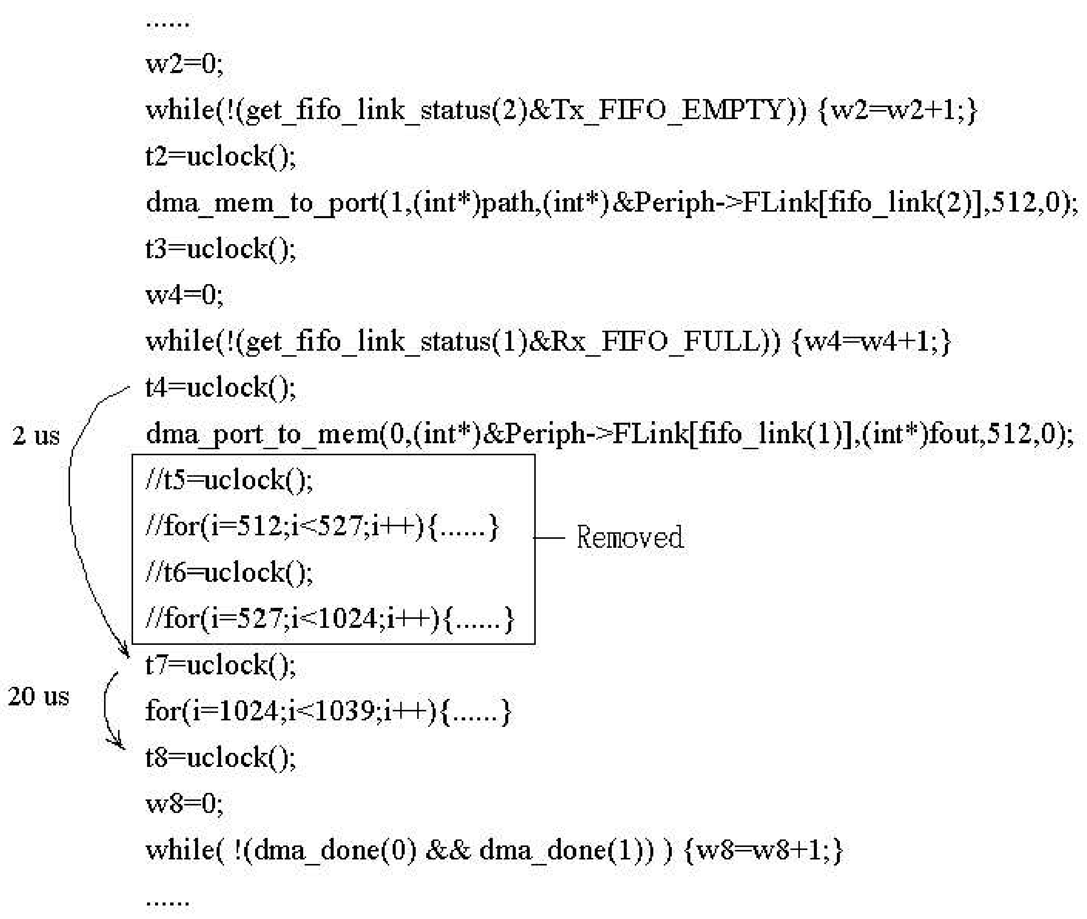
```
......
w2=0;
while(!(get_fifo_link_status(2)&Tx_FIFO_EMPTY)) {w2=w2+1;}
t2=uclock();
dma_mem_to_port(1,(int*)path,(int*)&Periph->FLink[fifo_link(2)],512,0);
t3=uclock();
w4=0;
while(!(get_fifo_link_status(1)&Rx_FIFO_FULL)) {w4=w4+1;}
t4=uclock();
dma_port_to_mem(0,(int*)&Periph->FLink[fifo_link(1)],(int*)fout,512,0);
t5=uclock();
for(i=512;i<527;i++){......}
t6=uclock();
for(i=527;i<1024;i++){......}
t7=uclock();
for(i=1024;i<1039;i++){......}
t8=uclock();
w8=0;
while( !(dma_done(0) && dma_done(1)) ) {w8=w8+1;}
......
```

Fig. 4.4: Case A0 program segment.

(A0) When run channel program with modulation and 9-tap receiver together

As shown in Fig. 4.4, the measured time difference between t5 and t6 is about 21 μs. Is it from the first for-loop computing-time itself or not?

The channel model program performs complex multiplications to input data and channel coefficient. If single path, then for each point, we need four real multiplications where two are for real part and two for imaginary part. The TMS320C6701 DSP chip of 167 MHz has two units to perform multiplication and six units for addition. Therefore, one multiplication requires on the average of 6 ns and 3 ns, depending on the utilization of the two multipliers.



```
......
w2=0;
while(!(get_fifo_link_status(2)&Tx_FIFO_EMPTY)) {w2=w2+1;}
t2=uclock();
dma_mem_to_port(1,(int*)path,(int*)&Periph->FLink[fifo_link(2)],512,0);
t3=uclock();
w4=0;
while(!(get_fifo_link_status(1)&Rx_FIFO_FULL)) {w4=w4+1;}
t4=uclock();
dma_port_to_mem(0,(int*)&Periph->FLink[fifo_link(1)],(int*)fout,512,0);
//t5=uclock();
//for(i=512;i<527;i++){......}
//t6=uclock();
//for(i=527;i<1024;i++){......}
t7=uclock();
for(i=1024;i<1039;i++){......}
t8=uclock();
w8=0;
while( !(dma_done(0) && dma_done(1)) ) {w8=w8+1;}
......
```

Fig. 4.5: Case A1 program segment.

The 21 μs between t6 and t7 is reasonable for the second for-loop computing time. In the second-loop of 497 points, for both two paths, we need $4 \times 497 \times 2 = 3976$ multiplications. If 6 ns for a multiplication, then $3976 \times 6 = 23856$ ns, it is near 21 μs.

Similarly, the 1 μs between t7 and t8 is reasonable for third for-loop of 15 points to compute the second single path each point, we need $4 \times 15 \times 1 \times 6 = 360$ ns, it is smaller than 1 μs resolution of *uclock* command.

Therefore, the first for-loop of 15 points should take about 1 μs, but not measured 21 μs, to compute first single path each point. Where does this 21 μs between t5 and t6 come from?

(A1) When the first and second for-loop are removed

In this A1 case as shown in Fig. 4.5, we remove the first for-loop, the second for-loop, t5 and t6 measurement, but still keep the third for-loop in program. Therefore, the time difference between t7 and t8 become 20 μs, but not keep 1 μs as before. Thinking about the 21 μs between t5 and t6 in the previous case A0, and the 20 μs between t7 and t8 in this current case A1, it hints that the strange time appears around just next command after of the dma_port_to_mem(0,......,0) of non-blocking.

(A2) When dma_port_to_mem is changed to blocking mode

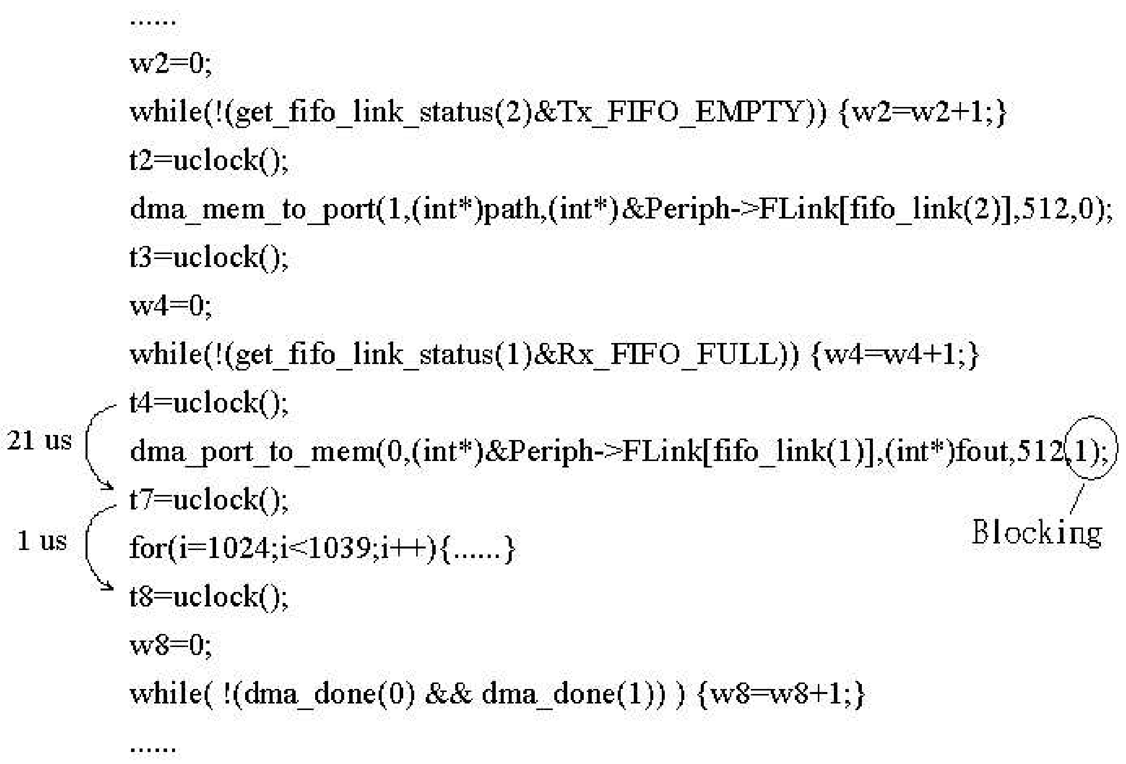
As shown in Fig. 4.6, we modified to case A2 from case A1, only changed to blocking mode in dma_port_to_mem. Then time difference between t4 and t7 become to 21 μs in case A2 from 2 μs in case A1, also time difference between t7 and t8 become to 1 μs in case A2 from 20 μs in case A1. It hints that the strange time such as 20 μs in case A1 is correlated to dma_port_to_mem of non-blocking mode. Moreover, we know that dma_port_to_mem of blocking mode take about 21 μs to move 512 32-bit integer data through bursting FIFO buffer.

(A3) When dma_mem_to_port is changed to blocking mode

As shown in Fig. 4.7, case A3 is modified from case A2; the dma_mem_to_port between t2 and t3 is changed to use blocking mode. Then time difference between t2 and t3 is 19 μs, also time difference between t4 and t7 is 21 μs. We know again that dma_mem_to_port or dma_port_to_mem of blocking mode takes around 20 μs to move 512 32-bit integer data through bursting FIFO.

(A4) When both the first and the second for-loops are recovered

As shown in Fig. 4.8, case A4 is modified from case A3; both the first for-loop of prolog 15 points and the second for-loop of looping 497 points are recovered. Then blocking mode dma_mem_to_port takes 18 μs between t2 and t3, and blocking mode



```
......
w2=0;
while(!(get_fifo_link_status(2)&Tx_FIFO_EMPTY)) {w2=w2+1;}
t2=uclock();
dma_mem_to_port(1,(int*)path,(int*)&Periph->FLink[fifo_link(2)],512,0);
t3=uclock();
w4=0;
while(!(get_fifo_link_status(1)&Rx_FIFO_FULL)) {w4=w4+1;}
t4=uclock();
dma_port_to_mem(0,(int*)&Periph->FLink[fifo_link(1)],(int*)fout,512,1);
t7=uclock();
for(i=1024;i<1039;i++){......}
t8=uclock();
w8=0;
while( !(dma_done(0) && dma_done(1)) ) {w8=w8+1;}
......
```

21 us
1 us
Blocking

Fig. 4.6: Case A2 program segment.

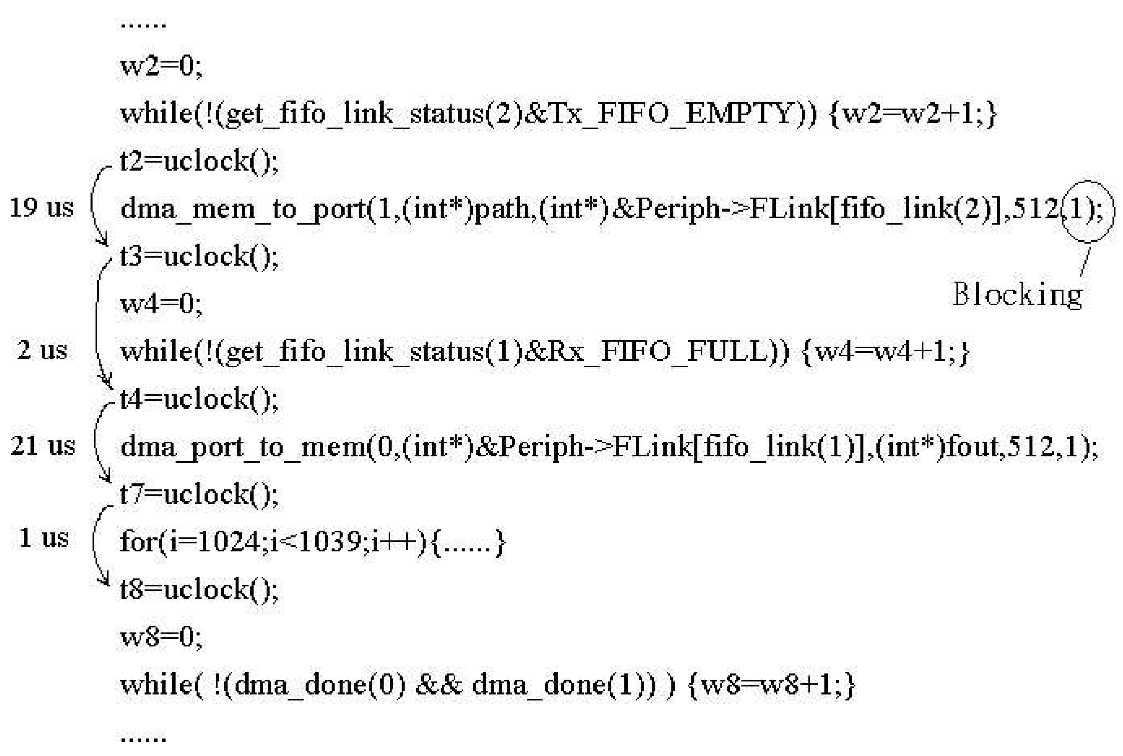
dma_port_to_mem takes 21 μs between t4 and t5. The 21 μs between t6 and t7 is reasonable for the second for-loop to compute two paths of 497 points. Note that the time difference between t5 and t6 is 2 μs after blocking mode DMA in the case A4, but not still 21 μs after non-blocking mode DMA in case A0.

4.3.4 Observation by Removing Data Transfer Commands

As shown in Table 4.5, from case B0 toward B5, we observe time-consumption by removing gradually the commands used for data transfer.

(B0) When running channel program with modulation and 9-tap receiver together

The actual run time is about 63.73 ms per run after we connect the three main parts of channel, modulation, and 9-tap receiver, and download them to Quatro6x DSP board.



```
......
w2=0;
while(!(get_fifo_link_status(2)&Tx_FIFO_EMPTY)) {w2=w2+1;}
t2=uclock();
dma_mem_to_port(1,(int*)path,(int*)&Periph->FLink[fifo_link(2)],512,1);
t3=uclock();
w4=0;
while(!(get_fifo_link_status(1)&Rx_FIFO_FULL)) {w4=w4+1;}
t4=uclock();
dma_port_to_mem(0,(int*)&Periph->FLink[fifo_link(1)],(int*)fout,512,1);
t7=uclock();
for(i=1024;i<1039;i++){......}
t8=uclock();
w8=0;
while( !(dma_done(0) && dma_done(1)) ) {w8=w8+1;}
......
```

19 us
2 us
21 us
1 us
Blocking

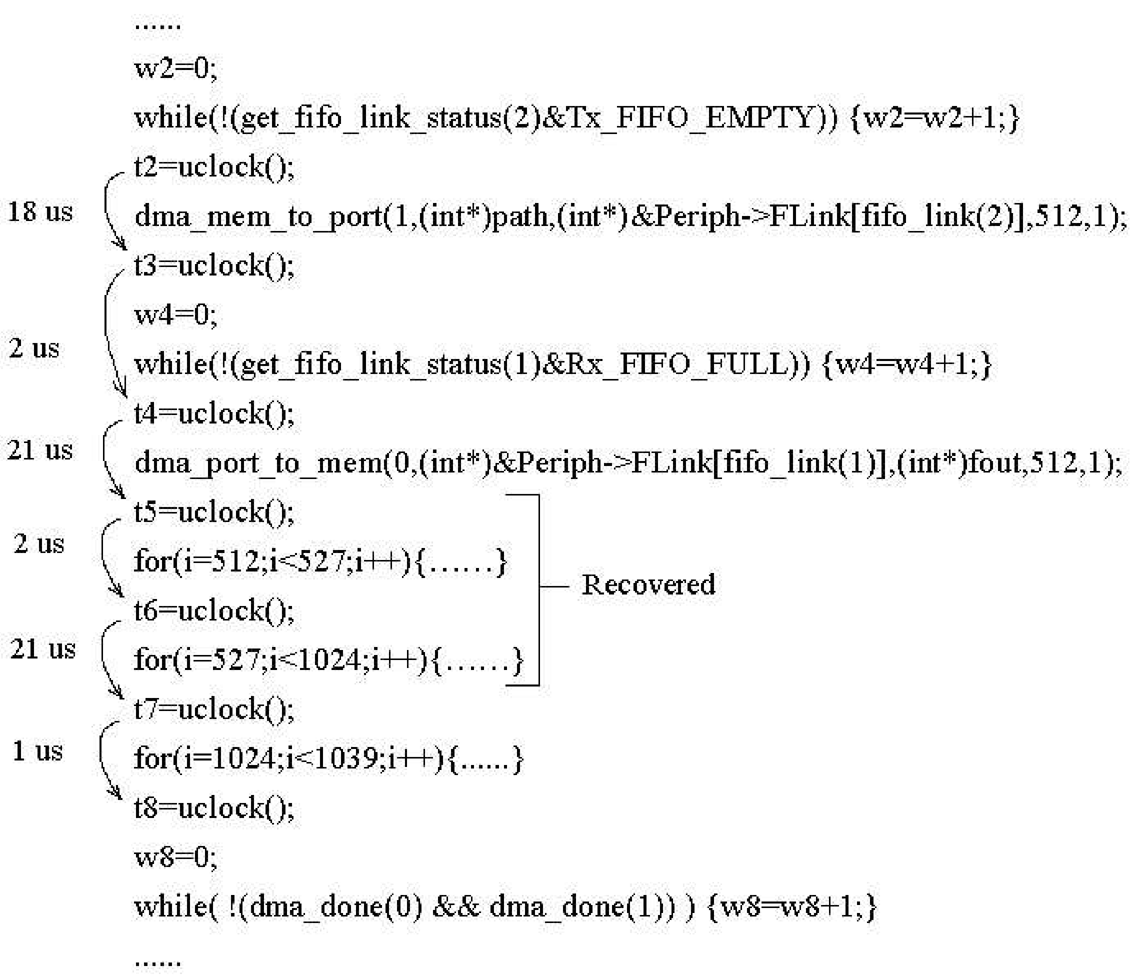
Fig. 4.7: Case A3 program segment.

(B1) When commands starting with "while(!(get_fifo_link_status" are removed

In case B1, we remove commands starting with "while(!(get_fifo_link_status." Only single DSP of channel program is run. Then the run time reduces to 31.24 ms from 63.73 per run. It is surprising that handshaking waits around 30 ms for scramble code generation to be ready. However, there still exists a strange 20 μs between t5 and t6 after non-blocking mode DMA.

(B2) When the "dma_port_to_mem" commands are removed

In case B2, we remove the "dma_port_to_mem" commands from case B1. Only single DSP of channel program run. Then the run time reduces to 27.91 from 31.24 ms per run. Moreover, we observe the time difference between t5 and t6 becomes 2 μs in current case B2 from previous case B1. Excitingly, the strange time-consumption almost vanish in the first for-loop between t5 and t6, that is, it reduces 18 μs largely since the non-blocking



```
......
w2=0;
while(!(get_fifo_link_status(2)&Tx_FIFO_EMPTY)) {w2=w2+1;}
t2=uclock();
dma_mem_to_port(1,(int*)path,(int*)&Periph->FLink[fifo_link(2)],512,1);
t3=uclock();
w4=0;
while(!(get_fifo_link_status(1)&Rx_FIFO_FULL)) {w4=w4+1;}
t4=uclock();
dma_port_to_mem(0,(int*)&Periph->FLink[fifo_link(1)],(int*)fout,512,1);
t5=uclock();
for(i=512;i<527;i++){......}
t6=uclock();
for(i=527;i<1024;i++){......}
t7=uclock();
for(i=1024;i<1039;i++){......}
t8=uclock();
w8=0;
while( !(dma_done(0) && dma_done(1)) ) {w8=w8+1;}
......
```

Fig. 4.8: Case A4 program segment.

mode dma_port_to_mem commands are removed. It is a close hint to find why the first-loop takes a strange 20 μs but not reasonable 1 μs.

(B3) When the "dma_mem_to_port" commands are removed

In case B3, we remove the "dma_mem_to_port" commands from case B2. Only single DSP of channel program is run. Then the run time reduces to 18.66 ms from 27.91 per run. Moreover, we observe the time difference between t3 and t4 becomes 1 μs in current case B3 from 16 μs in previous case B2. The reduced 15 μs is correlated to that 600 times of commands in non-blocking mode dma_mem_to_port are removed.

(B4) When the "dma_copy_mem" and "dma_copy" commands are removed

In case B4, we remove the "dma_copy_mem" and "dma_copy" commands from case B3. Only single DSP of channel program is run. Then the run time reduces to 14.03 ms from 18.66 per run.

(B5) When commands starting with "while(!dma_done" are removed

In case B5, we remove commands starting with "while(!dma_done" from case B4. Only single DSP of channel program is run. Then the run time reduces to 13.8 ms from 14.03 per run. The 13.8 ms is reasonable for the computing time without DMA or FIFO related command, that is, we need 13.8 ms for 600 times of 512-integer data each. In other words, it is about 23 μs per each 512-integer data; it is consistent with computing time of 512 points (Refer to analysis in case A0; the 23 μs should be consisting of 1 μs of the 15-points prolog, 21 μs of the 497-points looping, and 1 μs of the 15-points epilog).

4.3.5 Observation on the Sequential-10240 Structure

As discussed previously, the existing simulator simulator employ sequential-10240 structure. As shown in Tables 4.6, 4.7, and 4.8, we zoom in the time-consumption per slot, per 512 move-in and move-out 512 samples, respectively.

Table 4.5: Observation on Case B Type of Removing Data Transfer Commands

Case	B0	B1	B2	B3	B4	B5
ms per run	63.72	31.24	27.91	18.66	14.03	13.8
t3-t2 (μs)		2	2	1	1	1
t4-t3 (μs)		17	16	1	2	1
t5-t4 (μs)		2	1	1	1	1
t6-t5 (μs)		20	2	2	2	2
t7-t6 (μs)		21	21	21	20	21
t8-t7 (μs)		1	1	1	2	1

Table 4.6: Observation on the Sequential-10240 Structure per Slot

Slot Index	Move-in 10240	Computing 10240	Move-out 10240
[0]	31279 $\sim$ 31308 μs	1161 μs	735 $\sim$ 759 μs
[1]$\sim$[29]	738 $\sim$ 762 μs	1161 μs	735 $\sim$ 761 μs
Remark	Table 4.7	include noise move	Table 4.8

As observed previously, the scramble code generation is time-time-consuming about 29 ms on single modulation processor. As shown in Table 4.6, we see that it is to result in about 30 ms handshaking wait at the first slot only.

The sequential-10240 structure does blocking-mode DMA movement continues 20 times via FIFO buffer of 512 samples size. Each DMA movement execute after FIFO status handshaking ready. In order to double check the handshaking behavior, we insert an accumulated adder inside the while-loop to count how many times the while-loop to be testing. Here, we use the "add" as units meaning one testing happens per "add".

As shown in Tables 4.7 and 4.8, we observe only 1st-512 movement has no handshaking wait almost, which means that the period of a slot is enough to get FIFO status ready. Nevertheless, the other movement per 512 samples exist wait about 0.6 μs/add, which means that the period between sequential movements 512-samples is not enough to get FIFO status ready. The handshaking wait is a looping cost.

Table 4.7: Observation on the Sequential-10240 Structure per Move-in 512

Index per 512	Handshaking Wait Time	While-looping Times	Move-in 512
[0]	1 μs	count 0 times add.	22 μs
[1]$\sim$[19]	16 $\sim$ 17 μs	count 26 $\sim$ 27 times add.	21 $\sim$ 22 us

Table 4.8: Observation on the Sequential-10240 Structure per Move-out 512

Index per 512	Handshaking Wait Time	While-looping Times	Move-out 512
[0]	2 us	count 0 times add.	18 us
[1]$\sim$[19]	19 $\sim$ 20 us	count 32 $\sim$ 33 times add.	18 $\sim$ 19 us

4.3.6 Observation on the Pipelining-512 Structure

As discussed previously, in the subsection of the observation of overall connection speed, there are 251, 92, 75 ms per run, respectively, as shown in Table 4.4. Among these three kinds of versions, the 251 ms version is using 33-tap receiver, the others is using 9-tap receiver. All these three versions employ pipelining-512 structure for channel and modulator. In this subsection, we pay attention to profile these three versions.

Table 4.9 is a summary table, where rich information hint unexpected time-consumption. There are five tables zoom in this table, including Tables 4.10, 4.11, 4.12, 4.13, and 4.14, where Tables 4.10 and 4.11 are expanded from $tm14$ and $tm35$, respectively, which are belonging to the looping-512 kernel. Table 4.12 shows while-loop behavior of FIFO handshaking between channel and receiver. Table 4.13 shows while-loop behavior of FIFO handshaking between modulator and channel. Table 4.14 shows the while-loop behavior for checking dma_done.

As shown in Table 4.9, there are eight columns. The most right column shows the corresponding piecewise part in the pipelining-512 channel program. The middle six columns used for the three versions of 251, 92, and 75 ms, where two columns per version: one for 1st slot, another for other 29 slots. The most left column used for index stating with tm including $tm1$ to $tm43$. The rows between $tm1$ and $tm43$ is processing one slot: half slot between $tm1$ and $tm22$, another half slot between $tm23$ and $tm43$.

We summarizes several observations listed as below.

Normal verse strange while-looping wait time per "add"

As discussed previously, we use $uclock$ command to profile time-consumption, and we insert an accumulated adder inside the while-loop to count how many times the while-loop to be testing. Here, we the "add" as units meaning one testing happens per "add". The data starting with the "/" means that the data is belonging to "how many times while-loop" in "add" unit; otherwise, it is time-consumption in μs unit. We check many data of "μs/add" form. In general, it is ranging between 0.4 and 0.6 μs/add for normal case. That is, a while-loop test takes about 0.5 μs one times in average. However, it is not reasonable for enough long time μs per zero count "add". For example, such as 17/0 (or 35/0) hints that it is impossible to take 17 (or 35) μs but do nothing for a while-loop test. But, such as 1/0 (or 2/0) is usually normal due to μs resolution of "$uclock$" command.

Scramble code generation and 1st slot handshaking wait

As shown in $tm4$ with "/add", there are 28973 /70846, 29958 /70867, and 29951 /70854 for 251, 92, 75 ms versions, respectively. It is about 0.4 μs/add, that is, it is belonging to the while-looping wait time itself. As observed previously, the scramble code generation takes around 30 ms. We know that it results in the first slot handshaking wait only.

Long-time while-loop of "/0 add" is usually after commands starting with "dma"

For example, the while-loop of the 35/0 in row $tm4$/add is after the "dma_copy_mem", the while-loop of the 20/0 in row $tm6$ is after the "dma_port_to_mem". Similarly, the 34/0 in $tm26$ and 20/0 in $tm28$ as shown in Table 4.9; The 17/0 in $at4$ and $at12$ in Table 4.10 and 4.11.

Fast DSP wait slow DSP for FIFO handshaking status

As shown in Table 4.12, there are long waiting time for 251065 μs version due to 33-tap receiver. For 91815 μs version, it almost vanish except for $at10[3]$. It is because this version use 9-tap receiver but still sequential structure. Therefore, it also vanish for 75031 μs because the 9-tap receiver changed to pipelining.

4.4 Zoom in DMA Library

After observation previously, we know that there is correlation between some unexpected time-consumption and the DMA transfer commands. In this section, we pay attention to see II support library, including “dma_mem_to_port”, “dma_port_to_mem”, and “dma_copy_mem”.

As shown in Fig. 4.9, the source code of these three functions are almost same. A key difference is the different setting values located at the “dma→control.primary”. The primary control register controls the main operation of the DMA and contains 16-bit fields [12]. Refer to [41] for the bit-field description. The bit 5 to 4 are “SRC DIR” field, used for source address modification after element transfers. The bit 7 to 6 are “DST DIR” field, used for destination address modification after element transfers. The bit 24 is “PRI” field, used for priority mode of DMA versus CPU: PRI=0 for CPU priority, and PRI=1 for DMA priority [41].

As shown in Fig. 4.9, these three DMA functions set 1 at the PRI. That is, DMA priority is higher than CPU. In other words, CPU maybe waits for DMA done ready. It is a more close hint for our finding the unexpected time-consumption.

4.5 Double Buffering Technique

Double buffering is a method of using dual buffers to achieve efficient one-way data transmission between two processors or between a processor and a peripheral device. Each buffer is a block of storage through which data transmit from one processor or device to the other. The receiving processor reads the transmitted data from one buffer while the sending processor simultaneously prepares the data for the next transmission in the alternate buffer [35].

Table 4.9: Observation on the Pipelining-512 Channel

μs /add	251ms [0]	slot [1∼29]	92ms [0]	slot [1∼29]	75ms [0]	slot [1∼29]	corresponding piecewise part in the pipelining-512 channel
$tm1$	1	1	1	1	1	1	j=(slot*10240)%298801
$tm2$	1	1	1	1	1	1	dma_copy_mem(2,*,*,2560,0)
$tm3$	34	34	35	34	35	34	dma_copy_mem(3,*,*,2560,0)
$tm4$	28973	35	29958	35	29951	34	while(!(get_fifo_link_status(1...
/add	/70846	/0	/70867	/0	/70854	/0	/how many times while-loop
$tm5$	2	2	2	2	2	2	dma_port_to_mem(0,*,*,512,0)
$tm6$	20 /0	20 /0	20 /0	20 /0	20 /0	20 /0	while(!(dma_done(0)&&...(3)...
$tm7$	17 /36	17 /36	17 /36	17 /36	16 /36	17 /36	while(!(get_fifo_link_status(1...
$tm8$	2	2	1	2	2	1	dma_port_to_mem(0,*,*,512,0)
$tm9$	20	20	21	20	20	21	for(i=0;i<15;i++){...}
$tm10$	19	19	18	19	19	18	for(i=15;i<308;i++){...}
$tm11$	9	9	9	9	9	9	for(i=308;i<512;i++){...}
$tm12$	1	1	2	1	1	2	for(i=512;i<527;i++){...}
$tm13$	1 /0	1 /0	1 /0	1 /0	1 /0	1 /0	while(!dma_done(0))...
$tm14$	2052	2894	1035	1112	536	536	for(p=3;p<=10;p=p+2){...}
$tm15$	313	313	1	1	1	2	while(!(get_fifo_link_status(2...
/add	/749	/749	/0	/0	/0	/0	/how many times while-loop
$tm16$	2	2	2	2	2	1	dma_mem_to_port(1,*,*,512,0)
$tm17$	17	17	18	17	17	18	for(i=512;i<527;i++){...}
$tm18$	21	21	20	21	21	20	for(i=527;i<1024;i++){...}
$tm19$	1	1	1	1	1	1	for(i=1024;i<1039;i++){...}
$tm20$	1 /0	1 /0	2 /2	2 /2	1 /2	2 /2	while(!dma_done(1))...
$tm21$	336	336	1	1	2	1	while(!(get_fifo_link_status(2...
/add	/805	/805	/0	/0	/0	/0	/how many times while-loop
$tm22$	2	2	2	2	2	2	dma_mem_to_port(1,*,*,512,0)
$tm23$	17	17	17	17	16	17	j=(slot*10240+5120)%298801
$tm24$	1	1	1	1	2	1	dma_copy_mem(2,*,*,2560,0)
$tm25$	35	35	35	35	34	34	dma_copy_mem(3,*,*,2560,0)
$tm26$	34 /0	34 /0	34 /0	34 /0	34 /0	34 /0	while(!(get_fifo_link_status(1...
$tm27$	2	2	1	2	2	2	dma_port_to_mem(0,*,*,512,0)
$tm28$	20 /0	20 /0	21 /0	20 /0	21 /0	21 /0	while(!(dma_done(0)&&...(3)...
$tm29$	16 /35	16 /35	16 /35	16 /35	16 /35	16 /35	while(!(get_fifo_link_status(1...
$tm30$	2	2	2	1	2	2	dma_port_to_mem(0,*,*,512,0)
$tm31$	20	20	20	21	20	20	for(i=0;i<15;i++){...}
$tm32$	21	20	21	20	20	21	for(i=15;i<512;i++){...}
$tm33$	1	2	1	2	2	1	for(i=512;i<527;i++){...}
$tm34$	1 /0	1 /0	1 /0	1 /0	1 /0	1 /0	while(!dma_done(0))...
$tm35$	2815	2815	535	536	536	536	for(p=13;p<=20;p=p+2){...}
$tm36$	299	299	1	1	2	1	while(!(get_fifo_link_status(2...
/add	/714	/714	/0	/0	/0	/0	/how many times while-loop
$tm37$	2	2	2	2	2	2	dma_mem_to_port(1,*,*,512,0)
$tm38$	17	17	17	17	17	17	for(i=512;i<527;i++){...}
$tm39$	20	20	20	20	20	21	for(i=527;i<1024;i++){...}
$tm40$	1	1	2	2	1	1	for(i=1024;i<1039;i++){...}
$tm41$	1 /0	1 /0	1 /0	1 /0	2 /0	2 /0	while(!dma_done(1))...
$tm42$	321	322	1	1	1	1	while(!(get_fifo_link_status(2...
/add	/770	/770	/0	/0	/0	/0	/how many times while-loop
$tm43$	18	18	18	18	18	18	dma_mem_to_port(1,*,*,512,1)

Table 4.10: Observation on the Pipelining-512 Channel tm14

μs /add	2894 [3]	2894 [5]	2894 [7,9]	1112 [3]	1112 [5,7,9]	536 [3,5,7,9]	corresponding piecewise part in Table 4.9 tm14: 2894,1112,536
$at1$	2	2	1	2	2	1	for(p=3;p<=10,p=p+2){
$at2$	176	1	313	1	1	1	while(!(get_fifo_link_status(2...
/add	/418	/0	/748	/0	/0	/0	/how many times while-loop
$at3$	2	2	1	2	2	2	dma_mem_to_port(1,*,*,512,0)
$at4$	17 /0	17 /0	17 /0	17 /0	17 /0	17 /0	while(!(get_fifo_link_status(1...
$at5$	1	1	2	1	2	2	dma_port_to_mem(0,*,*,512,0)
$at6$	21	21	20	20	20	20	for(i=512;i<527;i++){...}
$at7$	20	20	21	21	20	20	for(i=527;i<1024;i++){...}
$at8$	2	2	1	1	2	1	for(i=1024;i<1039;i++){...}
$at9$	1 /0	1 /0	1 /0	2 /0	2 /0	2 /0	while(!(dma_done(0)&&...(1)...
$at10$	668	277	314	579	1	2	while(!(get_fifo_link_status(2...
/add	/1603	/659	/751	/1391	/0	/0	/how many times while-loop
$at11$	2	2	1	2	2	2	dma_mem_to_port(1,*,*,512,0)
$at12$	17 /0	16 /0	17 /0	17 /0	17 /0	17 /0	while(!(get_fifo_link_status(1...
$at13$	2	2	2	1	2	2	dma_port_to_mem(0,*,*,512,0)
$at14$	20	20	20	22	20	20	for(i=0;i<15;i++){...}
$at15$	21	21	20	20	20	20	for(i=15;i<512;i++){...}
$at16$	1	1	2	2	2	1	for(i=512;i<527;i++){...}
$at17$	1 /0	2 /0	1 /0	1 /0	1 /0	1 /0	while(!(dma_done(0)&&...(1)...

Table 4.11: Observation on the Pipelining-512 Channel tm35

μs /add	2815 [13]	2815 [15,17,19]	535 [13,15,17,19]	536 [13,15,17,19]	corresponding piecewise part in Table 4.9 tm35: 2815,535,536
$at1$	1	1	1	1	for(p=13;p<=20,p=p+2){
$at2$	205	299	1	2	while(!(get_fifo_link_status(2...
/add	/487	/714	/0	/0	/how many times while-loop
$at3$	2	1	2	2	dma_mem_to_port(1,*,*,512,0)
$at4$	16 /0	17 /0	17 /0	17 /0	while(!(get_fifo_link_status(1...
$at5$	2	2	2	2	dma_port_to_mem(0,*,*,512,0)
$at6$	21	20	20	20	for(i=512;i<527;i++){...}
$at7$	20	21	20	20	for(i=527;i<1024;i++){...}
$at8$	2	1	2	2	for(i=1024;i<1039;i++){...}
$at9$	1 /0	2 /0	1 /0	2 /0	while(!(dma_done(0)&&...(1)...
$at10$	300	299	2	1	while(!(get_fifo_link_status(2...
/add	/717	/715	/0	/0	/how many times while-loop
$at11$	2	2	2	2	dma_mem_to_port(1,*,*,512,0)
$at12$	17 /0	17 /0	17 /0	17 /0	while(!(get_fifo_link_status(1...
$at13$	1	1	2	2	dma_port_to_mem(0,*,*,512,0)
$at14$	21	21	20	20	for(i=0;i<15;i++){...}
$at15$	20	20	21	20	for(i=15;i<512;i++){...}
$at16$	2	1	1	1	for(i=512;i<527;i++){...}
$at17$	1 /0	2 /0	2 /0	2 /0	while(!(dma_done(0)&&...(1)...

Table 4.12: Observation on the “while(!(get_fifo_link_status(2)&Tx_FIFO_EMPTY))”

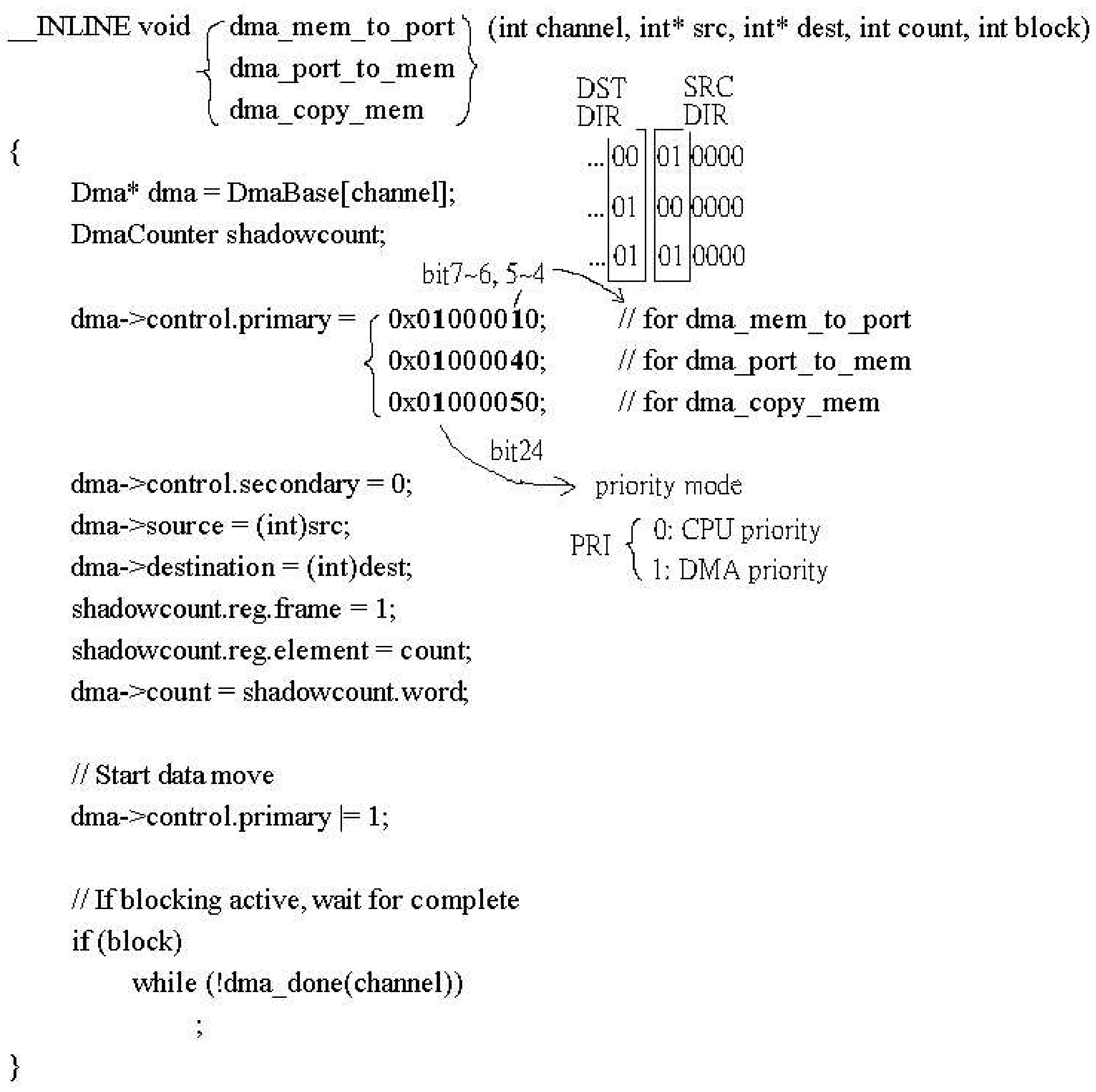
μs / add	251065 μs	91815 μs	75031 μs
$tm15$[0∼29]	314 / 749	1 / 0	1 / 0
$tm21$[0∼29]	336 / 805	1 / 0	1 / 0
$tm36$[0∼29]	299 / 714	1 / 0	1 / 0
$tm42$[0∼29]	322 / 770	1 / 0	1 / 0
$at2$[3]	176 / 418	1 / 0	1 / 0
$at2$[5]	1 / 0	1 / 0	1 / 0
$at2$[7,9]	313 / 748	1 / 0	1 / 0
$at2$[13]	205 / 487	1 / 0	1 / 0
$at2$[15,17,19]	299 / 714	1 / 0	2 / 0
$at10$[3]	668 / 1603	579 / 1392	2 / 0
$at10$[5]	277 / 659	1 / 0	2 / 0
$at10$[7,9]	314 / 751	1 / 0	2 / 0
$at10$[13,15,17,19]	300 / 716	2 / 0	1 / 0

Table 4.13: Observation on the “while(!(get_fifo_link_status(1)&Rx_FIFO_FULL))”

μs / add	251065 μs	91815 μs	75031 μs
$tm4$[0]	28973 / 70846	29958 / 70867	29951 / 70854
$tm4$[1∼29]	35 / 0	35 / 0	34 / 0
$tm7$[0∼29]	17 / 36	17 / 36	17 / 36
$tm26$[0∼29]	34 / 0	34 / 0	34 / 0
$tm29$[0∼29]	16 / 35	16 / 35	16 / 35
$at4$[3,5,7,9,13,15,17,19]	17 / 0	17 / 0	17 / 0
$at12$[3,5,7,9,13,15,17,19]	17 / 0	17 / 0	17 / 0

Table 4.14: Observation on Commands Starting with “while!(dma_done”

μs / add	251065 μs	91815 μs	75031 μs
$tm6$[0∼29]	20 / 0	20 / 0	20 / 0
$tm13$[0∼29]	1 / 0	1 / 0	1 / 0
$tm20$[0∼29]	1 / 0	2 / 2	2 / 2
$tm28$[0∼29]	20 / 0	20 / 0	21 / 0
$tm34$[0∼29]	1 / 0	1 / 0	1 / 0
$tm41$[0∼29]	1 / 0	1 / 0	2 / 0
$at9$[3,5,7,9,13,15,17,19]	1 / 0	2 / 0	2 / 0
$at17$[3,5,7,9,13,15,17,19]	2 / 0	1 / 0	1 / 0



```
__INLINE void  dma_mem_to_port  (int channel, int* src, int* dest, int count, int block)
               dma_port_to_mem
               dma_copy_mem
{
    Dma* dma = DmaBase[channel];
    DmaCounter shadowcount;

    dma->control.primary =  0x01000010;      // for dma_mem_to_port
                            0x01000040;      // for dma_port_to_mem
                            0x01000050;      // for dma_copy_mem

    dma->control.secondary = 0;
    dma->source = (int)src;
    dma->destination = (int)dest;
    shadowcount.reg.frame = 1;
    shadowcount.reg.element = count;
    dma->count = shadowcount.word;

    // Start data move
    dma->control.primary |= 1;

    // If blocking active, wait for complete
    if (block)
        while (!dma_done(channel))
            ;
}
```

Fig. 4.9: Zoom in II supported DMA library.

4.5.1 Data Transfer Flow

Much description given in this subsection is taken from [34] and [36]. This subsection describes the data transfer flow through FIFO using two blocks.

Much high-speed data converters cannot be connected directly to a digital signal processor (DSP). The required transfer rates would tie up to most of the DSP's I/O bandwidth (refer to [36] for bandwidth calculation). A FIFO is one of solutions for this problem because it can buffer a large block of data, and the DSP can read data from the FIFO in a burst mode. This is more efficient compared to single reads for every sampled value.

Depending on the application, there are numerous possibilities for software control. In most applications, different data blocks are used; one or more blocks for CPU to work on, and other blocks that are used for the data transfer between the EDMA/DMA controller and the FIFOs.

It is possible for the TMS320C6x DSP to read in one data block from the FIFO and to write out another block in the same time frame. In this case, the DMA would automatically switch between the two transfer channels. But, this causes delays every time the EMIF switches from input to output mode. Therefore, for best performance, only one block transfer (read or write) should be active.

Debugging a DSP system with FIFOs is not an easy work, especially if DSP is halted for emulation. Fig. 4.10 shows an example of a simple data-transfer method for an application in which the ADC and DAC run at the same clock speed. Only two data blocks are used in this example, data block A and data block B [34].

4.5.2 Memory Constraints

Much description given in this subsection is taken from [8] and [44]. This subsection describes a key point for so-called "different data blocks" to avoid performance degradation.

As shown in Fig. 4.11 [44], internal memory is arranged through various banks of memory so that loads and stores can occur simultaneously. Since each bank of memory is

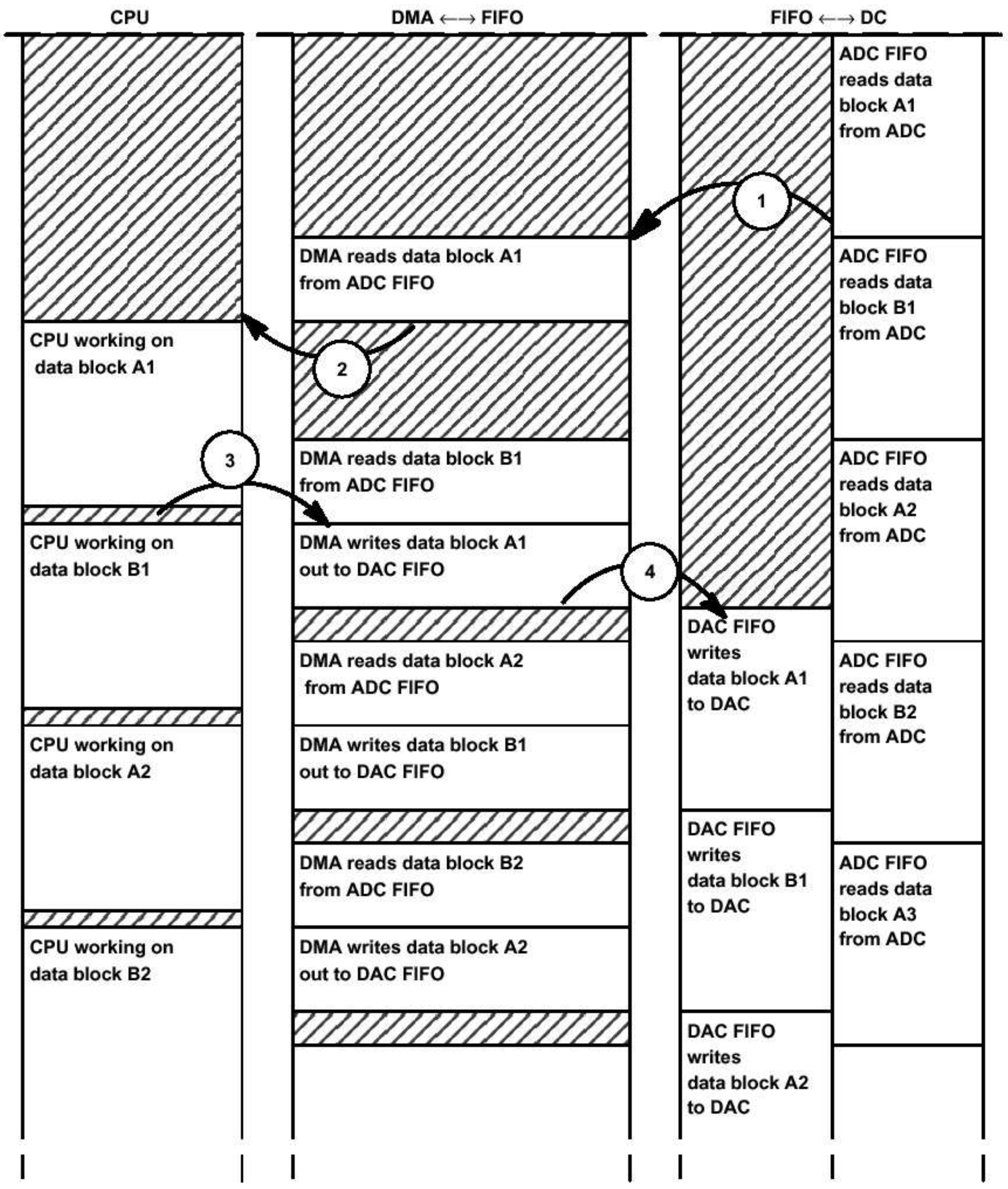

CPU
DMA ⟷ FIFO
FIFO ⟷ DC
ADC FIFO reads data block A1 from ADC
1
DMA reads data block A1 from ADC FIFO
ADC FIFO reads data block B1 from ADC
CPU working on data block A1
2
3
DMA reads data block B1 from ADC FIFO
ADC FIFO reads data block A2 from ADC
CPU working on data block B1
DMA writes data block A1 out to DAC FIFO
4
DAC FIFO writes data block A1 to DAC
DMA reads data block A2 from ADC FIFO
ADC FIFO reads data block B2 from ADC
CPU working on data block A2
DMA writes data block B1 out to DAC FIFO
DAC FIFO writes data block B1 to DAC
DMA reads data block B2 from ADC FIFO
ADC FIFO reads data block A3 from ADC
CPU working on data block B2
DMA writes data block A2 out to DAC FIFO
DAC FIFO writes data block A2 to DAC

Fig. 4.10: Software flow in a simple application (from [34]).

single-ported, only one access to each bank is performed per cycle. Two memory accesses per cycle can be performed if they do not access the same bank of memory. If multiple accesses are performed to the same bank of memory (within the same space), the pipeline will stall. This causes additional cycles for execution to complete [8].

The 64 Kbytes of data memory of the 'C6x is organized into two blocks of 32 Kbytes: the TMS320C6701 have eight banks per block, and the TMS320C6201 have four banks per block [44]. Therefore, in order to upgrade performance, we need to set separate memory blocks between CPU and DMA, and separate memory banks between CPU 2 sides (side A and B).

The so-called "different data blocks" need to be arranged into to separate memory blocks. For example, in the linker command file (.cmd) of the existing simulator and other structures observed previously, we assign both dual buffers located into the ".my" memory space. It is an improper assignment for double buffering. It answers why the unexpected time-consumption happens. Therefore, we split ".my" into ".my0" and ".my1" for different 32-Kbytes blocks separately. Then, after we apply correctly dual buffers into ".my0" and ".my1", respectively, the pipelining between CPU and DMA reach really.

4.5.3 DMA Channels

In previous subsection, we reach pipelining between CPU and DMA after corrected memory arrangement for dual buffers. There are four independent DMA channels for TI DSP chip 'C6201 or 'C6701. Furthermore, in this subsection, we want to pipeline these four channels. However, only one timing-sharing DMA bus exists. A simple code and vendor's email reply show that these DMA channels could not move data simultaneously. We explain the simple code as below, and append the email in the appendix.

After zoom in II's library of dma_port_to_mem, dma_mem_to_port, and dma_copy_mem, we understand they use the same transfer mechanism, just different sources and destinations. The dma_port_to_mem and dma_mem_to_port do DMA transfer via FIFO buffer be-

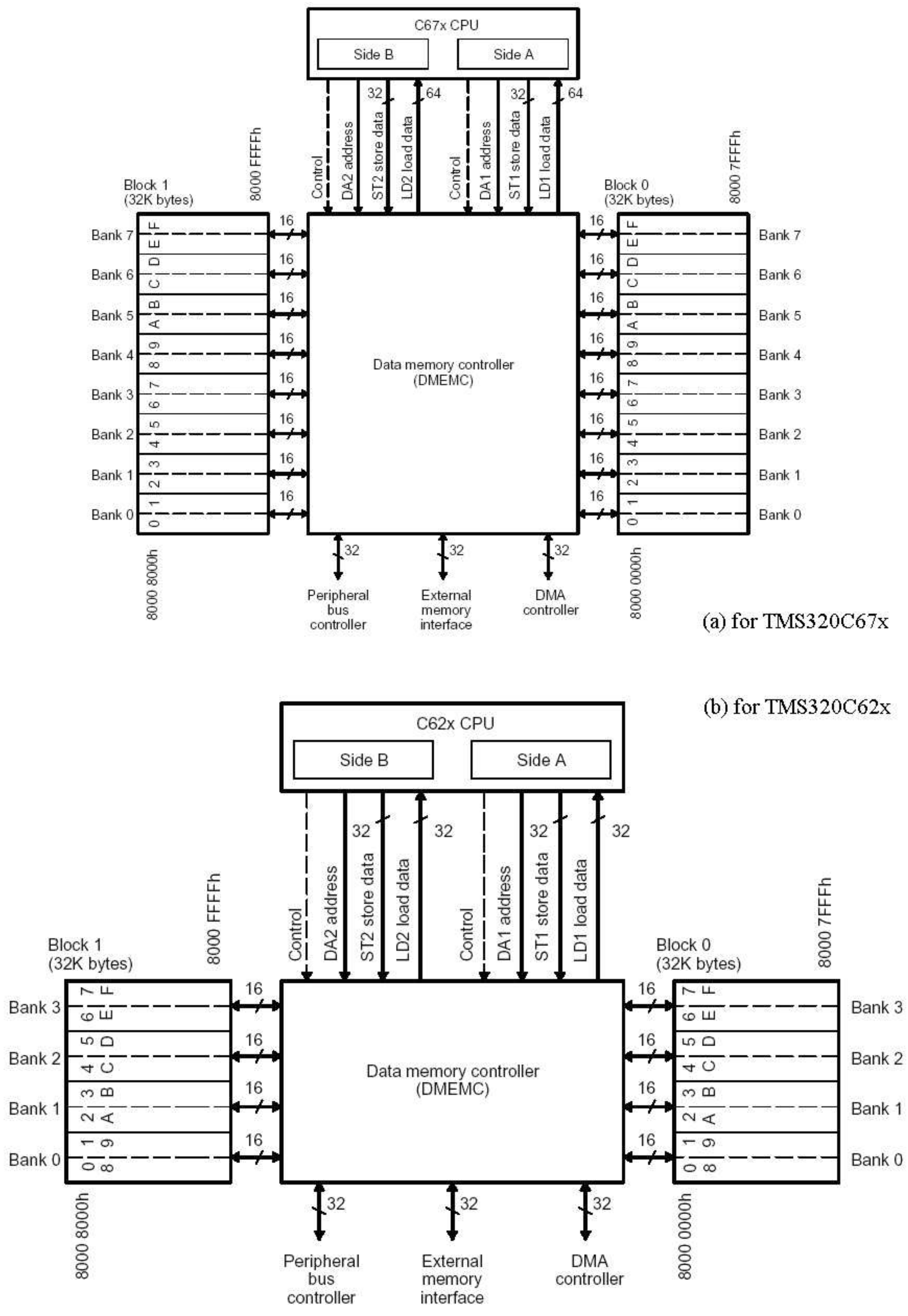
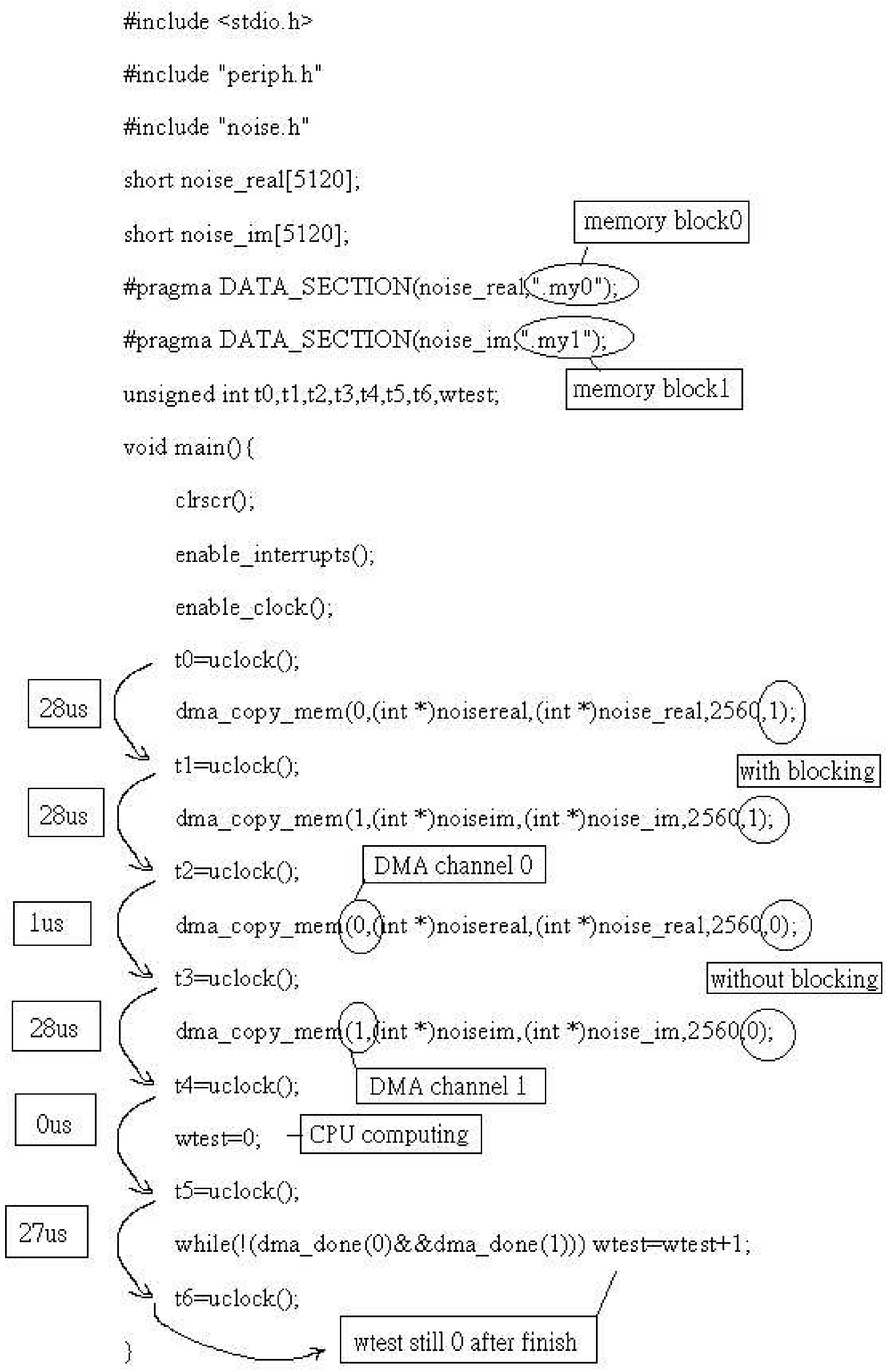


Fig. 4.11: ’C6x data memory controller interconnect to memory banks (from [44]).



```
#include <stdio.h>
#include "periph.h"
#include "noise.h"
short noise_real[5120];
short noise_im[5120];
#pragma DATA_SECTION(noise_real,".my0");
#pragma DATA_SECTION(noise_im,".my1");
unsigned int t0,t1,t2,t3,t4,t5,t6,wtest;
void main(){
      clrscr();
      enable_interrupts();
      enable_clock();
      t0=uclock();
      dma_copy_mem(0,(int *)noisereal,(int *)noise_real,2560,1);
      t1=uclock();
      dma_copy_mem(1,(int *)noiseim,(int *)noise_im,2560,1);
      t2=uclock();
      dma_copy_mem(0,(int *)noisereal,(int *)noise_real,2560,0);
      t3=uclock();
      dma_copy_mem(1,(int *)noiseim,(int *)noise_im,2560,0);
      t4=uclock();
      wtest=0;
      t5=uclock();
      while(!(dma_done(0)&&dma_done(1))) wtest=wtest+1;
      t6=uclock();
}
```

Fig. 4.12: A simple code showing that DMA channels could not move data simultaneously.

tween two processors. On the other hand, dma_copy_mem does DMA movement directly from external to internal memory on single processor. In other words, dma_copy_mem does not need to go through FIFO buffer. Here, we want to avoid FIFO that may confuse our observation; we just want to check DMA to see if we set two channels at two separate memory blocks, then whether DMA channels move at the same time or not. Therefore, it is simpler to observe DMA by dma_copy_mem in the simple code on single processor than dma_port_to_mem or dma_mem_to_port between processors.

In the source code of dma_copy_mem, there is a while loop in the last line to monitor whether dma_done is ready or not. If we set 1 for blocking, then this while loop blocks next command until DMA is done, which is the sequential running style. If we set 0 for non-blocking, then it skips the blocking of dma_done-related while loop to execute immediately next command. If resource confliction does not happen, then it is possible to pipeline by insert some code between non-blocking dma_copy_mem and dma_done-related while loop.

In the simple code, we set 1 for blocking in the first and second dma_copy_mem, so it is about 28 μs to move by DMA channel 0 and 1 respectively. So, we set 0 for non-blocking in the third and fourth dma_copy_mem. Then we know that channel 1 does DMA move sequentially after channel 0. Even separate memory blocks are set for two channels respectively, the simple code shows that DMA channels could not move data simultaneously.

For the third dma_copy_mem, the 1 μs between t2 and t3 is due to non-blocking mode. DMA start movement after informing CPU, and then CPU proceeds immediately to the next command. Note that both the DMA channel 0 movements is on-going and CPU is running. The 28 μs between t3 and t4 means that the fourth dma_copy_mem delays this 28 μs to execute DMA channel 1 after the DMA channel 0 movement from third dma_copy_mem until around t4. The 0 μs between t4 and t5 means that CPU finishes the initial setting of wtest fast and the DMA is ongoing in background until finish. Both the

Table 4.15: Improved Actual Performance

	Modulation & Channel	Matched Filter	Run Time per Slot	Memory Amount
Original Version (the Existing Simulator)	Sequential-10240 (a slot each time)	9-tap Sequential-512	2637 μs (3.96x666)	10240 samples
Proposal Version (Run on the Quatro67)	Pipelining-512	9-tap Pipelining-512	1170 μs (1.75x666)	1024 samples
Proposal Version (Run on the Quatro62)	Pipelining-512	9-tap Pipelining-512	940 μs (1.41x666)	1024 samples

wtest still zero after finish and the 27 μs between t5 and t6 means that this 27 μs primarily comes from this while loop stalling for about 27 μs until DMA channel 1 is done.

4.6 Summary

In a system of multiprocessors, DSP program development may become very complex. Because we not only need to code program for individual DSPs respectively, but also need to consider data transmission and handshaking among closely coupling multiprocessors. Unexpected behaviors arose in the actual performance of our initial multiprocessor implementation. Two key factors resulted in the extra run time; one is waiting for FIFO handshaking status to get ready, and the other is waiting for conflicting DSP resources to be released, such as same memory bank bus or only one DMA bus. Double buffering scheme could efficiently achieve that one block memory is used in computing while another block is used for data move by one time-sharing DMA bus. As shown in Table 4.15, the actual run time per slot became 940 μs, which improved about 86.1% toward 666 μs of real-time criterion from 2637 μs of original version. In addition, the memory used amount reduced about 90%, to compare original 10240 samples with proposal dual 512 samples.

Chapter 5

Fixed-Point Arithmetic on TMS320C62

5.1 Overview

This chapter is concerned with identifying efficient digital algorithms for the generation of elementary functions including sine, cosine, natural logarithm and square-rooting, with a goal towards their suitability for fixed-point DSP implementation, to serve as the generation of channel coefficients and Gaussian noise discussed in chapter 2. The problem arises because we want to port our channel simulator to fixed-point DSP board, but these four elementary functions, in fixed-point arithmetic, are not available yet in TI TMS320C6x or II Quatro6x library. We consider using CORDIC (COordinate Rotation DIgital Computer) and CCM (Convergence Computation Method) algorithms to effect these four elementary functions in integer arithmetic on a fixed-point TI DSP TMS320C6201.

5.2 Methods of Channel Model Generation

The existing simulator [32] almost finishes fixed-point implementation, except generation of channel coefficients and Gaussian noise, for lack of efficient fixed-point library. In this section, we discuss briefly several methods for channel coefficients and noise generation.

5.2.1 Fading Channel Coefficients

Tsai [32] point out the performance of the fading channel is not so good because the programs have to call mathematic functions to generate channel coefficients. Therefore, we need to employ efficient cosine and sine functions for improving fading channel performance.

A number of different models have been proposed for the simulation of Rayleigh fading channels over the past decades. Generally, these models can be classified as either being statistical or deterministic [22]. One is Doppler filtering complex Gaussian variables [14]; another is Jakes' model or its modification. The classical Jakes' model is a deterministic method to approximate fading channel, which has been widely used for about three decades because of its simplicity. However, the signal generated by the classical Jakes' simulator is not wide-sense stationary (WSS). Pop and Beaulieu [30] proposed a WSS simulator by introducing random phase shifts in the low-frequency oscillators to remove the stationary problem, but some problems with higher-order statistics remain. A WSS Jakes' model with more correct statistical properties proposed by Zheng and Xiao in [48] reintroduces the randomness to the path gain, the Doppler frequency, and the initial phase of the sinusoids. In [47], the authors analyze the statistical properties of the improved Jakes' simulator proposed by Li and Huang [22], which can generate multiple independent Rayleigh faders easily but need random phases. The point is that the Jakes' model requires efficient sine and cosine functions, besides, the other methods above also require an efficient random number generation.

5.2.2 Gaussian Random Number

Gaussian random number generators are employed to simulate the fading phenomena and additive white Gaussian noise of the radio channel. High speed Gaussian random number generators are most important components for the real-time channel simulation of the CDMA systems, because of channel's wideband nature [5].

In general, two kinds of methods used to compute Gaussian noise. One is via the central limit theorem; another is the Box-Muller method or the polar method. The Box-Muller method generates Gaussian noise by mapping two uniform distribution random sequences using four elementary functions including sine, cosine, logarithm, and square rooting [21]. The polar method does the same mapping operation in alternative formula form without sine and cosine functions [31]. The central limit theorem requires enough samples size to form Gaussian distribution. A technique of the probability density conversion (PDC) before addition [5] omit some of the input random numbers and adders. The sum-of-12 method [20] compromises 12 samples between speed and accuracy.

5.3 Introduction to CORDIC and CCM

In this section, we describe the basic theory of CORDIC (COordinate Rotation DIgital Computer) and CCM (Convergence Computation Method). They are both iterative techniques, which are alike in some way. The generalized convergence computation method [4] generalizes Chen's CCM [10] to multidimensional quantities, resulting in a unified algorithm for elementary function generation of which CORDIC and CCM are special cases.

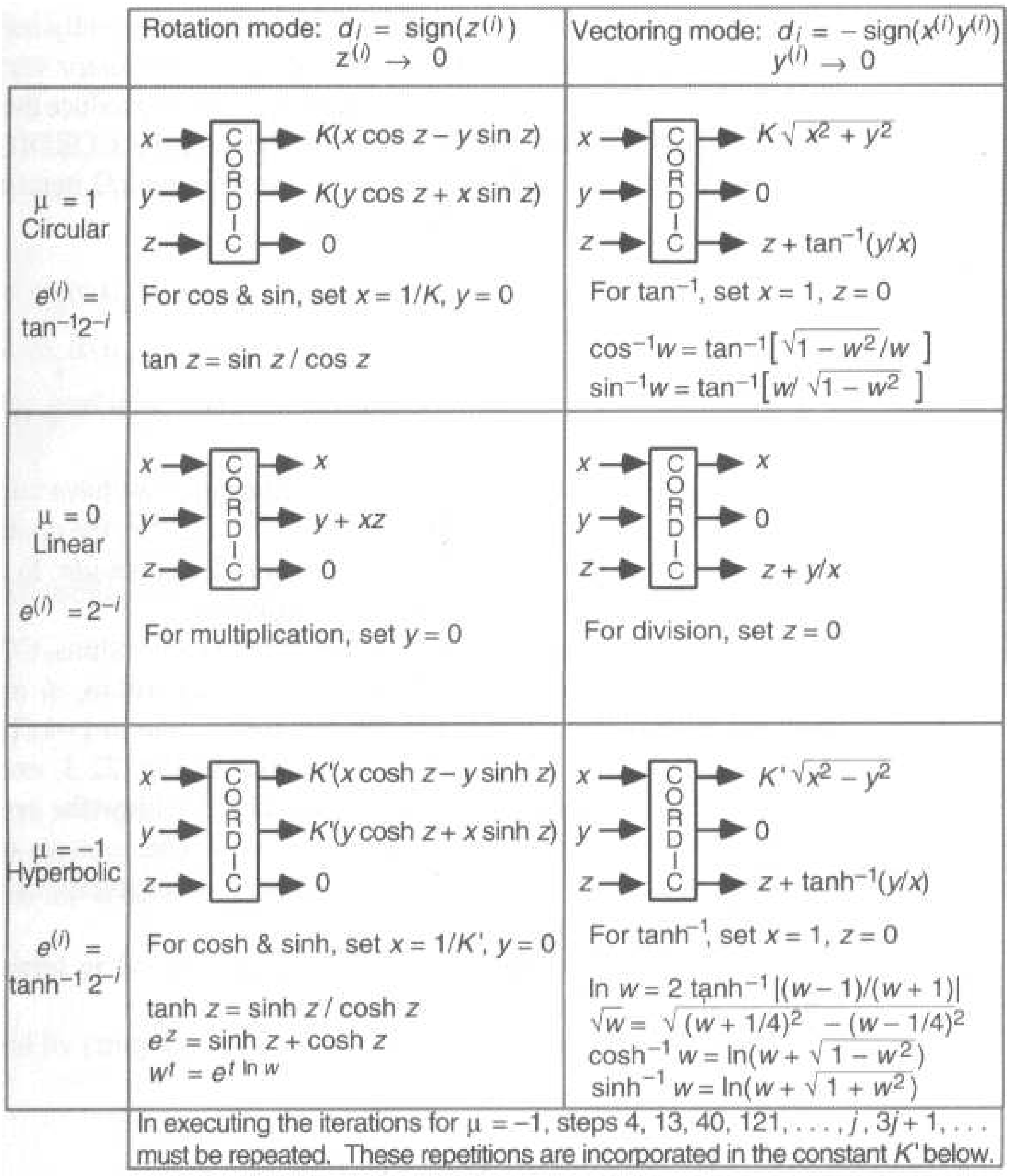
5.3.1 The CORDIC Method

The following introduction to CORDIC is largely taken from [6], [17], [27], [28], [29], [25], and [16]. CORDIC is an iterative algorithm invented by Volder [45] and refined by Walther [46]. As shown in Table 5.1, the functions that can be evaluated using CORDIC are trigonometric functions ($\sin$, $\cos$, $\tan$), inverse trigonometric functions ($\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$), hyperbolic functions ($\sinh$, $\cosh$, $\tanh$), inverse hyperbolic functions ($\sinh^{-1}$, $\cosh^{-1}$, $\tanh^{-1}$), exponent, natural logarithm, square-rooting, multiplication, division, and vector magnitude. The CORDIC algorithm has found a wide range of applications.

Weighted sum of rotation angles

The basic concept of the CORDIC computation is to decompose the desired rotation angle into the weighted sum of a set of predefined elementary rotation angles such that the rotation through each of them can be accomplished with simple shift-and-add operations.

Table 5.1: Summary of Generalized CORDIC Algorithms [27]



	Rotation mode: $d_i = \text{sign}(z^{(i)})$, $z^{(i)} \to 0$	Vectoring mode: $d_i = -\text{sign}(x^{(i)}y^{(i)})$, $y^{(i)} \to 0$
$\mu = 1$ Circular $e^{(i)} = \tan^{-1}2^{-i}$	$\begin{tikzcd} x \arrow[r] & \text{CORDIC} \arrow[r] & K(x\cos z - y\sin z) \\ y \arrow[r] & \text{CORDIC} \arrow[r] & K(y\cos z + x\sin z) \\ z \arrow[r] & \text{CORDIC} \arrow[r] & 0 \end{tikzcd}$ For cos & sin, set $x = 1/K$, $y = 0$; $\tan z = \sin z / \cos z$	$\begin{tikzcd} x \arrow[r] & \text{CORDIC} \arrow[r] & K\sqrt{x^2 + y^2} \\ y \arrow[r] & \text{CORDIC} \arrow[r] & 0 \\ z \arrow[r] & \text{CORDIC} \arrow[r] & z + \tan^{-1}(y/x) \end{tikzcd}$ For $\tan^{-1}$, set $x = 1$, $z = 0$; $\cos^{-1}w = \tan^{-1}[\sqrt{1-w^2}/w]$; $\sin^{-1}w = \tan^{-1}[w/\sqrt{1-w^2}]$
$\mu = 0$ Linear $e^{(i)} = 2^{-i}$	$\begin{tikzcd} x \arrow[r] & \text{CORDIC} \arrow[r] & x \\ y \arrow[r] & \text{CORDIC} \arrow[r] & y + xz \\ z \arrow[r] & \text{CORDIC} \arrow[r] & 0 \end{tikzcd}$ For multiplication, set $y = 0$	$\begin{tikzcd} x \arrow[r] & \text{CORDIC} \arrow[r] & x \\ y \arrow[r] & \text{CORDIC} \arrow[r] & 0 \\ z \arrow[r] & \text{CORDIC} \arrow[r] & z + y/x \end{tikzcd}$ For division, set $z = 0$
$\mu = -1$ Hyperbolic $e^{(i)} = \tanh^{-1}2^{-i}$	$\begin{tikzcd} x \arrow[r] & \text{CORDIC} \arrow[r] & K'(x\cosh z - y\sinh z) \\ y \arrow[r] & \text{CORDIC} \arrow[r] & K'(y\cosh z + x\sinh z) \\ z \arrow[r] & \text{CORDIC} \arrow[r] & 0 \end{tikzcd}$ For cosh & sinh, set $x = 1/K'$, $y = 0$; $\tanh z = \sinh z / \cosh z$; $e^z = \sinh z + \cosh z$; $w^t = e^{t \ln w}$	$\begin{tikzcd} x \arrow[r] & \text{CORDIC} \arrow[r] & K'\sqrt{x^2 - y^2} \\ y \arrow[r] & \text{CORDIC} \arrow[r] & 0 \\ z \arrow[r] & \text{CORDIC} \arrow[r] & z + \tanh^{-1}(y/x) \end{tikzcd}$ For $\tanh^{-1}$, set $x = 1$, $z = 0$; $\ln w = 2\tanh^{-1}\|(w-1)/(w+1)\|$; $\sqrt{w} = \sqrt{(w+1/4)^2 - (w-1/4)^2}$; $\cosh^{-1}w = \ln(w + \sqrt{1-w^2})$; $\sinh^{-1}w = \ln(w + \sqrt{1+w^2})$
	In executing the iterations for $\mu = -1$, steps 4, 13, 40, 121, . . . , j, $3j+1$, . . . must be repeated. These repetitions are incorporated in the constant K' below.	

$$x^{(i+1)} = x^{(i)} - \mu d_i (2^{-i} y^{(i)}) \qquad \mu \in \{-1, 0, 1\},\ d_i \in \{-1, 1\}$$
$$y^{(i+1)} = y^{(i)} + d_i (2^{-i} x^{(i)}) \qquad K = 1.646\,760\,258\,121\ldots$$
$$z^{(i+1)} = z^{(i)} - d_i e^{(i)} \qquad K' = 0.828\,159\,360\,960\,2\ldots$$

The angle of rotation θ is approximated by the sum of the N elementary rotations as

$$\theta \simeq \sum_{i=0}^{N-1} d_i e(i),$$

where N is the number of rotations, $e(i)$ is the predefined elementary rotation angle of the ith rotation, and d_i is a sequence of ± 1s which determines the direction of the remaining angle.

Unified CORDIC iteration equations

Walther [46] summarizes the algorithm using a set of unified CORDIC iteration equations as

$$\begin{bmatrix} x(i+1) \\ y(i+1) \end{bmatrix} = \begin{bmatrix} 1 & -\mu d_i 2^{-S(\mu,i)} \\ d_i 2^{-S(\mu,i)} & 1 \end{bmatrix} \begin{bmatrix} x(i) \\ y(i) \end{bmatrix},$$

$$z(i+1) = z(i) - d_i e(\mu, i),$$

where $x(i)$ and $y(i)$ are the x and y components of the interested vector, $z(i)$ is the residue rotation angle, and the constants μ, d_i, $e(\mu, i)$, and $S(\mu, i)$ depend on the specific computation being performed, as explained below.

Coordinate system parameter

The coordinate system parameter μ is either 1, 0, or -1 corresponds to, respectively, the rotation operation in a circular coordinate system, a linear coordinate system, and a hyperbolic coordinate system. $\mu = 1$ is used for trigonometric and inverse trigonometric fuctions, $\mu = 0$ is used for multiplication and division, and $\mu = -1$ is used for hyperbolic, inverse hyperbolic, expontential and logarithmic functions, as well as square roots.

Modes of operations

In the algorithm equations, an auxiliary variable $z(i)$ serves the purpose of accumulating the step angles and determining the sign of the next step rotation. The rotations of the CORDIC algorithm are usually carried out in two modes, called *rotation* and *vectoring*.

In the rotation mode, which is also known as the forward rotation mode, the input vector is rotated by a given angle θ, the angle accumulator $z(i)$ is initialized with the rotation angle and rotation at each iteration is aimed at making the angle accumulator converge towards zero. The objective is to compute the final coordinate.

In the vectoring mode, which is also known as the Y-reduction mode, or the backward rotation mode, the desired angle θ is not given. The input vector is rotated to the x-axis through whatever angle is necessary to align the result vector with the x-axis. The result is a rotation angle and the scaled magnitude of the original vector.

Rotation direction parameter

The rotation direction parameter d_i requires only a comparison. It is one of the following two signum functions:

$$d_i = \begin{cases} \operatorname{sgn}(z(i)), & \text{for} \quad z(i) \to 0 \quad \text{in rotation mode}, \\ -\operatorname{sgn}(x(i)y(i)), & \text{for} \quad y(i) \to 0 \quad \text{in vectoring mode}, \end{cases}$$

where $\operatorname{sgn}(x) = +1$ for $x \geq 0$ and $\operatorname{sgn}(x) = -1$ for $x < 0$. The elementary rotations can be positive or negative depending on the direction of rotation, denoted by $d_i \in \{-1, 1\}$.

Elementary rotation angle

The ith elementary rotation angle $e(\mu, i)$, which is extended from $e(i)$, is given by constants stored in a lookup table which depend on the value of μ. It is an acrtangent, a power of two, or a hyperbolic arctangent, for $\mu = 1$, 0, or -1 correspondingly:

$$e(\mu, i) = \frac{1}{\sqrt{\mu}} \tan^{-1}[\sqrt{\mu} 2^{-S(\mu,i)}] = \begin{cases} \tan^{-1} 2^{-S(1,i)}, & \mu = 1, \\ 2^{-S(0,i)}, & \mu \to 0, \\ \tanh^{-1} 2^{-S(-1,i)}, & \mu = -1. \end{cases}$$

Scale factor

The norm of a vector $[\ x \quad y\ ]^t$ in these three coordinate systems are defined as $\sqrt{x^2 + \mu y^2}$. For $\mu = 0$, the rotation has a unity gain, so no scaling operation is needed. For $\mu \neq 0$, the rotation changes the norm of the vector. The scale factor $K_\mu(N)$ is given by

$$K_\mu(N) = \prod_{i=0}^{N-1} \sqrt{1+\mu d_i^2 2^{-2S(\mu,i)}} = \begin{cases} \prod_{i=0}^{N-1} \sqrt{1+4^{-i}} & \equiv K, \quad \mu = 1, \\ \prod_{i=0}^{N-1} \sqrt{1-4^{-S(-1,i)}} & \equiv K', \quad \mu = -1, \end{cases}$$

where K is an expansion factor and K' is a shrinkage factor. For $N > 4$, $K \approx 1.64676$ and $K' \approx 0.82816$. The factor $K_\mu(N)$ is dependent on the total number of iterations N, the shift sequence $S(\mu, i)$ and a function of d_i. Once a shift sequence $S(\mu, i)$ is decided and fixed with $d_i \in \{-1, 1\}$, $K_\mu(N)$ is constant which can be pre-calculated. For applications where unity-gain rotation is required, normalization using a multiplication by $1/K_\mu(N)$ imposes a computation overhead. As long as the total count of rotations N is fixed, making the norm of initial coordinate exactly a constant $1/K_\mu(N)$, the rotation produces the unscaled operation for final coordinate.

Shift sequence

$S(\mu, i)$ is a non-decreasing integer shift sequence which is usually determined in advance. The shift sequence determines the magnitude of scaling factor $K_\mu(N)$, as well as the convergence of the CORDIC iteration. For $\mu = 1$ or 0, $\{S(\mu, i) = i, 0 \le i \le N-1\}$. For hyperbolic CORDIC iterations with $\mu = -1$, ensuring convergence is a bit more tricky, since whereas $\tan^{-1}(2^{-(i+1)}) \ge 0.5 \tan^{-1}(2^{-i})$, the corresponding relation for $\tanh$, namely, $\tanh^{-1}(2^{-(i+1)}) \ge 0.5 \tanh^{-1}(2^{-i})$, does not hold in general [27]. It has been shown [46] that it is sufficient to repeat those iterations whose index $i = 4, 13, 40, \cdots, j, 3j+1, \cdots$. It is also an available shift sequence to repeat $i = 4, 7, 11, 14, 16, 18, \cdots$ for approximating hyperbolic arguments [29].

Convergence range

The CORDIC equations impose a limited domain upon the arguments for the evaluated mathematical functions. That is, the given rotation angle θ must not exceed the convergence range of the iteration, θ_{max}, which is given by the sum of all rotation angles plus the final angle as

$$\theta \le \sum_{i=0}^{N-1} e(\mu, i) + e(\mu, N-1) \equiv \theta_{max} \simeq \begin{cases} 1.7433\ (99.9^\circ), & \mu = 1, \\ 1, & \mu = 0, \\ 1.1182, & \mu = -1. \end{cases}$$

CORDIC can evaluate e^x only for x-values between 0.0 and 1.2364, and ln(x) only for x-values between 1.0 and 9.5149. Expanding the range of convergence of CORDIC is discussed in [16].

Precision

CORDIC is a digit-by-digit computation algorithm. It has the characteristic of linear convergence. That is, if we want to obtain n bit precision, then we need n iterations.

Unified CORDIC summary

Table 5.1 provides a summary of CORDIC. It also contains formulas for indirectly computing some other functions. For example, the $\tan$ function can be computed by first computing $\sin$ and $\cos$ and then perform a division, perhaps through another set of linear CORDIC iterations [27].

5.3.2 Convergence Computation Method

The following introduction to CCM is largely taken from [4], [9], and [15]. The convergence computation method (CCM) developed by Chen [10] is an algorithm that successively converges to the desired results. CCM computes exponential, logarithmic, and inverse square root functions, which are not directly available in the CORDIC algorithm. Suppose that we wish to evaluate a function

$$z_0 = f(x)_{x=x0} = f(x_0).$$

We introduce an auxiliary variable y to form the convergence function $F(x, y)$ that satisfies the following three properties:

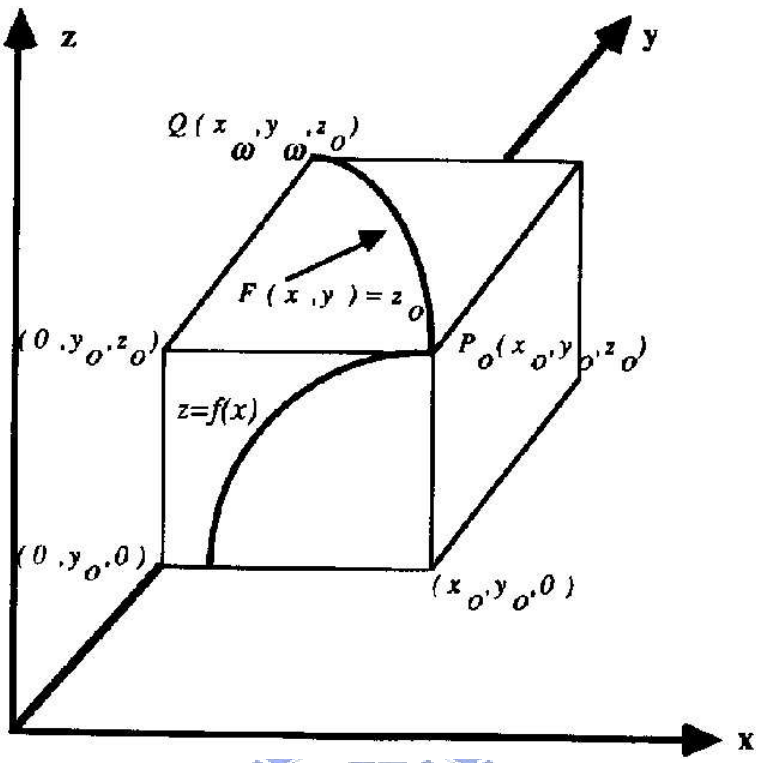


Fig. 5.1: A geometric interpretation of CCM (from [4]).

1. There exists a known initial value $y = y_0$ such that $z_0 = F(x_0, y_0)$.
2. There exists a convenient transformation of (x_k, y_k) into (x_{k+1}, y_{k+1}) such that $F(x_{k+1}, y_{k+1})$ is invariant for all $k \geq 0$.
3. A known destination value, x_w, is reached through the sequence of x-transformations such that the resulting y-transformations convergence to $y_w = F(x_w, y_w) = z_0$.

The transformation rule involves selecting a pair of functions φ and ψ such that

$$\begin{cases} x_{k+1} = \varphi(x_k, y_k), \\ y_{k+1} = \psi(x_k, y_k). \end{cases}$$

Fig. 5.1 provides a a geometric interpretation of CCM. The function $F(x, y)$ is constrained to lie in the plane $z = z_0$ of a three-dimensional cube which has a vertex at $P_0 = (x_0, y_0, z_0)$. The invariance of $F(x, y)$ implies that at each iteration, the point

$P_k = (x_k, y_k, z_k)$ lies on the curve $F(x, y)$, i.e.,

$$z_0 = F(x_0, y_0) = F(x_k, y_k) = \cdots = F(x_w, y_w) = y_w.$$

Furthermore, the curve $F(x, y)$ must pass through the point $Q = (x_w, y_w, z_0)$ as a consequence of the third condition.

The CCM can be readily understood by an example. Consider the computation of the exponential function $f(x) = we^x$. Define the convergence function to be $F(x, y) = ye^x$ with initial value $y_0 = w$ and destination $x_w = 0$. We see that $z_0 = F(x_0, y_0)$ as is required by condition 1. Next choose the transformations:

$$\begin{cases} x_{k+1} = \varphi(x_k, y_k) = x_k - \ln a_k, \\ y_{k+1} = \psi(x_k, y_k) = y_k a_k. \end{cases}$$

Now $F(x, y)$ is clearly invariant since

$$F(x_{k+1}, y_{k+1}) = a_k y_k e^{x_k - \ln a_k} = y_k e^{x_k} = F(x_k, y_k).$$

Finally, when $x \to 0$, $\sum \ln a_k \to x_0$, and thus

$$y_w \to y_0 \prod a_k = w \prod a_k = we_0^x.$$

The CCM algorithms for exponential and logarithm are as follows [15].

1. Exponential ($0 \le x < \ln 2$) :

   $$\begin{aligned} z_0 &= we^{x_0} &&= y_0 e^{x_0} \\ &= (y_0 a_0) e^{x_0 - \ln a_0} &&= y_1 e^{x_1} &&= \cdots \\ &= y_n e^{\mu} &&\simeq y_n + y_n \mu \end{aligned}$$

   Initialization: $y_0 = w$

   Function: $F(x, y) = ye^x$

   Transformation: $x_{k+1} = x_k - \ln a_k$ and $y_{k+1} = y_k a_k$

   Termination: $x_n = \mu$ and $z_0 \simeq y_n + y_n \mu$

   Sequence: $a_k = 1 + 2^{-m}$

   where m is the leading-1 bit position of the x_k value.

2. Logarithm (1/2 $\leq x <$1) :

$$\begin{aligned} z_0 &= w + \ln x_0 \quad = y_0 + \ln x_0 \\ &= (y_0 - \ln a_0) + \ln x_0 a_0 \quad = y_1 + \ln x_1 \quad = \cdots \\ &= y_n + \ln(1-\mu) \quad \simeq y_n - \mu \end{aligned}$$

Initialization: $y_0 = w$

Function: $F(x, y) = y + \ln x$

Transformation: $x_{k+1} = x_k a_k$ and $y_{k+1} = y_k - \ln a_k$

Termination: $1 - x_n = \mu$ and $z_0 \simeq y_n - \mu$

Sequence: $a_k = 1 + 2^{-m}$

where m is the leading-1 bit position of the $1 - x_k$ value.

The choice of a_k affects the algorithm convergence. Chen advocates $a_k = 1 + 2^{-m}$ which guarantees convergence, where the quantity m is elaborated in [10]. Furthermore, the multiplications by a_k can be replaced by shift-and-add kernels. In fact, x approaches x_w, but not exactly equal to x_w. That is, $x_n = x_w \pm \mu$, where $\mu \rightarrow$ 0 but $\mu \neq$ 0. We need k iterations to obtain k-bit precision.

5.4 Function Evaluation in Integer Arithmetic

For evaluation of sine, cosine, square-rooting and natural logarithm, we apply CORDIC and CCM algorithms in integer arithmetic on a fixed-point TI DSP TMS320C6201. This section presents the program development with aid of example.

5.4.1 Cosine and Sine by CORDIC

We follow CORDIC to develop a fixed-point program for calculation of sine and cosine.

CORDIC example for sine and cosine

Table 5.2 shows all the details of the simultaneous calculation of $\cos(70°)$ and $\sin(70°)$. Each row of the table shows the values of z, x, y, and d_i at the end of a computation cycle.

Table 5.2: CORDIC Example for $\sin(70°)$ and $\cos(70°)$ [29]

i	z	x	y	d_i	$e(i)$
0	70.00000000	0.60725294	0.00000000	1	45.00000000
1	25.00000000	0.60725294	0.60725294	1	26.56505118
2	-1.56505118	0.30362647	0.91087940	-1	14.03624347
3	12.47119229	0.53134632	0.83497279	1	7.12501635
4	5.34617594	0.42697472	0.90139108	1	3.57633437
5	1.76984157	0.37063778	0.92807700	1	1.78991061
...	...	...	...	...	...
15	-0.00264619	0.34197674	0.93970842	-1	0.00174853
16	-0.00089767	0.34200542	0.93969798	NA	NA

Each row of the table is produced by looking back at the previous row. Once the sign of d_i is set by the sign of the old value of z, all else follows. Except for $e(i)$, each row of the table replaces the previous row in the calculator memory [29].

CORDIC program for sine and cosine

As shown in Fig. 5.2, the program is presented in circular CORDIC rototation mode. It has been scaled by a factor, such as 2^{23} or other choices, to use only integers for sine and cosine evaluation in fixed-point format. The sine of the desired angle is now present in the variable y and the cosine of the desired angle is in the variable x. We assume a 16-step system, which will yield 16 bits of accuracy. We set a constant 5094007 at initial x to omit the post-multiplication at final coordinate. This constant is pre-calculated by $1/K$ with 2^{23} scale, where $K = \prod_{i=0}^{N-1} \sqrt{1+4^{-i}} \simeq 1.646760258$ for $N = 16$. We also assume that $\tan^{-1}(2^{-i})$ have been calculated with scaling operation before run time and stored at a 16 entry lookup table. In Fig. 5.2, we use radian unit with 2^{23} scale to setup the table, so we call the *cordic* routine by corresponding scaled radian unit. If we want to call the routine in degree unit directly, just change content in the elementary angle table $e[16]$, and the desired angle *theta* by corresponding scaled degree value. Moreover, here we also call TI's $\cos$ and $\sin$ library of floating-point format for comparison later.

```
#include <math.h>
int e[16]={ // table of elementary rotation angle: atan(2^i) in rad *2^23 scale
6588392, 3889355, 2055028, 1043164, 523606, 262058, 131061, 65535, 32768,
16384, 8192, 4096, 2048, 1024, 512, 256 };
int theta,x,y,z; // theta: desired angle, x,y: interested vector, z: residue angle
cordic(theta){ // circular CORDIC rotation mode for sin and cos in fixed-point
        int dx,dy;
        unsigned int i;         // index of i-th iteration
        z= theta;               // set z to the desired angle *2^23
        x=5094007;              // set x to 1/ 1.6467602… *2^23
        y=0;                    // set initial vector to lie in x-axis
        for(i=0;i<16;i++){  // computation of 16 iterations for 16-bit precision
                dx=x>>i;        // x shifted right by i places to get x *2^-i for this step
                dy=y>>i;        // y shifted right by i places to get y *2^-i for this step
                if(z>=0){      // decides if next rotation be clockwise or anticlockwise
                        x=x-dy;         // compute x - y *2^-i
                        y=y+dx;         // compute y + x *2^-i
                        z=z-e[i];       // update the current angle
                } else{        // rotation in opposite direction
                        x=x+dy;
                        y=y-dx;
                        z=z+e[i]; }
}       }
void main(void){
        float cf,sf;                    // two floating-point variables
        cordic(10248609);               // call cordic of 1.2217294 rad with 2^23 scale
        cf=cos(1.2217294);              // call cos of 1.2217294 rad in floating-point
        sf=sin(1.2217294);              // call sin of 1.2217294 rad in floating-point
}
```

Fig. 5.2: CORDIC program for sine and cosine.

5.4.2 Square Root and Logarithm by CORDIC

We follow the hyperbolic CORDIC vectoring mode to develop a fixed-point program for evaluation of square root and natural logarithm.

CORDIC example for square root and natural logarithm

Table 5.3 shows the computation of $\tanh^{-1}(1/3)$, which is initialized by setting $z(0)$=0, $x(0)$=3, and $y(0)$=1. As y is driven toward 0, z is driven toward the answer of inverse hyperbolic tangent. The d_i=± 1 is chosen so that $d_i x(i)$ and $y(i)$ have opposite signs. This drives $y(i)$ toward 0. In the process, $z(i)$ is driven toward $\tanh^{-1}(y(0)/x(0))$.

Our interest in the values of inverse hyperbolic tangent stems from its connection with natural logarithms:

$$\ln(w) = 2\tanh^{-1}\left|\frac{w-1}{w+1}\right|.$$

In particular, the calculation above shows that $2\tanh^{-1}(1/3) = 2z(24) = 0.69315262$, a good approximation to ln(2)=0.69314718.... This is how CORDIC evaluates logarithms.

The significance of the terminal value $x(24) = 2.34231665$ is K' times the initial value of $\sqrt{x(0)^2 - y(0)^2}$, where $K' = \prod_{i=0}^{N-1}\sqrt{1-4^{-S(-1,i)}} \simeq 0.8281$ for $N = 24$. This is because the N-step folding process always magnifies the length of the initial segment by a shrinkage factor of K'. It is not hard to verify that by using the hyperbolic identities as

$$x(i+1)^2 - y(i+1)^2 = \frac{x(i)^2 - y(i)^2}{[\cosh^{-1}(e(-1,i))]^2}.$$

In other words, $\sqrt{3^2 - 1^2} = \sqrt{x(0)^2 - y(0)^2} = x(24)/K' \simeq 2.34231665/0.8281$. This is in fact how CORDIC evaluates square roots, as a byproduct of the calculation of inverse hyperbolic tangents.

If the inverse hyperbolic tangent algorithm is applied to the seed values $x(0) = w+1/4$ and $y(0) = w - 1/4$. Then $\sqrt{w} = x(N)/K'$, because $x(0)^2 - y(0)^2 = w$. In constrast to many of the other evaluations of elementary functions, it is interesting that this application of CORDIC does require a postmultiplication by $1/K'$ to complete [29].

Table 5.3: CORDIC Example for $\tanh^{-1}(1/3)$ [29]

i	z	x	y	d_i	$S(-1,i)$	$e(-1,i)$
0	0.00000000	3.00000000	1.00000000	-1	1	0.54930614
1	0.54930614	2.50000000	-0.50000000	1	2	0.25541281
2	0.29389333	2.37500000	0.12500000	-1	3	0.12565721
3	0.41955055	2.35937500	-0.17187500	1	4	0.06258157
4	0.35696898	2.34863281	-0.02441406	1	4	0.06258157
5	0.29438740	2.34710693	0.12237549	-1	5	0.03126018
6	0.32564758	2.34328270	0.04902840	-1	6	0.01562627
7	0.34127385	2.34251663	0.01241460	-1	7	0.00781266
8	0.34908651	2.34241964	-0.00588631	1	7	0.00781266
9	$\cdots$	$\cdots$	$\cdots$	$\cdots$	8	0.00390627
10	$\cdots$	$\cdots$	$\cdots$	$\cdots$	9	0.00195315
11	$\cdots$	$\cdots$	$\cdots$	$\cdots$	10	0.00097656
12	0.34615669	2.34231751	0.00097652	-1	11	0.00048828
13	0.34664497	2.34231703	-0.00016719	1	11	0.00048828
14	$\cdots$	$\cdots$	$\cdots$	$\cdots$	12	0.00024414
15	0.34640083	2.34231671	0.00040466	-1	13	0.00012207
16	0.34652290	2.34231666	0.00011873	-1	14	0.00006104
17	0.34658393	2.34231665	-0.00002423	1	14	0.00006104
18	$\cdots$	$\cdots$	$\cdots$	$\cdots$	15	0.00003052
19	$\cdots$	$\cdots$	$\cdots$	$\cdots$	16	0.00001526
20	$\cdots$	$\cdots$	$\cdots$	$\cdots$	16	0.00001526
21	$\cdots$	$\cdots$	$\cdots$	$\cdots$	17	0.00000763
22	$\cdots$	$\cdots$	$\cdots$	$\cdots$	18	0.00000381
23	0.34657249	2.34231665	0.00000258	-1	18	0.00000381
24	0.34657631	2.34231665	-0.00000636	NA	NA	NA

Table 5.4: CCM Example for Logarithm in Floating-Point [15]

x	$1-x$ [bin.]	m	a_k	$\ln(a_k)$	y
0.785	0. 001$\cdots$	3	$1+2^{-3}$	0.117783	0
0.883125	0. 0001$\cdots$	4	$1+2^{-4}$	0.060625	-0.117783
0.938320	0. 0000 1$\cdots$	5	$1+2^{-5}$	0.030772	-0.178408
0.967643	0. 0000 1$\cdots$	5	$1+2^{-5}$	0.030772	-0.209180
0.997882	0. 0000 0000 1$\cdots$	9	$1+2^{-9}$	0.001951	-0.239951
0.999831	0. 0000 0000 1$\cdots$	13	$1+2^{-13}$	0.000122	-0.241902
0.999953	0. 0000 0000 0000 001$\cdots$	15	$1+2^{-15}$	0.000031	-0.242024
0.999983	0. 0000 0000 0000 0001$\cdots$	16	$1+2^{-16}$	0.000015	-0.242055
0.999999	NA	NA	NA	NA	-0.242070

CORDIC program for square root and natural logarithm

As shown in Fig. 5.3, the program is presented in hyperbolic CORDIC vectoring mode. It has been scaled by a factor, such as 2^{23} or other choices, to use only integers for inverse hyperbolic tangent evaluation of fixed-point format. Here we assume a 24-step system, which will yield 24 bits of accuracy. The natural logarithm of desired input w is presented in the variable x. The square root of desired w is formed with scaling in the variable y. This scaling is to do post-multiplication at final x by factor of $1/(2K')$, where $1/K' \simeq 1.20753406$ is the product of hyperbolic cosine for 24 step, and the 2 combined with this factor is due to $\sqrt{(w+1)^2-(w-1)^2} = 2\sqrt{w}$ for more compatible input pair of $x = w-1$ and $y = w+1$. Moreover, the $\tanh^{-1}(2^{-S(-1,i)})$ have been calculated with 2^{23} scaling before run time and stored at a 24 entry lookup table. Note that the shift sequence $S(-1,i)$ is not the same as iteration index i; it is necessary to repeat at some iterations for convergence.

5.4.3 Natural Logarithm by CCM

We follow the convergence computing method to develop a fixed-point program for evaluation of natural logarithm.

```
#include <stdio.h>
#include <math.h>
unsigned int sh[24]={1, 2, 3, 4, 4, 5, 6, 7, 7, 8, 9, 10, 11, 11, 12, 13, 14, 14, 15, 16,
16, 17, 18, 18 };                          // shift sequence to repeat some iterations
unsigned int eh[24]={ 4607914, 2142558, 1054089, 524972, 524972, 262229,
131082, 65537, 65537, 32768, 16384, 8192, 4096, 4096, 2048, 1024, 512, 512,
256, 128, 128, 64, 32, 32 };               // table of atanh(2^-S(-1,i)) *(2^23)
int x,y,z,dy,dx;                           // x,y: interested vector, z: residue angle
vec_hyp_cordic(w){                         // hyperbolic CORDIC vectoring mode
        unsigned int i;                    // index of i-th iteration
        z=0;                               // z is initialized by setting 0
        y=(w-1)<<23;                       // y is given from an 2^23 scaled (w-1)
        x=(w+1)<<23;                       // x is given from an 2^23 scaled (w+1)
        for(i=0;i<24;i++){                 // 24 iterations for 24-bit precision
                dy=y>>sh[i];               // y(i) *2^-S(-1,i)
                dx=x>>sh[i];               // x(i) *2^-S(-1,i)
                if(y<0){                   // di(i)=+1
                        z=z-eh[i];         // z(i+1)=z(i)-di(i)e(-1,i)
                        x=x+dy;            // x(i+1)=x(i)+di(i)y(i) *2^-S(-1,i)
                        y=y+dx;            // y(i+1)=y(i)+di(i)x(i) *2^-S(-1,i)
                } else{                    // di(i)=-1
                        z=z+eh[i];
                        x=x-dy;
                        y=y-dx;      }
}       }
void main(void){
        vec_hyp_cordic(2);                 // w=2 for ln(2), atanh(1/3), and sqrt(2)
        printf("\nln(w)=2*atanh((w-1)/(w+1))=2*%d*(2^-23)",z);
        printf("\nsqrt(w)=%d*(2^-23)*1.20753406*1/2",x);     }
```

Fig. 5.3: CORDIC program for square root and natural logarithm.

Table 5.5: CCM Example for Logarithm in Fixed-Point

X	Xb	m	Xm	$T(m)$	Y
51446	149090	3	6431	7719	0
57877	7659	4	3617	3973	-7719
61494	4042	5	1922	2017	-11692
63416	2120	5	1982	2017	-13709
65398	138	9	128	128	-15726
65526	10	13	8	8	-15854
65534	2	15	2	2	-15862
65536	NA	NA	NA	NA	-15864

CCM example for logarithm calculation

Table 5.4 shows the computation of $\ln(0.785)$, which is initialized by setting $x_0 = 0.785$, and $y_0 = 0$. As x is driven toward 1, y is driven the answer of natural logarithm. The transformation rule involves $x_{k+1} = x_k a_k$ and $y_{k+1} = y_k - \ln a_k$, where $\ln(a_k)$ is a pre-calculated look-up table, $a_k = 1 + 2^{-m}$, and m is the leading-1 bit position of the $(1 - x_k)$ value shown in Table 5.4.

CCM program and example for logarithm in fixed-point

Table 5.5 shows the computation of $\ln(0.785)$ in 16-bit fixed-point format, which is initialized by setting $X = 0.785 \times 2^{16} = 51446$, and $Y = 0$. As X is driven toward 2^{16}, Y is driven the answer of natural logarithm with 2^{16} scale. The 2^{16}-scaling $(1 - x_k)$ is $2^{16} - X$, which is defined as Xb to find the leading-1 position m for a_k =1+2^{-m}. The transformation rule of $y_{k+1} = y_k - \ln(a_k)$ with 2^{16} scale is

$$Y_{k+1} = Y_k - T(m),$$

where $T(m) \equiv 2^{16} \times \ln(a_k)$ is a pre-calculated look-up table. For the transformation rule of $X_{k+1} = X_k a_k$, the multiplications by a_k can be replaced by shift-and-add kernels as

$$X_{k+1} = X_k a_k = X_k(1 + 2^{-m}) = X_k + 2^{-m} X_k = X_k + X_m,$$

```
#include <stdio.h>
#include <math.h>
unsigned int T[16]={ 26573, 14624, 7719, 3973, 2017, 1016, 510, 256, 128, 64, 32, 16, 8, 4,
2, 1 };                          // ln(ak) table with 2^16 scale, that is, [ln(1+2^-m)]<<16
unsigned int X0,Y;               // X0: desired for z0=f(x), Y: auxiliary variable for z0=F(x,y)
ccmln(X0){                       // Convergence Computation Method for ln(X0) in fixed-point
    unsigned int X,Xb,Xm,W,k,m;  //
    X=X0;                        // Set initial value of X at desired input X0 for ln(X0)
    Y=0;                         // Set Y=0 for z0=y+ln(X0)=ln(X0)
    Xb=65536-X;                  // Xb is a 2^16 scaled (1-x), that is, 2^16-X
    while(Xb>T[15]){             // looping if convergence not reach yet
        W=1;
        for(k=16;k>0;k--){
            if(Xb>=W) m=k;
            W=W<<1;
        }// for(k                // got the leading-1 bit position m
        Xm=X>>m;                 // x*2^-m=x>>m, stored as Xm
        X=X+Xm;                  // transformation by xk*ak=x*(1+2^-m)=x+(x>>m)
        Y=Y-T[m-1];              // transformation by yk-ln(ak) via lookup-table T[.]
        Xb=65536-X;
    }// while(Xb
}//ccmln
void main(void){
    unsigned int t0,t1,t2,t3;
    float z;
    enable_interrupts();
    enable_clock();

    printf("\nCCM for ln");
    t0=uclock();
    ccmln(51446);     //call CCM natural log of 0.785 with 2^16 scale (integer 51446)
    t1=uclock();
     printf("\nCCM:%d*(2^-16),@%dus",Y,t1-t0);

    t2=uclock();
    z=log(0.785);                     //call TI's natural log of 0.785 in floating-point
    t3=uclock();
    printf("\nTI:%f,@%dus",z,t3-t2);     }//main
```

Fig. 5.4: CCM program for natural logarithm.

Table 5.6: DSP Speed of CORDIC/CCM in Fixed-Point vs. TI Floating-Point Library

Function	TI at 'C67	TI at 'C62	CORDIC at 'C62	CCM at 'C62
sin	$2\ \mu s \sim 3\ \mu s$	$16\ \mu s \sim 25\ \mu s$	$3\ \mu$s / 16 bit	
cos	$2\ \mu s \sim 3\ \mu s$	$17\ \mu s \sim 28\ \mu s$		
ln	$2\ \mu s \sim 3\ \mu s$	$19\ \mu s \sim 21\ \mu s$	$5\ \mu$s / 24 bit	$3\ \mu$s / 16 bit
sqrt	$2\ \mu s \sim 3\ \mu s$	$21\ \mu s \sim 22\ \mu s$		

where $Xm \equiv 2^{-m}X_k$ is belonging to shift operation. As shown in Fig. 5.4, the subroutine $ccmln$ is presented in CCM algorithm for natural logarithm in 16-bit fixed-point format. Moreover, here we also call TI's log library of floating-point format for comparison later.

5.5 Evaluation Results and Discussion

As developed in previous section, we apply CORDIC and CCM algorithms for function evaluation of in integer arithmetic. In this section, we show their evaluation performance on TI DSP chip with comparison to TI library. Then, we briefly discuss with potential optimization or variation in function evaluation.

5.5.1 DSP Performance of Function Evaluation

Table 5.6 shows DSP Speed of CORDIC and CCM in fixed-point arithmetic in comparison with TI support library. As discussed previously, we apply CORDIC algorithm for function evaluation of sine, cosine, natural logarithm and square rooting, and use CCM algorithm as an aid in computation of natural logarithm. However, these four elementary functions provided by TI support library are belonging to floating-point format. It is time-consuming to run floating-point format on fixed-point device. Our developed fixed-point routine avoids the extra time. For example, TI floating-point sin and cos take more than two 16 μs respectively on a TI's fixed-point DSP TMS320C6201, but our CORDIC routine takes one 3 μs to finish the simultaneous calculation of both cosine and sine of 16-bit precision. Similar case also happens for evaluation of natural logarithm and square rooting. Therefore, it is possible to port our channel simulator on II's fixed-point DSP board Quatro62. Furthermore, these elementary functions are helpful for the generation of channel model.

5.5.2 Discussion on Variations and Optimization

Our fixed-point CORDIC/CCM library is efficient enough to compare with TI floating-point library. It provides one of efficient generation as porting to the fixed-point device. However, it is not efficient enough with comparisons to DSP instructions of single cycle style. Therefore, we need optimization or variation methods for speedup function evaluation to serve as efficient generation of channel model.

There are several methods for evaluating elementary and other functions. For example, in the book of computer arithmetic [27], two chapters are devoted to square-rooting methods and table lookup, respectively; and two chapters to CORDIC, other convergence methods, and approximation [21]. However, many methods require conditional break in the loop. Unfortunately, the branching operation is not friendly for DSP resources. Besides seeking efficient algorithms, we are considering to optimize coding styles, such as loop unrolling, software pipelining, intrinsic operators or assembly language and so on [43], [39], [37]. For example, we should not have any if-else condition statements in the loop, otherwise, the loop is not software pipelined. Similar case happens in a time-consuming routine of scramble code generation in the existing simulator.

Chapter 6

Conclusion

The DSP is a programmable tool to achieve different functionalities. We wanted to improve the previously developed wireless channel simulation system consisting of three interconnected DSPs. We found ways of making the existing system work better.

Firstly, multiprocessor programs without specifying the start-up order could not run smoothly. We pointed out unlucky style and tools compatibility, and then did migration to a specified new platform. Therefore, the start-up problems went away.

Secondly, real-time performance degraded on the three connected DSPs. In order to reduce latency, we proposed the pipelining-512 structure employing non-blocking mode DMA and double buffering. We observed actual execution speed and pointed out time-consuming factors, including waiting for conflicting resources or FIFO handshaking. Then, for applying double buffering correctly, we paid attention to an improper memory management. Therefore, the three DSPs could efficiently achieve that one block used for computing while another block used for moving by a timing-sharing DMA.

Thirdly, fixed-point generation of channel coefficients was not available yet. However, vendor supported sine, cosine, logarithm, and square root functions belong to floating-point. Applying CORDIC and CCM algorithms, we evaluated these four functions in integer arithmetic, to serve as the generation of channel coefficients and Gaussian noise.

In conclusion, for efficient simulating wireless channel on multiprocessor platform, we sought out several methods to improve hardware and software efficiently.

Appendix A

Things to Note in DSP Implementation

This appendix is concerned with the Quator6x (Q6x) platform housing TMS320C6x ('C6x) devices. It is important to take care of memory management again in DSP Implementation. Besides tools compatibility, unlucky coding styles usually involve the linker command file (.cmd), memory map (.map), or header file (.h). All with all embedded systems, the command file is indispensable for real time application [12].

A.1 Emails Concerning Two Problems in Using MPO and CCS

"We face two issues that it's happening in environment of II Q6x or TI DSP tools. There are four kinds of environment as shown in row 1 to 2 of Table A.1. Old environment by II 2.75 or 2.70 CD with CCS1.2, and new environment by CCS2.0 and II 2.97 or 2.87 CD in a clean PC of Win2000SP2. Any user application run by two methods: one is run *.out in CCS conjunction with UniTerminal, another is *.out or *.mpo by UniTerminal without CCS.

We already use Q6x example and simple summation to check setting of the four environments as shown in row 15 to 16 of Table A.1. My major application train includes channel, modulation, and filter receiver as shown in row 9 to 11 of the Table. However, my application runs OK in CCS1.2 environment, but it face two issues in other environment.

For the issue 1, its .OUT or .MPO file could NOT run by UniTerminal without CCS1.2.

After some analysis, it seems sensitive to 2 factors: one is .cinit to DRAM or SDRAM, another is array size less or more than 500. As shown in row 6 to 7 of Table A.1, problem repeated clearly during .cinit session. However, it's OK for SDRAM as shown in row 3-both side and row 6-left side of the Table. Also see remark1 in Fig. 3.7.

For the issue 2, it repeated internal error during compile phase in CCS2.0 environment. As shown in log file of my application, that is TP>> internal error: bad type: TYPE::type_qualified(). As we known, many internal error of other kinds listed at bug list in Website of TI. However, internal error exist but 0 error during compiling, and then 1 error during linking due to no *.asm file generated. Also see remark2 in Fig. 3.8.

To demo issue as similar as possible, some zip files attached in this email. As shown in row 2 to 7 of Table A.1, the 5 simple codes provided to demo issue 1 of MPO or .cinit. As shown in row 12 to 14 of the Table, we also provide other 3 simple codes to demo issue 2 of bad type internal error. Finally, we need my original application run smoothly in the four environments. Hope this information will speed up your solution feedback as soon as possible."

Reply from TI teacher George Hsieh

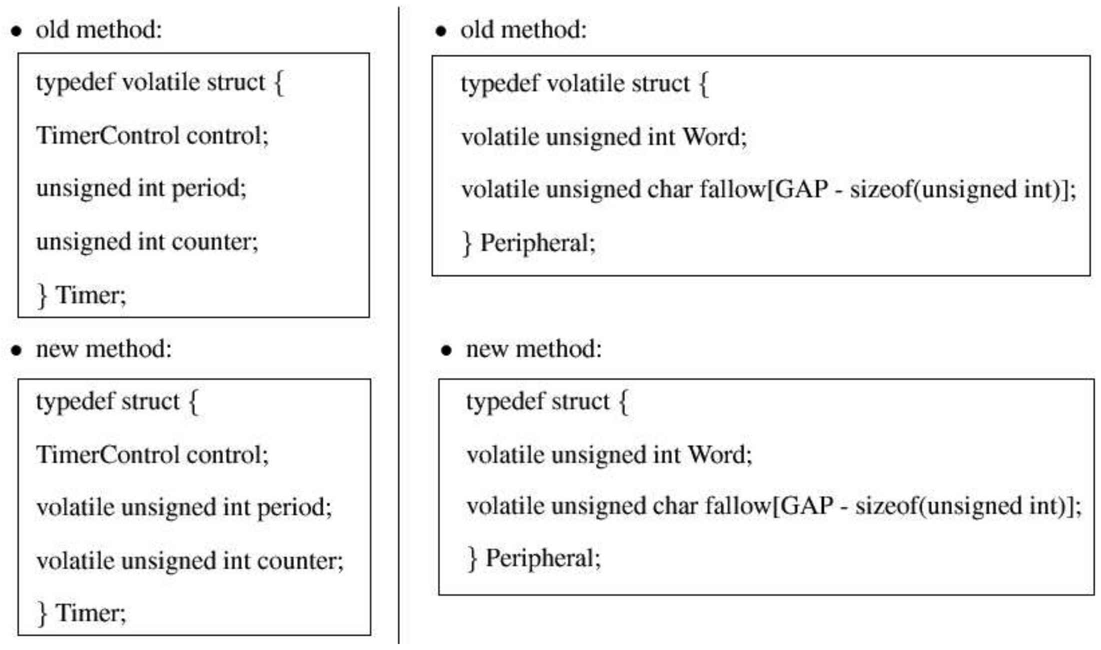
"I've forward the question to product information center, but they've responded with a message saying that this problem has never occurred in their FAQ record. I would say, if possible, I'll probably grab the libraries from you, just to test build the project on my machine, if the lib is not an issue for you. OK?"

Reply from techsprt@innovative-dsp.com

"You will have problems running in all the environments. Code Composer Studio has changed a lot over time. Version 2 of the studio is completely different and version 2.2 is different again. the Latest version on our web site has everything for version 2.2 of C.C.S. this will compile with no errors. You are facing two different approaches to compiler features. On compiler will take a "volatile struct" and the other will not. We had to

Table A.1: Summary of Two Issues in MPO and CCS

1	I.I. install CD	New: II 2.97 or 2.87		Old: II 2.75 or 2.70	
2	Environment	CCS 2.0	Uniterminal 1.0.14.2	CCS 1.2	Uniterminal 1.0.1.1
3	ORI500 (.cinit > DRAM) (Demo1 of Issue1)	OK	OK	OK	OK
4	CHA500 (.cinit > SDRAM) (Demo2 of Issue1)	OK	OK	OK	OK
5	ORI 501 (.cinit > DRAM) (Demo3 of Issue1)	OK	OK	OK	OK
6	CHA501 (.cinit > SDRAM) (Demo4 of Issue1)	OK	OK	OK	NG (Run time)
7	CHA501cr (.cinit > SDRAM) (Demo5 of Issue1)	OK	OK	OK	NG (Load time)
8	Channel (.cinit > DRAM)	Internal error	Not yet	Cannot allocate in DRAM (page0)	Not yet
9	Channel (.cinit > SDRAM) (User application)	Internal error	Not yet	OK	MPO NG
10	Modulation (User application)	Internal error	Not yet	OK	MPO NG
11	Filter_receiver (User application)	Internal error	Not yet	OK	MPO NG
12	Bad_A (Demo1 of Issue2)	Internal error	Not yet	OK	OK
13	Bad_B (Demo2 of Issue2)	Internal error	Not yet	OK	OK
14	Bad_C (Demo3 of Issue1)	Internal error	Not yet	OK	OK
15	TEST or TEST2 (Q6x Example)	OK	OK	OK	OK
16	Summation (1 to 100)	OK	OK	OK	OK



- old method:

```
typedef volatile struct {
TimerControl control;
unsigned int period;
unsigned int counter;
} Timer;
```

- new method:

```
typedef struct {
TimerControl control;
volatile unsigned int period;
volatile unsigned int counter;
} Timer;
```

- old method:

```
typedef volatile struct {
volatile unsigned int Word;
volatile unsigned char fallow[GAP - sizeof(unsigned int)];
} Peripheral;
```

- new method:

```
typedef struct {
volatile unsigned int Word;
volatile unsigned char fallow[GAP - sizeof(unsigned int)];
} Peripheral;
```

Fig. A.1: The volatile structure needs to be changed.

go in and change to each individual element to be volatile. you need to settle on one environment and use that. Even the Make files are different with different switches. Libraries are different from the versions. I have done what you have asked but it is a tedious effort to maintain. II does not support this approach although I can be of some help in parts of the upgrades. To convert for the new Code Composer Studio from older source code the volatile structure needs to be changed (as shown in Fig. A.1).

Two Issues Related to Quatro6x Still Remain

“After volatile struct changed to new method (Timer in ii_c6x.h,and Peripheral in Periph.h). Issue1 still exist in CCS1.2 of II2.70 or 2.75. Issue2 still exist in CCS2.0 of II2.97 or 2.85. Do you actually run the isssu1 demo program of CHA501 in your CCS1.2 of II2.70 or 2.75? Also run the isssu2 demo program of BAD_A in your CCS2.0 of II2.97 or 2.85? And actually run OK for the two issue demo program of in your newest CCS2.2 of newest II CD version? Do the above two violate struct change really hit the issues or not yet? Could you continue to teach me how to do?”

Reply from techsprt@innovative-dsp.com

"I need more specifics. No I have not looked at the project to update it. I have however done this before on other projects. You should just download the new toolset and discard the old ones. Update your code to the latest and then you will not have to worry about this. The libraries might have to be recompiled to insure all of the compiler issues are met. I did do that also with the new compiler. This may not be necessary. I do not remember now as this was a while ago. Run a find for all the possible volatile struct in your include files to see if there are others.

No the issue of the older toolset will always be there if you try to compile it on the Newer Code Composers. The new toolset has already been updated for the newer releases of Code Composer Studio. This is now aimed at version 2.2 of C.C.S. the latest version is on the WEB and can be downloaded by you after you register your computer."

A.2 Emails Concerning Four DMA Channels and One DMA Bus

"I'm a NCTU student using Quatro6x DSP board. Now, there are problems about DMA channels.

- Background:

  There are four DMA channels at TMS320C6x01 DSP. There are four TMS320C6x01 on Quatro6x Board. TI's on-chip memory: two blocks, four banks each. II's dma_copy_mem, dma_port_to_mem, and dma_mem_to_port used for data movement.

- Goal:

  We want to pipeline the five actions: one computing under background of four DMA channels moving data simultaneously.

- Status:

  Up to now, it is OK to pipeline one CPU and one DMA using double buffering at separate block. However, it is NG to pipeline one CPU and four DMA background movements simultaneously.

- Problem:

  1) Could the four DMA channels move data simultaneously, or not?

  2) If yes, how to do? Need to set data at separate bank?

  3) Set by Data alignment or Structure for interleave LSB address?

  4) If no, why could not? Limit from only one DMA controller having only one DMA bus?

  5) Any suggestion for my goal in Q6x board?"

Reply from techsprt@innovative-dsp.com

"I do not fully understand your problem. There are four DMA channels on each processor of the Q6x. Each processor is responsible for there own DMAs. DMA uses the same data bus as the processor so this is a time shared operation. If you have multiple DMA running they also time share the bus. The DMA when complete calls a DMA interrupt so you can use this to set flags and/or reset the DMA for new addresses. You can use the DMA with any of TI or II setup routines. If you have special needs then use the TI functions and if they have what you need then you will not have to build your own. At any time you can set the DMA for any special needs of data flow. We just give you the most common in the libraries. DSP0 does use one of the DMA channels for the BusMaster and this should not be changed unless you are not Busmastering data to or from the host."

Reply from TI teacher George Hsieh

"DMA in 'C6x01 has only one controller and one 32-bit bus. Access priorities are fixed for each channel. Although you can pre-configure 4 different memory access configuration through 4 channels, only the highest priority channel can use the bus if started simultaneously, once this channel completes, the next highest priority channel would begin, and so on. If double buffer works for you, with a little modification, you can make it as many buffer you want. Try to increase the frame count, and allocate more buffers as required, use frame and element index mechanism to direct the input to 4 different buffers, this can be done with only ONE channel of DMA. For details, please refer to the C6000 peripheral reference guide (SPRU190). It's difficult to use DMA to move data in to/out from another DSP without a share memory. But simply for this case, I guess 3 channel will do, theoretically."

Reply from support@ti.com

"Thank you for contacting Texas Instruments Technical Support. Your email has been received and a case number 35347272 has been assigned to your inquiry. Regarding your query, it is not possible to have 4 DMA channels move data simultaneously. There is only one 32-bit DMA bus to the data access controller of C6201 which is the limitation. Please refer to figure 1-1 of SPRU577. Note that the 4 DMA channels have fixed priorities with channel 0 having highest priority and channel 3 having lowest priority. For more details on this please refer to section 2.7 of DMA Reference Guide - SPRU577. For your application, you can use Split-channel operation as described in section 2.6 of SPRU577. Also CPU and DMA has separate buses for memory and hence they can have simultaneous access. Also refer to the following: Chapter 3 of SPRU577 TMS320C620x/TMS3206701 DMA and CPU: Data Access Performance - SPRA614A Hope this helps. Please get back to us if you have any further questions."

Bibliography

[1] 3GPP, *Technical Specification Group Radio Access Network; UE Radio Transmission and Reception (FDD)*, doc. no. 3G TS 25.101 version 4.1.0, June 2001.

[2] 3GPP, *Technical Specification Group Radio Access Network: Physical Channels and Mapping of Transport Channels onto Physical Channels (FDD)*, doc. no. 3G TS 25.211 version 4.1.0, June 2001.

[3] 3GPP, *Technical Specification Group Radio Access Network; Spreading and Modulation (FDD)*, doc. no. 3G TS 25.213 version 4.1.0, June 2001.

[4] H. M. Ahmed, "The generalized convergence computation method," in *Proc. IEEE Int. Conf. Acoust Speech Signal Processing, 1989*, Glasgow, UK, pp. 849–852.

[5] B. Ahn, "A study on high-speed Gaussian random number generator," in *Proc. IEEE Asia Pacific Conf. Circuits Syst.*, Nov. 1996, Seoul, Korea, pp. 30–32.

[6] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proc. ACM/SIGDA Sixth Int. Symp. FPGAs*, Feb. 1998, Monterey, CA, pp. 191–200, also available at http://www.andraka.com/files/crdcsrvy.pdf.

[7] Yu-Jung Chang, "Study and DSP implementation of successive interference cancellation (SIC) receiver for 3GPP wideband-CDMA uplink transmission," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2002.

[8] R. Chassaing, *DSP Applications Using C and the TMS320C6x DSK.* New York: Wiley, 2002.

[9] S. G. Chen, C. C. Li, and Y. D. Hou, "Compatible CORDIC and CCM algorithm for small area realization," *VLSI Signal Processing VIII, IEEE Press*, pp. 572–579, 1995.

[10] T. C. Chen, "Automatic computation of exponentials, logarithms, ratios and square roots," *IBM J. Res. and Dev.*, pp. 380–388, July 1972.

[11] Wei-Yu Chen, "Study and DSP implementation of 3GPP WCDMA uplink multiplexing and channel coding methods," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2002.

[12] N. Dahnoun, *Digital Signal Processing Implementation Using the TMS320C6000^{TM} DSP Platform.* Prentice Hall, 2000.

[13] P. Dent, G. E. Bottomley, and T. Croft, "Jakes' fading model revisited," *Electron. Lett.*, vol. 29, no. 13, June 1993.

[14] Jian-Hau Her, "DSP implementation of CDMA based on parallel interference cancellation receiver," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2001.

[15] Yuan-Der Hou, "Unified design of an efficient CORDIC/CCM Processor," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 1995.

[16] X. Hu, R. G. Harber, and S. C. Bass, "Expanding the range of convergence of CORDIC algorithm," *IEEE Trans. Computers*, vol. 40, no. 1, pp. 13–21, Jan. 1991.

[17] Y. H. Hu, "CORDIC-based VLSI architecture for digital signal processing," *IEEE Signal Processing Mag.*, pp. 16–35, July 1992.

[18] The official web site of Innovative Integration, http://www.innovative-dsp.com/.

[19] Innovative Intergration, *Quatro6x Development Package Manual.* Jan. 2001.

[20] M. C. Jeruchim, P. Balaban, and K.S. Shanmugan, *Simulation of Communication Systems: Modeling, Methodology, and Techniques, 2nd ed.* Kluwer Academic, 2000.

[21] Dong-U Lee, "Reconfigurable hardware for function evaluation and LDPC coding," MPhil/PhD Transfer Report, Department of Computing, Imperial College, London, United Kingdom, July 2003.

[22] Y. Li and X. Huang, "The simulation of independent Rayleigh faders," *IEEE Trans. Commun.*, vol. 50, no. 9, pp. 1503–1514, Sep. 2002.

[23] Jia-Ching Lin, "DSP implementation of uplink code synchronization for WCDMA wireless system," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2002.

[24] Jing-Shyong Lin, "DSP implementation and error performance study on speech source/channel coding," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2002.

[25] D. E. Metafas, G. A. Krikis, and C. E. Goutis, "VLSI design of 8-bit fixed point CORDIC processor with extended operation set," in *Proc. EURO ASIC, 1991*, Paris, pp. 158–161.

[26] A. V. Oppenhiem and R. W. Schafer, *Discrete-Time Signal Processing.* Upper Saddle River, New Jersey: Prentice Hall, 1989.

[27] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs.* New York: Oxford University Press, 2000.

[28] S. Y. Park and N. I. Cho, “Fixed point error analysis of CORDIC processor based on the variance propagation,” in *Proc. IEEE Int. Conf. Acoust. Speech Siginal Processing*, Apr. 2003, pp. 565–568.

[29] R. Parris, “Elementary functions and calculators,” Phillips Exeter Academy, http://www.swarthmore.edu/NatSci/smaurer1/Math6B/calculatorsParris-rev.pdf.

[30] M. F. Pop and N. C. Beaulieu, “Design of wide-sense stationary sum-of-sinusoids fading channel simulators,” in *Conf. Rec., IEEE Int. Conf. Commun.*, April 2002, pp. 709–716.

[31] M. F. Schollmeyer and W. H. Tranter, “Noise generator for the simulation of digital communication systems,” in *Proc. 24th Ann. Simulation Symp.*, pp. 264–275, 1991.

[32] Song-Ling Tsai, “Study and DSP implementation of 3GPP wideband-CDMA transmission signal processing and wireless channel simulation,” M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., June 2002.

[33] The official web site of Texas Instruments, http://www.ti.com/.

[34] Texas Instruments, *Using TI FIFOs to Interface High-Speed Data Converters With TI TMS320 ™ DSPs.* Literature number SDMA003, June 2001.

[35] Texas Instruments, *Parallel 2-D FFT Implementation With TMS320C4x DSPs.* Literature number SPRA027A, Feb. 1994.

[36] Texas Instruments, *TMS320C620x/TMS320C6701 DMA and CPU: Data Access Performance.* Literature number SPRA614A, Aug. 2000.

[37] Texas Instruments, *C Implementation of the TMS320C62xx Intrinsic Operators.* Literature number SPRA616, Dec. 1999.

[38] Texas Instruments, *TMS320C6000^{TM} DSP Multiprocessing with a Crossbar Switch.* Literature number SPRA725, Feb. 2001.

[39] Texas Instruments, *TMS320C6000 Optimizing Compiler User's Guide.* Literature number SPRU187I, Apr. 2001.

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

[41] Texas Instruments, *TMS320C6000 Peripherals Reference Guide.* Literature number SPRU190D, Mar. 2001.

[42] Texas Instruments, *TMS320C6000 Technical Brief.* Literature number SPRU197D, Feb. 1999.

[43] Texas Instruments, *TMS320C6000 Programmer's Guide.* Literature number SPRU198F, Feb. 2001.

[44] Texas Instruments, *TMS320C620x/C670x DSP Program and Data Memory Controller/ Direct Memory Access (DMA) Controller Reference Guide.* Literature number SPRU577, July 2003.

[45] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Computers*, vol. 8, no. 3, pp. 330–334, Sep. 1959.

[46] J. S. Walther, "A unified algorithms for elementary functions," in *AFIPS Spring Joint Computer Conference*, pp. 379–385, 1971.

[47] J. Wang *et al.*, "Comparison of statistical properties between an improved Jakes' model and the classical one," in *Proc. IEEE PIMRC*, pp. 380–384, Sep. 2003.

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

作者簡歷

李建興，民國五十九年生於台南市。民國八十四年畢業於國立台灣科技大學電子系，民國八十八年進入國立交通大學電機資訊學院碩士在職專班電子與光電組，民國九十三年取得碩士學位，論文題目為:在多數位訊號處理器系統上進行高效率無線通道模擬之研討。