

國立交通大學

電子工程學系 電子研究所碩士班

碩 士 論 文

MPEG-4 先進音訊編碼

在 DSP/FPGA 平台上的實現與最佳化



**MPEG-4 AAC Implementation
and Optimization on DSP/FPGA**

研 究 生：曾建統

指 導 教 授：杭學鳴 博士

中 華 民 國 九 十 三 年 六 月

MPEG-4 先進音訊編碼
在 DSP/FPGA 平台上的實現與最佳化

MPEG-4 AAC Implementation
and Optimization on DSP/FPGA

研 究 生：曾建統

Student : Chien-Tung Tseng

指 導 教 授：杭學鳴 博士

Advisor : Dr. Hsueh-Ming Hang



A Thesis

Submitted to Institute of Electronics

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of Requirements

for the Degree of

Master of Science

in

Electronics Engineering

June 2004

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 三 年 六 月

MPEG-4 先進音訊編碼在 DSP/FPGA 平台上的實現與最佳化

學生：曾建統

指導教授：杭學鳴 博士

國立交通大學 電子工程學系電子研究所碩士班

摘要

MPEG-4 先進音訊編碼(AAC)是由 ISO/IEC MPEG 所制訂的一套非常有效率的音訊壓縮編碼標準。

在本篇論文當中，我們首先統計 MPEG-4 先進音訊編碼在 DSP 上的執行情況，發現霍夫曼解碼(Huffman decoding)和反修正離散餘弦轉換(IMDCT)所需要的時脈週期總數為最多，因為針對反修正離散餘弦轉換在 DSP 上的實現作最佳化，同時我們也希望利用 FPGA 來克服用 DSP 執行的瓶頸部分，所以將霍夫曼解碼以及反修正離散餘弦轉換的一部份反快速傅立葉轉換(IFFT)放到 FPGA 實現。

在 DSP 實現方面，我們針對 DSP 的架構使用運算量更少的演算法，使用適合 DSP 處理的資料型態，並使用 TI DSP 特殊指令來改寫程式，大幅提高其執行效率，這個部分大約增加了 503 倍的速度。在 FPGA 實現方面，我們設計針對霍夫曼解碼以及反快速傅立葉轉換的架構，並針對硬體架構設計來作調整，使其運算效能提高，同時兼顧減少使用面積的考量。霍夫曼解碼大約比 DSP 的版本增加了 56 倍的速度，反快速傅立葉轉換大約較 DSP 最快的版本增加了 4 倍的速度。最後並考慮 DSP 和 FPGA 設計之間的溝通問題。

MPEG-4 AAC Implementation and Optimization on DSP/FPGA

Student: Chien-Tung Tseng

Advisor: Dr. Hsueh-Ming Hang

Department of Electronics Engineering
Institute of Electronics
National Chiao Tung University

Abstract

MPEG-4 AAC (Advanced Audio Coding) is an efficient audio coding standard. It is defined by the MPEG (Moving Pictures Experts Groups) committee, which is one of ISO (International Standard Organization) working groups. In this thesis, we first analyze the computational complexity of MPEG-4 AAC decoder program. We found that the Huffman decoding and the IMDCT (inverse modified discrete cosine transform) require the most clock cycles to execute on DSP. Hence, we optimize the IMDCT codes on DSP. In addition, we use FPGA to remove the bottleneck in DSP execution. Thus, we implement the Huffman decoding and the inverse fast Fourier transform), which is a part of IMDCT, on FPGA.

In order to speed up the AAC decoder on DSP, we need to choose appropriate algorithms for DSP implementation. Thus, appropriate data types are chosen to present the data. Furthermore, we use the TI (Texas Instruments) DSP intrinsic functions to increase the DSP execution efficiency. The modified version of IMDCT is about 503 times faster than the original version. For the FPGA implementation, we adopt and modify the existing architectures for Huffman decoding and 512-point IFFT. In addition, we use VLSI design techniques to improve the performance and reduce the chip area in FPGA implementation. The FPGA implementation of Huffman decoding and 512-point IFFT is about 56 and 4 times faster than the corresponding DSP implementations, respectively. Also, in this project, we design and implement the communication interface between DSP and FPGA.

誌謝

本論文承蒙恩師杭學鳴教授細心的指導與教誨，方得以順利完成。在研究所生涯的兩年中，杭教授不僅在學術研究上給予學生指導，在研究態度亦給許相當多的建議，在此對杭教授獻上最大的感謝之意。

此外，感謝所有通訊電子暨訊號處理實驗室的成員，包括多位師長、同學、學長姊和學弟妹們，特別是楊政翰、陳繼大、吳俊榮、蔡家揚學長給予我在研究過程中的指導與建議。同時也要感謝實驗室同窗仰哲、明瑋、子瀚、筱晴、盈縈、明哲、宗書在遇到困難的時候能夠互相討論和砥礪，並希望接下我們工作的學弟盈閩、志楹、昱昇學弟能傳承實驗室認真融洽的氣氛，在學術上有所貢獻。感謝我的女朋友佳韻，在生活中給予我的支持與鼓勵，使我在艱難的研究過程中，能夠保持身心的健康與平衡。

謝謝養育我多年的父親及母親，還有我的弟弟，沒有你們的栽培與鼓勵，我無法有今天的成就。

要感謝的人很多，無法一一列述，謹以這篇論文，獻給全部讓我在研究所生涯中難忘的人，謝謝。

曾建統

民國九十三年六月 於新竹

Contents

Chapter 1 Introduction	1
Chapter 2 MPEG-2/4 Advanced Audio Coding.....	3
2.1 MPEG-2 AAC.....	3
2.1.1 Gain Control.....	4
2.1.2 Filterbank	5
2.1.3 Temporal Noise Shaping (TNS).....	7
2.1.4 Intensity Coupling.....	8
2.1.5 Prediction	8
2.1.6 Middle/Side (M/S) Tool	9
2.1.7 Scalefactors	10
2.1.8 Quantization.....	10
2.1.9 Noiseless Coding	10
2.2 MPEG-4 AAC Version 1.....	11
2.2.1 Long Term Prediction (LTP).....	12
2.2.2 Perceptual Noise Substitution (PNS).....	13
2.2.3 TwinVQ.....	14
2.3 MPEG-4 AAC Version 2.....	15
2.3.1 Error Robustness.....	15
2.3.2 Bit Slice Arithmetic Coding (BSAC).....	16
2.3.3 Low-Delay Audio Coding.....	17
2.4 MPEG-4 AAC Version 3.....	17
Chapter 3 Introduction to DSP/FPGA	19
3.1 DSP Baseboard	19
3.2 DSP Chip.....	20
3.2.1 Central Processing Unit (CPU).....	21
3.2.2 Data Path.....	23
3.2.3 Pipeline Operation	25
3.2.4 Internal Memory	26
3.2.5 External Memory and Peripheral Options	26
3.3 FPGA Chip.....	27

3.4 Data Transmission Mechanism	28
3.4.1 Message Interface	29
3.4.2 Streaming Interface	29
Chapter 4 MPEG-4 AAC Decoder Implementation and Optimization on DSP	31
4.1 Profile on DSP	31
4.2 Optimizing C/C++ Code	32
4.2.1 Fixed-point Coding	32
4.2.2 Using Intrinsic Functions	33
4.2.3 Packet Data Processing	33
4.2.4 Loop Unrolling and Software Pipelining	34
4.2.5 Linear Assembly and Assembly	34
4.3 Huffman Decoding	35
4.4 IMDCT	36
4.4.1 N/4-point FFT Algorithm for MDCT	37
4.4.2 Radix-2 ³ FFT	39
4.4.3 Implementation of IMDCT with Radix-2 IFFT	41
4.4.4 Implementation of IMDCT with Radix-2 ³ IFFT	41
4.4.5 Modifying of the Data Calculation Order	42
4.4.6 Using Intrinsic Functions	43
4.4.7 IMDCT Implementation Results	44
4.5 Implementation on DSP	45
Chapter 5 MPEG-4 AAC Implementation and Optimization on DSP/FPGA	47
5.1 Huffman Decoding	47
5.1.1 Integration Consideration	47
5.1.2 Fixed-output-rate Architecture	49
5.1.3 Fixed-output-rate Architecture Implementatiopn Result	51
5.1.4 Variable-output-rate Architecture	52
5.1.5 Variable-output-rate Architecture Implementation Result	54
5.2 IFFT	55
5.2.1 IFFT Architecture	55
5.2.2 Quantization Noise Analysis	57
5.2.3 Radix-2 ³ SDF SDF IFFT Architecture	59
5.2.4 IFFT Implementation Result	62
5.3 Implementation on DSP/FPGA	65
Chapter 6 Conclusions and Future Work	67
Bibliography	69
Appendix A N/4-point FFT Algorithm for MDCT	71
Appendix B Radix-2 ² and Radix-2 ³ FFT	75

List of Tables

Table 4.1 Profile of AAC decoding on C64x DSP.....	32
Table 4.2 Processing time on the C64x DSP with different datatypes.....	33
Table 4.3 Comparison of computational load of FFT.....	40
Table 4.4 DSP implementation result of different datatypes	41
Table 4.5 SNR of IMDCT of different datatypes.....	41
Table 4.6 DSP implementation result of different datatypes	42
Table 4.7 SNR of IMDCT of different datatypes.....	42
Table 4.8 DSP implementation results of the modified data calculation order.....	42
Table 4.9 DSP implementation results of using intrinsic functions.....	44
Table 4.10 DSP implementation results of IMDCT.....	45
Table 4.11 Comparison of modification IMDCT and IMDCT with TI IFFT library	45
Table 4.12 Comparison of original and the optimized performance	46
Table 4.13 The ODG of test sequence “guitar”	46
Table 4.14 The ODG of test sequence “eddie_rabbitt”.....	46
Table 5.1 The performance Comparison of DSP and FPGA implementation	52
Table 5.2 Comparison of hardware requirements	56
Table 5.3 The performance comparison of DSP and FPGA implementation	64
Table 5.4 Implementation on DSP/FPGA.....	65

List of Figures

Fig. 2.1 Block diagram for MPEG-2 AAC encoder.....	4
Fig. 2.2 Block diagram of gain control tool for encoder	5
Fig. 2.3 Window shape adaptation process.....	6
Fig. 2.4 Block switching during transient signal conditions.....	7
Fig. 2.5 Pre-echo distortion	7
Fig. 2.6 Prediction tool for one scalefactor band.....	9
Fig. 2.7 Block diagram of MPEG-4 GA encoder.....	12
Fig. 2.8 LTP in the MPEG-4 General Audio encoder	13
Fig. 2.9 TwinVQ quantization scheme	15
Fig. 3.1 Block Diagram of Quixote	20
Fig. 3.2 Block diagram of TMS320C6x DSP	21
Fig. 3.3 TMS320C64x CPU Data Path.....	23
Fig. 3.4 Functional Units and Operations Performed	24
Fig. 3.5 Functional Units and Operations Performed (Cont.).....	25
Fig. 3.6 General Slice Diagram.....	28
Fig. 4.1 Intrinsic functions of the TI C6000 series DSP (Part.).....	33
Fig. 4.2 Sequential model of Huffman decoder	35
Fig. 4.3 Parallel model of Huffman decoder.....	36
Fig. 4.4 Fast MDCT algorithm	38
Fig. 4.5 Fast IMDCT algorithm	39
Fig. 4.6 Butterflies for 8-point radix-2 FFT.....	40
Fig. 4.7 Butterflies for a radix-2 ³ FFT PE	40
Fig. 4.8 Simplified data flow graph for 8-point radix-2 ³ FFT	40
Fig. 4.9 Comparison of the data calculation order.....	42
Fig. 4.10 Intrinsic functions we used	44
Fig. 4.11 TI IFFT library.....	45
Fig. 5.1 Flow diagram of MPEG-4 AAC Huffman decoding.....	48
Fig. 5.2 Block diagram of DSP/FPGA integrated Huffman decoding.....	49
Fig. 5.3 Block diagram of fixed-output-rate architecture	50
Fig. 5.4 Output Buffer of code index table	50

Fig. 5.5 Waveform of the fixed-output-rate architecture	51
Fig. 5.6 Synthesis report of the fixed-output-rate architecture	51
Fig. 5.7 P&R report of the fixed-output-rate architecture	52
Fig. 5.8 Block diagram of the variable-output-rate architecture.....	53
Fig. 5.9 Comparison of the waveform of the two architectures.....	53
Fig. 5.10 Synthesis report for the variable-output-rate architecture	54
Fig. 5.11 P&R report for the variable-output-rate architecture.....	55
Fig. 5.12 Block diagram of shifter-adder multiplier	57
Fig. 5.13 Quantization noise analysis of twiddle multiplier is 256	58
Fig. 5.14 Quantization noise analysis of twiddle multiplier is 4096	58
Fig. 5.15 Block diagram of radix 2^3 SDF 512-point IFFT pipelined architecture.....	59
Fig. 5.16 Simplified data flow graph for each PE.....	59
Fig. 5.17 Block diagram of the PE1.....	60
Fig. 5.18 Block diagram of the PE2.....	61
Fig. 5.19 Block diagram of the PE3.....	61
Fig. 5.20 Block diagram of the twiddle factor multiplier	62
Fig. 5.21 Waveform of the radix- 2^3 512-point IFFT.....	62
Fig. 5.22 Synthesis report of radix- 2^3 512-point IFFT.....	63
Fig. 5.23 P&R report of radix- 2^3 512-point IFFT	64

Chapter 1

Introduction

MPEG stands for ISO “Moving Pictures Experts Groups.” It is a group working under the directives of the International Standard Organization (ISO) and the International Electro-technical Commission (IEC). This group work concentrates on defining the standards for coding moving pictures, audio and related data.

The MPEG-4 AAC (Advanced Audio Coding) standard is a very efficient audio coding standard at the moment. Similar to many other audio coding schemes, MPEG-4 AAC compresses audio data by removing the redundancy among samples. In addition, it includes several tools to enhance the coding performance, temporal noise shaping (TNS), perceptual noise substitution (PNS), spectral band replication (SBR) and others. Hence, the MPEG-4 AAC standard can compress audio data at high quality with high compression efficiency.

We implement the MPEG-4 AAC encoder and decoder on a DSP processor. Some of the MPEG-4 AAC tools’ efficiencies are limited by the data processing mechanism of the DSP processors. In this project, we try to use VLSI (very large scale integration) design concept to improve the implementation. The idea is based on the SoC (System on a Chip) methodology.

We thus adopt the DSP/FPGA (Digital Signal Processor/Field Programmable Gate Array) platform to implement MPEG-4 AAC encoder and decoder. The DSP baseboard is made by Innovative Integration's Quixote. It houses a Texas Instruments' TMS320C6416 DSP and a Xilinx Virtex-II FPGA. We also need the communication interface provided by the DSP baseboard manufacture. This thesis will describe the implementation and optimization of an AAC decoder on the DSP and on the FPGA.

The organization of the thesis is as follows. In chapter 2, we describe the operations of MPEG-2 AAC and MPEG-4 AAC. Then, in chapter 3, we describe the DSP/FPGA environment. In chapter 4, we speed up the decoder process on DSP. In chapter 5, we include

FPGA for implementing Huffman decoding and IFFT to improve the overload performance.
At the end, we give a conclusion and future work of our system.



Chapter 2

MPEG-2/4

Advanced Audio Coding

In this chapter, we will briefly describe the MPEG-2/4 AAC (Advanced Audio Coding) operating mechanism. Details can be found in [1] and [2] respectively.

2.1 MPEG-2 AAC



In 1994, a MPEG-2 audio standardization committee defined a high quality multi-channel standard without MPEG-1 backward compatibility. It was the beginning of the development of “MPEG-2 AAC.” The aim of MPEG-2 AAC was to reach “indistinguishable” audio quality at data rate of 384 kbps or lower for five full-bandwidth channel audio signals as specified by the ITU-R (International Telecommunication Union, Radio-communication Bureau). Testing result showed that MPEG-2 AAC needed 320 kbps to achieve the ITU-R quality requirements. This result showed that MPEG-2 AAC satisfied the ITU-R standard, and then MPEG-2 AAC was finalized in 1997.

Like most digital audio coding schemes, MPEG-2 AAC algorithm compresses audio signals by removing the redundancy between samples and the irrelevant audio signals. We can use time-frequency analysis for removing the redundancy between samples, and make use of the signal masking properties of human hearing system to remove irrelevant audio signals. In order to allow tradeoff between compression the audio quality, the memory requirement and

the processing power requirement, the MPEG-2 AAC system offers three profiles: main profile, low-complexity (LC) profile, and scalable sampling rate (SSR) profile. Fig 2.1 gives an overview of a MPEG-2 AAC encoder block diagram. We will describe each tool briefly in this section.

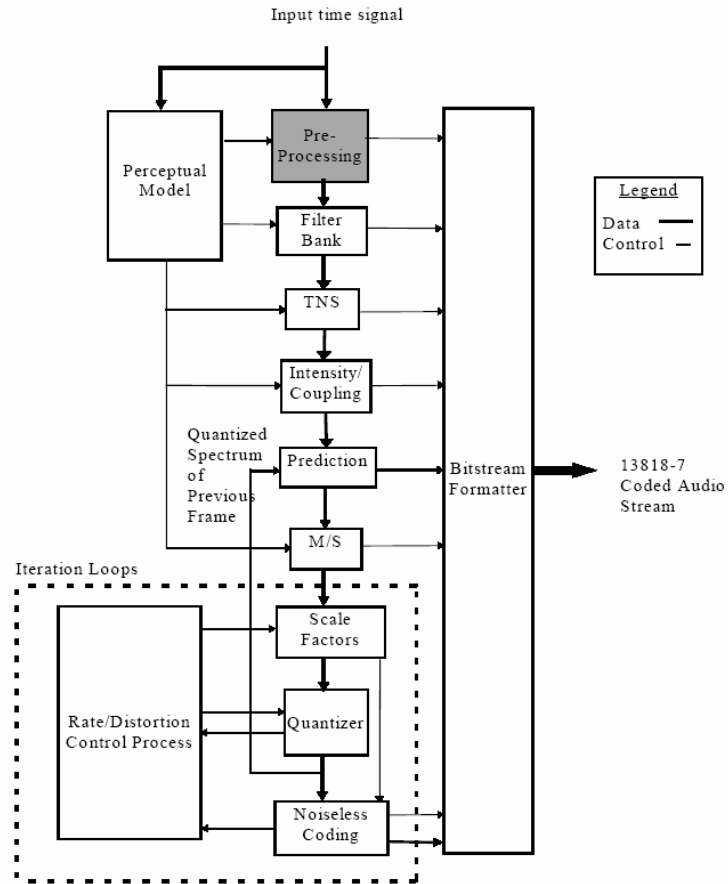


Fig. 2.1 Block diagram for MPEG-2 AAC encoder [1]

2.1.1 Gain Control

The gain control tool receives the time-domain signals, and outputs gain control data and signal whose length is equal of the modified discrete cosine transform (MDCT) window. Fig 2.2 shows the block diagram for the tool. This tool consists of a polyphase quadrature filterbank (PQF), gain detectors and gain modifiers. The PQF divided input signals into four equal bandwidth frequency bands. The gain detectors produce the gain control data which satisfies the bitstream syntax. The gain modifiers control the gain of each signal band. The

gain control tool can be applied to each of four bands independently.

The tool is only available for the SSR profile because of the features of SSR profile. If we need lower bandwidth for output signals, lower sampling rate signals can be obtained by draping the signal from the upper bands of the PQF. The advantage of this scalability is that the decoder complexity can be reduced as the output bandwidth is reduced.

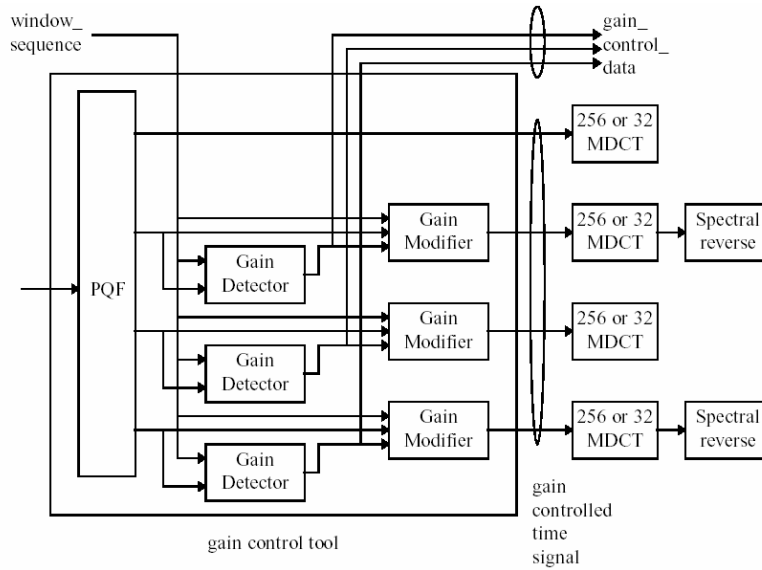


Fig. 2.2 Block diagram of gain control tool for encoder [2]

2.1.2 Filterbank

The filterbank tool converts the time-domain signals into a time-frequency representation. This conversion is done by a MDCT (modified discrete cosine transform), which employs TDAC (time-domain aliasing cancellation) technique.

In the encoder, this filterbank takes in a block of time samples, modulates them by an appropriate window function, and performs the MDCT to ensure good frequency selectivity. Each block of input samples is overlapped by 50% with the immediately preceding block and the following block in order to reduce the boundary effect. Hence in the decoder, adjacent blocks of samples are overlapped and added after inverse MDCT (IMDCT).

The mathematical expression for the MDCT is

$$X_{i,k} = 2 \sum_{n=0}^{N-1} x_{i,n} \cos \left[\frac{2\pi}{N} \left(n + n_0 \right) \left(k + \frac{1}{2} \right) \right], \quad k = 0, 1, \dots, \frac{N}{2} - 1$$

(2.1)

The mathematical expression of the IMDCT is

$$x_{i,n} = \frac{2}{N} \sum_{k=0}^{N/2-1} X_{i,k} \cos \left[\frac{2\pi}{N} (n + n_0) \left(k + \frac{1}{2} \right) \right], \quad n = 0, 1, \dots, N - 1 \quad (2.2)$$

where

- n = sample index
- N = transform block length
- i = block index
- k = coefficient index
- $n_0 = (N/2+1)/2$

Since the window function has a significant effect on the filterbank frequency response, the filterbank has been designed to allow a change in window length and shape to adapt to input signal condition. There are two different lengths and two different shapes for window selection. Relatively short windows suit to signals in transient, and the relatively long ones suit to signals in steady-state. The sine windows are narrow passband selective, and the other choices Kaiser-Bessel Derived (KBD) windows are strong stopband attenuated.

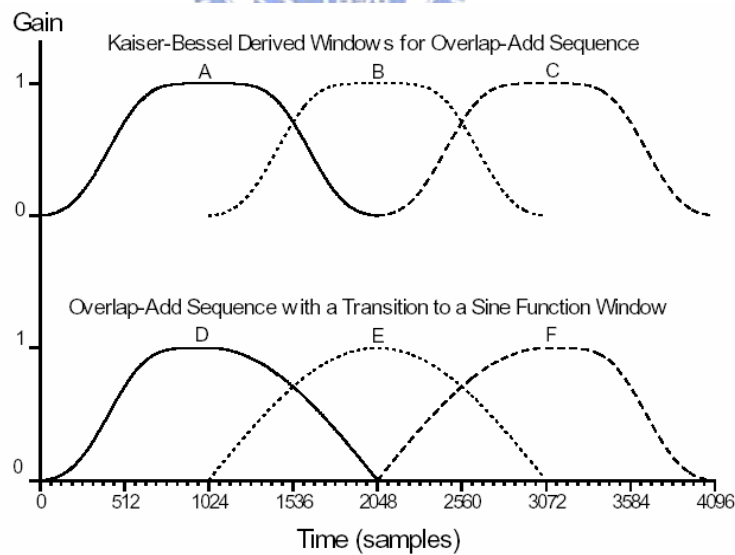


Fig. 2.3 Window shape adaptation process [2]

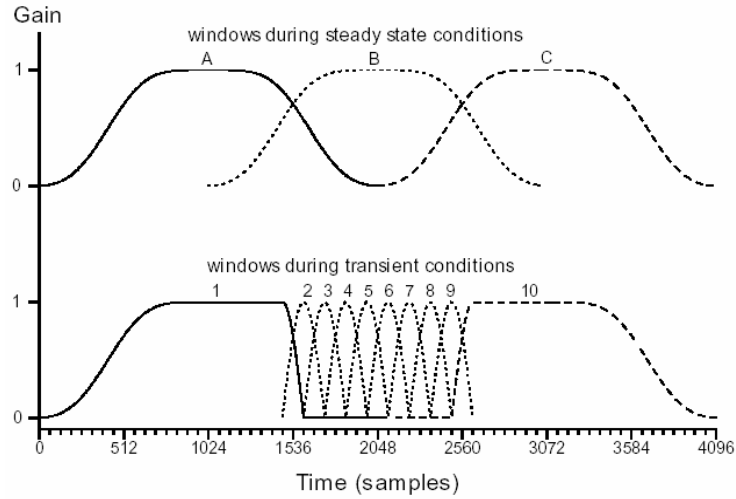


Fig. 2.4 Block switching during transient signal conditions [2]

2.1.3 Temporal Noise Shaping (TNS)

The temporal noise shape (TNS) is used to control the temporal shape of the quantization noise within each window of the transform. This is done by applying a filtering process to parts of the spectral data of each channel.

To handle the transient and pitched signals is a major challenge in audio coding. This is due to the problem of maintaining the masking effect in the reproduced audio signals. Because of the temporal mismatch between masking threshold and quantization noise, the phenomenon is called by “pre-echo” problem. Fig 2.5 illustrates this phenomenon, the left figure shows the original temporal signals in a window, and the right figure shows the quantized spectral coefficients transform to the time domain.

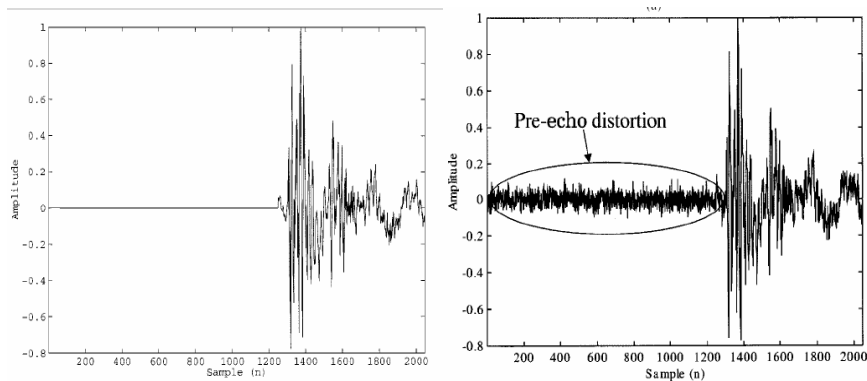
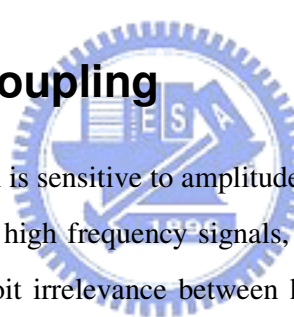


Fig. 2.5 Pre-echo distortion [3]

The duality between time domain and frequency domain is used in predictive coding techniques. The signals with an “unflat” spectrum can be coded efficiently either by directly coding the spectral coefficients or predictive coding the time domain signals. According to the duality property, the signals with an “unflat” time structure, like transient signals, can be coded efficiently either by directly coding time-domain samples or applying predictive coding to the spectral coefficients. The TNS tool uses prediction mechanism over frequency-domain to enhance its temporal resolution.

In addition, if predictive coding is applied to spectral coefficients, the temporal noise will adapt to the temporal signal when decoded. Hence the quantization noise is put into the original signal, and in this way, the problem of temporal noise in transient or pitched signals can be avoided.

2.1.4 Intensity Coupling



The human hearing system is sensitive to amplitude and phase of low frequency signals. It is also sensitive to amplitude of high frequency signals, but insensitive to phase. The intensity coupling tool is used to exploit irrelevance between high frequency signals of each pair of channels. It adds high frequency signals from left and right channel and multiplies to a factor to rescale the result. The intensity signals are used to replace the corresponding left channel high frequency signals, and corresponding signals of the right channel are set to zero.

2.1.5 Prediction Tool

Prediction tool is used for improved redundancy reduction in spectral coefficients. If the spectral coefficients are stationary between adjacent frames, the prediction tool will estimate the possible coefficients in the later blocks by coefficients in the prior ones. Then encode the difference part of these spectral coefficients, the required bits to code these coefficients will be less. If the signals are nonstationary, the short window in the filterbank will be selected, hence prediction tool is only used for long windows.

For each channel, there is one predictor corresponding to the spectral component from the spectral decomposition of the filterbank. The predictor exploits the autocorrelation between the spectral component values of consecutive frames. The predictor coefficients are calculated from preceding quantized spectral components in the encoder. In this case, the spectral component can be recovered in the decoder without other predictor coefficients. A second-order backward-adaptive lattice structure predictor is working on the spectral component values of the two preceding frames. The predictor parameters are adapted to the current signal statistics on a frame-by-frame base, using an LMS-based adaptation algorithm. If prediction is activated, the quantizer is fed with a prediction error instead of the original spectral component, resulting in a higher coding efficiency.

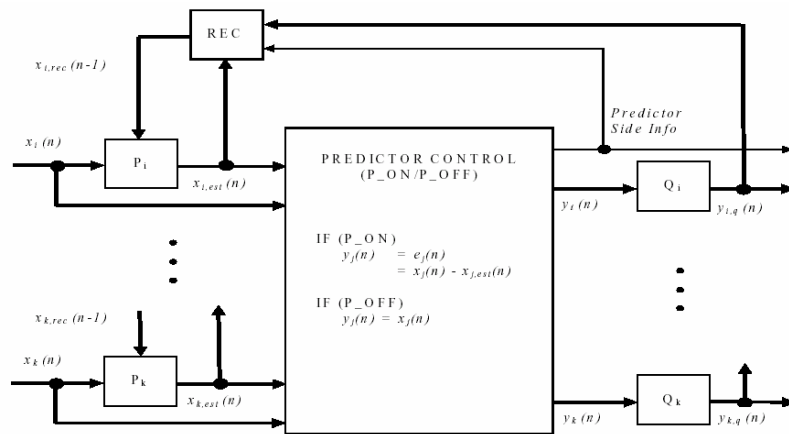


Fig. 2.6 Prediction tool for one scalefactor band [2]

2.1.6 Middle/Side Tool

There are two different choices to code each pair of the multi-channel signals, the original left/right (L/R) signals or the transformed middle/side (M/S) signals. If the high correlated left and right signals could be summed, the require bits to code this signals will be less. Hence in the encoder, the M/S tool will operate when the left and right signals' correlation is higher than a threshold. The M/S tool transform the L/R signals to M/S signals, where the middle signal equals to the sum of left and right signals, and the side signal equals to the difference of left and right ones.

2.1.7 Scalefactors

The human hearing system can be modeled as several over-lapped bandpass filters. With higher central frequency, each filter has larger bandwidth. These bandpass filters are called critical bands. The scalefactors tool divides the spectral coefficients into groups, called scalefactor bands, to imitate critical bands. Each scalefactor band has a scalefactor, and all the spectral coefficients in the scalefactor band are divided by this corresponding scalefactor. By adjusting the scalefactors, quantization noise can be modified to meet the bit-rate and distortion constraints.

2.1.8 Quantization

While all previous tools perform some kind of preprocessing of audio data, the real bit-rate reduction is achieved by the quantization tool. On the one hand, we want to quantize the spectral coefficients in such a way that quantization noise under the masking threshold; on the other hand, we want to limit the number of bits requested to code this quantized spectral coefficients.

There is no standardized strategy for gaining optimum quantization. One important issue is the tuning between the psychoacoustic model and the quantization process. The main advantage of nonuniform quantizer is the built-in noise shaping depending on the spectral coefficient amplitude. The increase of the signal-to-noise ratio with rising signal energy is much lower values than in a linear quantizer.

2.1.9 Noiseless Coding

The noiseless coding is done via clipping spectral coefficients, using maximum number of sections in preliminary Huffman coding, and then merging section to achieve lowest bit count. The input to the noiseless coding tool is a set of 1024 quantized spectral coefficients. Up to four spectral coefficients can be coded separately as magnitude in excess of one, with value of ± 1 left in the quantized coefficients array to carry the sign. The clipped spectral coefficients are coded as integer magnitude and an offset from the base of the coefficient array to mark

their location. Since the side information for carrying the clipped spectral coefficients costs some bits, this compression is applied only if it results in a net saving of bits.

The Huffman coding is used to represent n-tuples of quantized spectral coefficients, with 12 codebooks can be used. The spectral coefficients within n-tuples are ordered from low frequency to high frequency and the n-tuple size can be two or four spectral coefficients. Each codebook specifies the maximum absolute value that it can represent and the n-tuple size. Two codebooks are available for each maximum absolute value, and represent two distinct probability distributions. Most codebooks represent unsigned values in order to save codebook storage. Sign bits of nonzero coefficients are appended to the codeword.

2.2 MPEG-4 AAC Version 1

MPEG-4 AAC Version 1 was approved in 1998 and published in 1999. It has all the tools of MPEG-2 AAC. It includes additional tools such as the long term predictor (LTP) tool, perceptual noise substitution (PNS) tool and transform-domain weighted interlaced vector quantization (TwinVQ) tool. The TwinVQ tool is an alternative tool for the MPEG-4 AAC quantization tool and noiseless coding tool. This new scheme which combined AAC with TwinVQ is officially called "General Audio (GA)." We will introduce these new tools in this section.

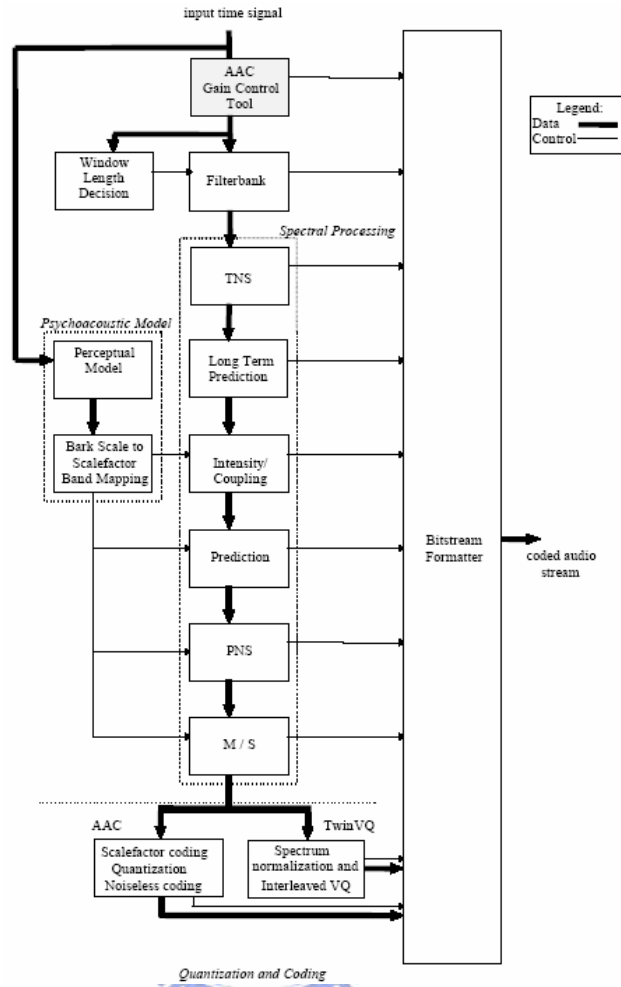


Fig. 2.7 Block diagram of MPEG-4 GA encoder [2]

2.2.1 Long Term Prediction

The long term prediction (LTP) tool uses to exploit the redundancy in the speech signal which is related to the signal periodicity as expressed by the speech pitch. In speech coding, the sounds are produced in a periodical way so that the pitch phenomenon is obvious. Such phenomenon may exist in audio signals as well.

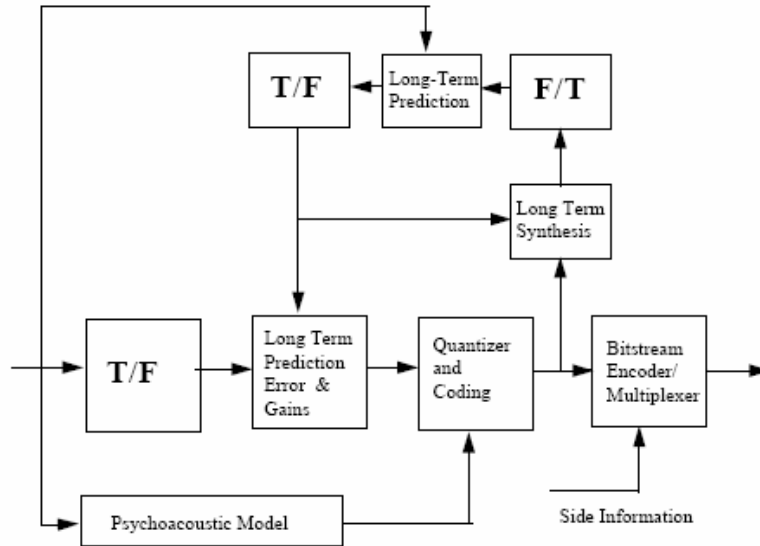


Fig. 2.8 LTP in the MPEG-4 General Audio encoder [2]

The LTP tool performs prediction to adjacent frames while MPEG-2 AAC prediction tool perform prediction on neighboring frequency components. The spectral coefficients transform back to the time-domain representation by inverse filterbank and the associated inverse TNS tool operations. Comparing the locally decoded signal to the input signal, the optimum pitch lag and gain factor can be determined. The difference between the predicted signal and the original signal then is calculated and compared with the original signal. One of them is selected to be coded on a scalefactor band basis depending on which alternative is more favorable.

The LTP tool provides considerable coding gain for stationary harmonic signals as well as some non-harmonic tonal signals. Besides, the LTP tool is much less computational complexity than original prediction tool.

2.2.2 Perceptual Noise Substitution

The perceptual noise substitution (PNS) tool gives a very compact representation of noise-like signals. In this way, the PNS tool provides that increasing of the compression efficiency for some type of input signals.

In the encoder, the noise-like component of the input signal is detected on a scalefactor band basis. If spectral coefficients in a scalefactor band are detected as noise-like signals, they will not be quantized and entropy coded as usual. The noise-like signals omit from the quantization and entropy coding process, but coded and transmitted a noise substitution flag and the total power of them.

In the decoder, a pseudo noise signal with desired total power is inserted for the substituted spectral coefficients. This technique results in high compression efficiency since only a flag and the power information is coded and transmitted rather than whole spectral coefficients in the scalefactor band

2.2.3 TwinVQ

The TwinVQ tool is an alternative quantization/coding kernel. It is designed to provide good coding efficiency at very low bit-rate (16kbps or even lower to 6kbps). The TwinVQ kernel first normalizes the spectral coefficients to a specified range, and then the spectral coefficients are quantized by means of a weighted vector quantization process.

The normalization process is carried out by several schemes such as linear predictive coding (LPC) spectral estimation, periodic component extraction, Bark-scale spectral estimation, and power estimation. As a result, the spectral coefficients are "flattened" and normalized across the frequency axis.

The weighted vector quantization process is carried out by interleaving the normalized spectral coefficients and dividing them into sub-vectors for vector quantization. For each sub-vector, a weighted distortion measure is applied to the conjugate structure VQ which uses a pair of code books. Perceptual control of quantization noise is achieved in this way. The process is shown in Fig 2.9.

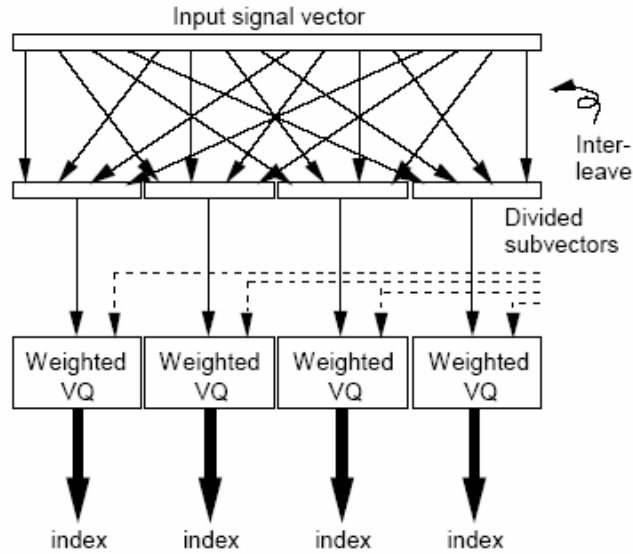


Fig. 2.9 TwinVQ quantization scheme [2]

2.3 MPEG-4 AAC Version 2

MPEG-4 AAC Version 2 was finalized in 1999. Compared to MPEG-4 Version 1, Version 2 adds several new tools in the standard. They are Error Robustness tool, Bit Slice Arithmetic Coding (BSAC) tool, Low Delay AAC (LD-AAC). The BSAC tool is for fine-grain bitrate scalability, and the LD-AAC for coding of general audio signals with low delay. We will introduce these new tools in this section.

2.3.1 Error Robustness

The Error Robustness tools provide improved performance on error-prone transmission channels. The two classes of tools are the Error Resilience (ER) tool and Error Protection (EP) tool.

The ER tool reduces the perceived distortion of the decoded audio signal that is caused by corrupted bits in the bitstream. The following tools are provided to improve the error robustness for several parts of an AAC bitstream frame: Virtual CodeBook (VCB), Reversible Variable Length Coding (RVLC), and Huffman Codeword Reordering (HCR). These tools

allow the application of advanced channel coding techniques, which are adapted to the special needs of the different coding tools.

The EP tool provides Unequal Error Protection (UEP) for MPEG-4 Audio. UEP is an efficient method to improve the error robustness of source coding schemes. It is used by various speech and audio coding systems operating over error-prone channels such as mobile telephone networks or Digital Audio Broadcasting (DAB). The bits of the coded signal representation are first grouped into different classes according to their error sensitivity. Then error protection is individually applied to the different classes, giving better protection to more sensitive bits.

2.3.2 Bit Slice Arithmetic Coding Tool

The Bit-Sliced Arithmetic Coding (BSAC) tool provides efficient small step scalability for the GA coder. This tool is used in combination with the AAC coding tools and replaces the noiseless coding of the quantized spectral data and the scalefactors. The BSAC tool provides scalability in steps of 1 kbps per audio channel, which means 2 kbps steps for a stereo signal. One base layer bitstream and many small enhancement layer bitstreams are used. The base layer contains the general side information, specific side information for the first layer and the audio data of the first layer. The enhancement streams contain only the specific side information and audio data for the corresponding layer.

To obtain fine step scalability, a bit-slicing scheme is applied to the quantized spectral data. First the quantized spectral coefficients are grouped into frequency bands. Each of group contains the quantized spectral coefficients in their binary representation. Then the bits of a group are processed in slices according to their significance. Thus all of the most significant bits (MSB) of the quantized spectral coefficients in each group are processed. Then these bit-slices are encoded by using an arithmetic coding scheme to obtain entropy coding with minimal redundancy. Various arithmetic coding models are provided to cover the different statistics of the bit-slices.

The scheme assigns the bit-slices of the different frequency bands to the enhancement layers. Thus if the decoder processes more enhancement layers, quantized spectral

coefficients are refined by providing more less significant bits (LSB), and the bandwidth is increased by providing bit-slices of the spectral coefficients in higher frequency bands.

2.3.3 Low-Delay Audio Coding

The MPEG-4 General Audio Coder provides very efficient coding of general audio signals at low bitrates. However it has an algorithmic delay of up to several 100ms and is thus not well suited for applications requiring low coding delay, such as real-time bi-directional communication. To enable coding of general audio signals with an algorithmic delay not exceeding 20 ms, MPEG-4 Version 2 specifies a Low-Delay Audio Coder which is derived from MPEG-2/4 Advanced Audio Coding (AAC). It operates at up to 48 kHz sampling rate and uses a frame length of 512 or 480 samples, compared to the 1024 or 960 samples used in standard MPEG-2/4 AAC. Also the size of the window used in the analysis and synthesis filterbank is reduced by a factor of 2. No block switching is used to avoid the “look-ahead” delay due to the block switching decision. To reduce pre-echo phenomenon in case of transient signals, window shape switching is provided instead. For non-transient parts of the signal a sine window is used, while a so-called low overlap window is used in case of transient signals. Use of the bit reservoir is minimized in the encoder in order to reach the desired target delay. As one extreme case, no bit reservoir is used at all.

2.4 MPEG-4 AAC Version 3

MPEG-4 AAC Version 3 was finalized in 2003. Like MPEG-4 Version2, Version 3 adds some new tools to increase the coding efficiency. The main tool is SBR (spectral band replication) tool for a bandwidth extension at low bitrates encodings. This result scheme is called High-Efficiency AAC (HE AAC).

The SBR (spectral band replication) tool improves the performance of low bitrate audio by either increasing the audio bandwidth at a given bitrate or by improving coding efficiency at a given quality level. When the MPEG-4 AAC attaches to SBR tool, the encoders encode

lower frequency bands only, and then the decoders reconstruct the higher frequency bands based on an analysis of the lower frequency bands. Some guidance information may be encoded in the bitstream at a very low bitrate to ensure the reconstructed signal accurate. The reconstruction is efficient for harmonic as well as for noise-like components and allows for proper shaping in the time domain as well as in the frequency domain. As a result, SBR tool allows a very large bandwidth audio coding at low bitrates.

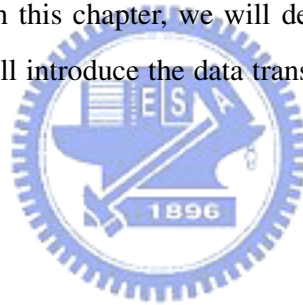


Chapter 3

Introduction to

DSP/FPGA

In our system, we will use Digital Signal Processor/Field Programmable Gate Array (DSP/FPGA) to implement MPEG-4 AAC encoder and decoder. The DSP baseboard is made by Innovative Integration's Quixote, which houses Texas Instruments' TMS320C6416 DSP and Xilinx Virtex-II FPGA. In this chapter, we will describe DSP baseboard, DSP chip and FPGA chip. At the end, we will introduce the data transmission between the Host PC and the DSP/FPGA



3.1 DSP Baseboard

Quixote combines one TMS320C6416 600MHz 32-bit fixed-point DSP with a Xilinx Virtex-II XC2V2000/6000 FPGA on the DSP baseboard. Utilizing the signal processing technology to provide processing flexibility, efficiency and deliver high performance. Quixote has 32MB SDRAM for use by DSP and 4 or 8Mbytes zero bus turnaround (ZBT) SBSRAM for use by FPGA. Developers can build complex signal processing systems by integrating these reusable logic designs with their specific application logic.

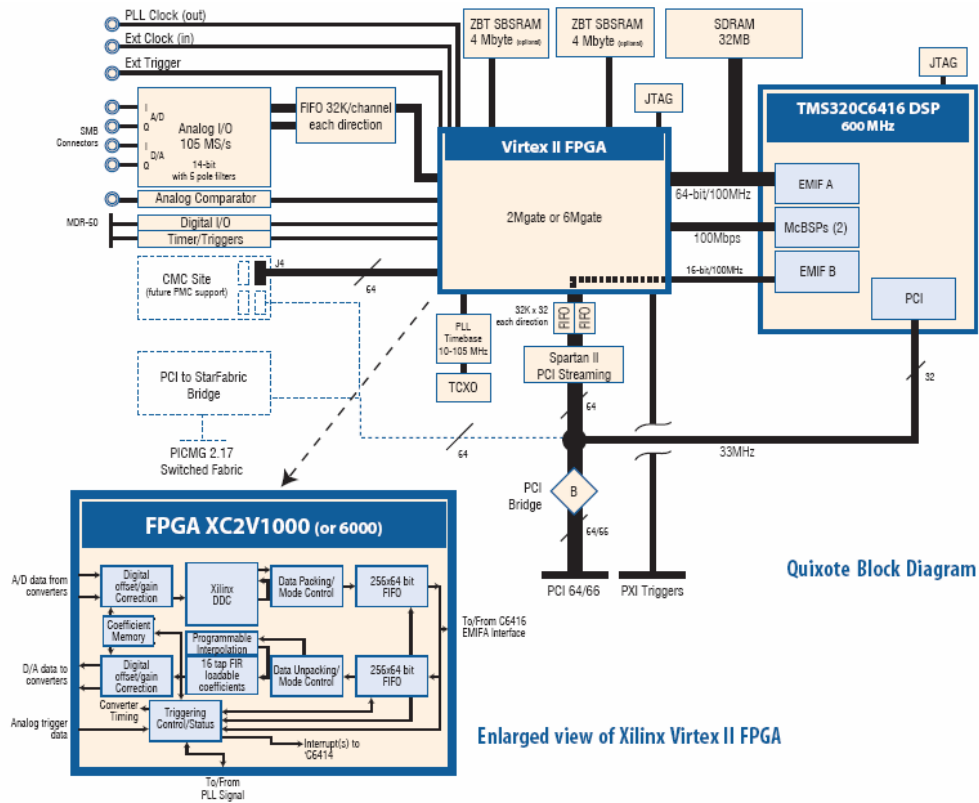


Fig. 3.1 Block Diagram of Quixote [5]

3.2 DSP Chip

The TMS320C64x fixed-point DSP is using the VelociTI architecture. The VelociTI architecture of the C6000 platform of devices use advanced VLIW (very long instruction word) to achieve high performance through increased instruction-level parallelism, performing multiple instructions during a single cycle. Parallelism is the key to extremely high performance, taking the DSP well beyond the performance capabilities of traditional superscalar designs. VelociTI is a highly deterministic architecture, having few restrictions on how or when instructions are fetched, executed, or stored. It is this architectural flexibility that is the key to the breakthrough efficiency levels of the TMS320C6000 Optimizing C compiler. VelociTI advanced features include.

- Instruction packing: reduced code size
- All instructions can operate conditionally: flexibility of code
- Variable-width instructions: flexibility of data types
- Fully pipelined branches: zero-overhead branching

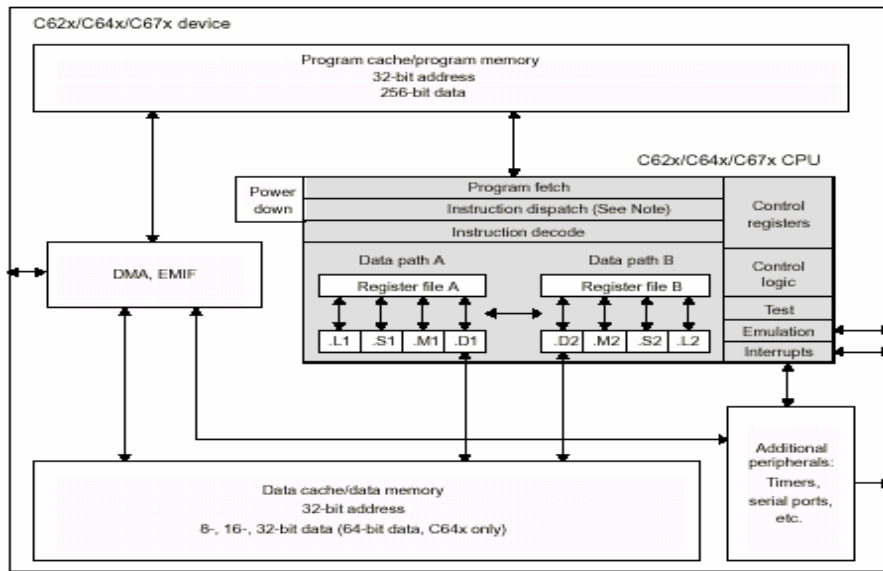


Fig 3.2 Block diagram of TMS320C6x DSP [6]

TMS320C6416 has internal memory includes a two-level cache architecture with 16 KB of L1 data cache, 16 KB of L1 program cache, and 1 MB L2 cache for data/program allocation. On-chip peripherals include two multi-channel buffered serial ports (McBSPs), two timers, a 16-bit host port interface (HPI), and 32-bit external memory interface (EMIF). Internal buses include a 32-bit program address bus, a 256-bit program data bus to accommodate eight 32-bit instructions, two 32-bit data address buses, two 64-bit data buses, and two 64-bit store data buses. With 32-bit address bus, the total memory space is 4 GB, including four external memory spaces: CE0, CE1, CE2, and CE3. We will introduce several important parts in this section.

3.2.1 Central Processing Unit (CPU)

Fig. 3.2 shows the CPU, and it contains

- Program fetch unit
- Instruction dispatch unit, advanced instruction packing
- Instruction decode unit
- Two data path, each with four functional units
- 64 32-bit registers

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

The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, and .D) and 32 32-bit general-purpose registers. Fig. 3.3 shows the comparison of C62x/C67x with C64x CPU.



3.2.2 Data Path

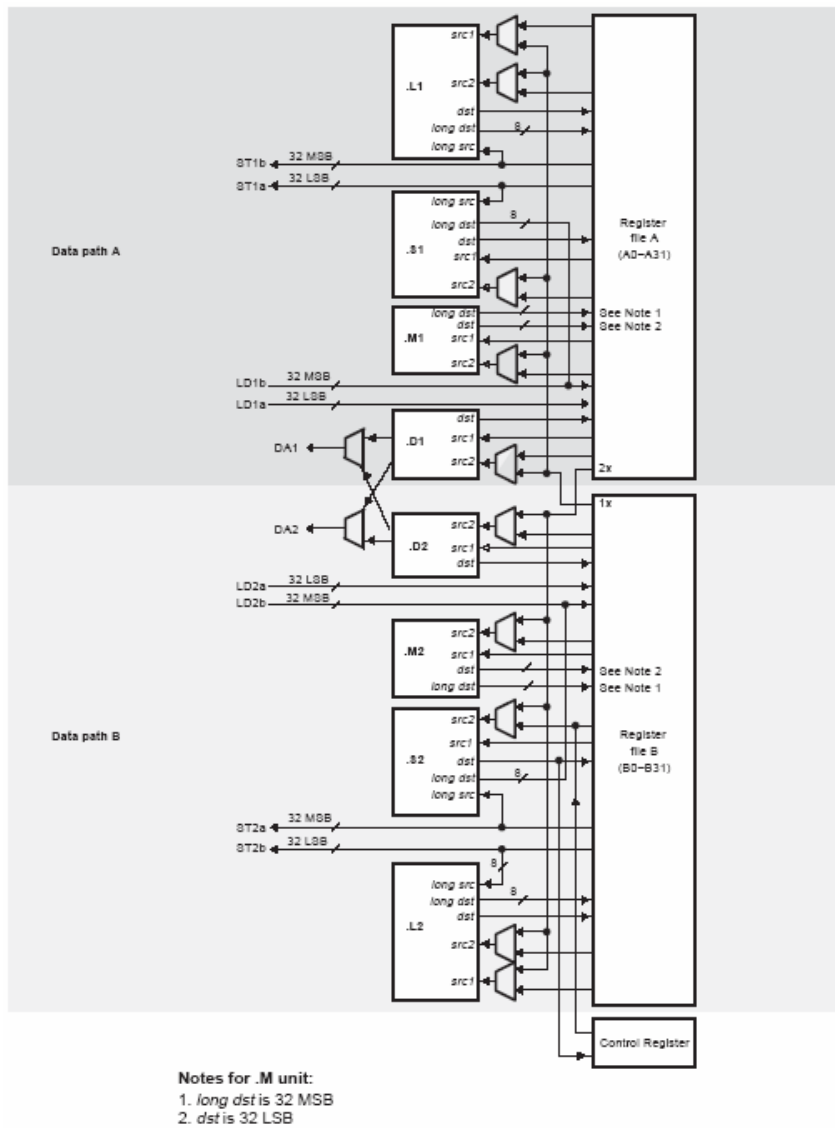


Fig 3.3 TMS320C64x CPU Data Path [6]

There are two general-purpose register files (A and B) in the C6000 data paths. The C64x DSP register is double the number of general-purpose registers that are in the C62x/C67x cores, with 32 32-bit registers (A0-A31 for file A and B0-B31 for file B).

There are eight independent functional units divided into two data paths. Each path has a unit for multiplication operations (.M), for logical and arithmetic operations (.L), for branch, bit manipulation, and arithmetic operations (.S), and for loading/storing and arithmetic

operations (.D). The .S and .L units are for arithmetic, logical, and branch instructions. All data transfers make use of the .D units. Two cross-paths (1x and 2x) allow functional units from one data path to access a 32-bit operand from the register file on the opposite side. It can be a maximum of two cross-path source reads per cycle. Fig. 3.4 and 3.5 show the functional unit and its operations.

Functional Unit	Fixed-Point Operations	Floating-Point Operations
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits Byte shifts Data packing/unpacking 5-bit constant generation Dual 16-bit arithmetic operations Quad 8-bit arithmetic operations Dual 16-bit min/max operations Quad 8-bit min/max operations	Arithmetic operations DP → SP, INT → DP, INT → SP conversion operations
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from control register file (.S2 only) Byte shifts Data packing/unpacking Dual 16-bit compare operations Quad 8-bit compare operations Dual 16-bit shift operations Dual 16-bit saturated arithmetic operations Quad 8-bit saturated arithmetic operations	Compare Reciprocal and reciprocal square-root operations Absolute value operations SP → DP conversion operations

Fig. 3.4 Functional Units and Operations Performed [7]

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

Fig. 3.5 Functional Units and Operations Performed (Cont.) [7]



3.2.3 Pipeline Operation

Pipelining is the key feature to get parallel instructions working properly, requiring careful timing. There are three stages of pipelining: program fetch, decode, and execute, and each stage contains several phases. We will describe the function of the three stages and their associated multiple phases in the section.

The fetch stage is composed of four phases

- PG: Program address generate
- PS: Program address send
- PW: Program address ready wait
- PR: Program fetch packet receive

During the PG phase, the program address is generated in the CPU. In the PS phase, the program address is sent to memory. In the PW phase, a memory read occurs. Finally, in the PR phase, the fetch packet is received at the CPU.

The decode stage is composed of two phases.

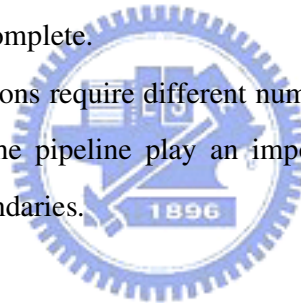
- DP: Instruction dispatch
- DC: Instruction decode

During the DP phase, the instructions in execute packet are assigned to the appropriate functional units. In the DC phase, the source registers, destination registers, and associated paths are decoded for the execution of the instructions in the functional units.

The execute stage is composed of five phases.

- E1: Single cycle instruction complete.
- E2: Multiply instruction complete.
- E3: Store instruction complete.
- E4: Multiply extensions instruction complete.
- E5: Load instruction complete.

Different types of instructions require different numbers of these phases to complete their execution. These phases of the pipeline play an important role in your understanding the device state at CPU cycle boundaries.



3.2.4 Internal Memory

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

3.2.5 External Memory and Peripheral Options

The external memory and peripheral options of C6416 contain

- 16 KB data L1 cache

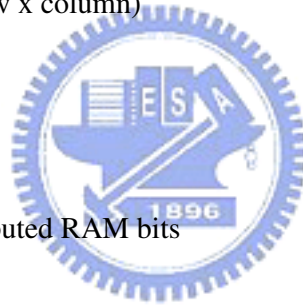
- 16 KB program L1 cache
- 1M L2 cache
- 64 EDMA channels
- 3 32-bit timers

3.3 FPGA

The Xilinx Virtex-II FPGA is made by 0.15 μ m, 8-layer metal process; it offers logic performance in excess of 300MHz. We will introduce the FPGA logic in this section.

Virtex-II XC2V2000 FPGA contains

- 2M system gates
- 56 x 48 CLB array (row x column)
- 10752 slices
- 24192 logic cells
- 21504 CLB flip-flops
- 336K maximum distributed RAM bits



Virtex-II XC2V6000 FPGA contains

- 6M system gates
- 96 x 88 CLB array (row x column)
- 33792 slices
- 76032 logic cells
- 675844 CLB flip-flops
- 1056K maximum distributed RAM bits

Configurable Logic Blocks (CLB) is a block of logic surrounded by routing resources. The functional elements are need to logic circuits. One CLB contains four slices; each slice contains two Logic Cells (LC); each LC includes a 4-input function generator, carry logic, and a storage element.

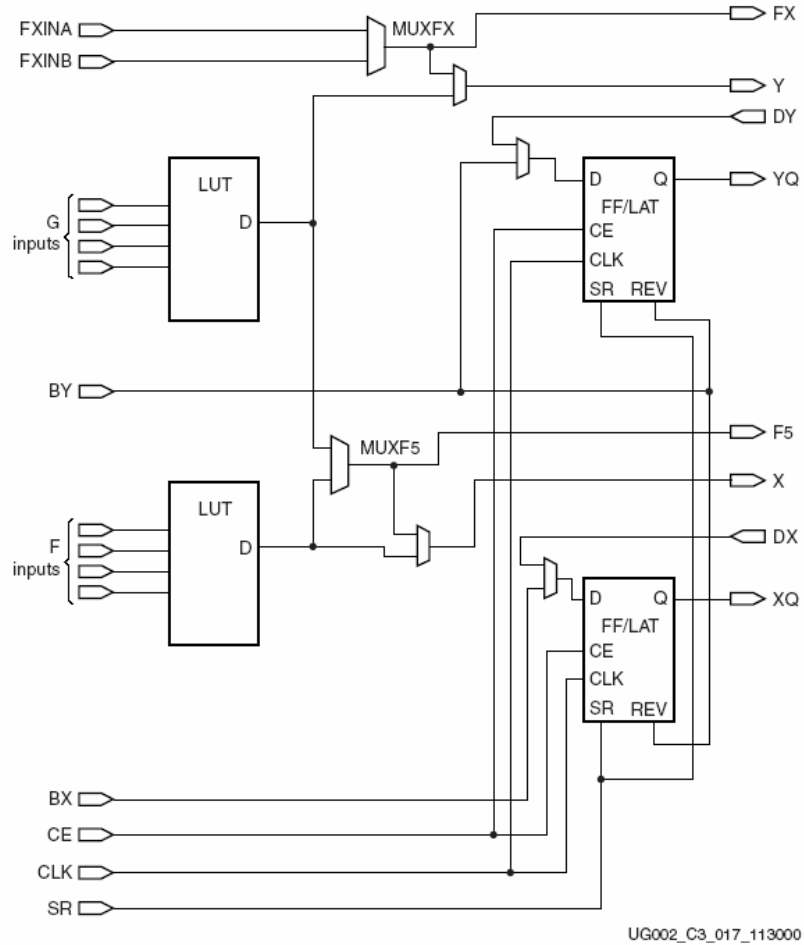


Fig 3.6 General Slice Diagram [10]

The synthesizer of the Xilinx FPGA is the Xilinx ISE 6.1. The simulation result was reference by the synthesizer report and the P&R report in the ISE.

3.4 Data Transmission Mechanism

In this section, we will describe the transmission mechanism between the Host PC and the DSP/FPGA. There are two data transmission mechanism for the DSP baseboard. That is message interface and the streaming interface.

3.4.1 Message Interface

The DSP and Host PC have a lower bandwidth communications link for sending commands or side information between host PC and target DSP. Software is provided to build a packet-based message system between the target DSP and the Host PC. A set of sixteen mailboxes in each direction to and from Host PC are shared with DSP to allow for an efficient message mechanism that complements the streaming interface. The maximum data rate is 56 kbps, and the higher data rate requirements should use the streaming interface.

3.4.2 Streaming Interface

The primary streaming interface is based on a streaming model where logically data is an infinite stream between the source and destination. This model is more efficient because the signaling between the two parties in the transfer can be kept to a minimum and transfers can be buffered for maximum throughput. On the other hand, the streaming model has relatively high latency for a particular piece of data. This is because a data item may remain in internal buffering until subsequent data accumulates to allow for an efficient transfer.



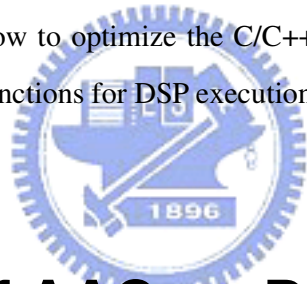
Chapter 4

MPEG-4 AAC Decoder

Implementation and

Optimization on DSP

In this chapter, we will describe the MPEG-4 AAC implementation and optimization on DSP. We will first describe how to optimize the C/C++ code for DSP architecture, and then discuss how to optimize the functions for DSP execution.



4.1 Profile of AAC on DSP

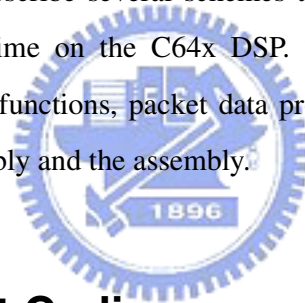
We do the essential modification on the MPEG-4 AAC source C code, and then implement this modified code on DSP. We first optimize the most computational complexity parts of these modified codes. We profile this code by TI CCS profiler. The length of the test sequence is about 0.95 second, and the C64x DSP takes 0.18 second to decode this test sequence. Table 4.1 shows the profile result. We find that the IMDCT and the Huffman decoding require 66% and 21% of total clock cycle respectively. Hence, we optimize these two functions first.

Function	Clock Cycle	Percent (%)
Total	107506166	100
IMDCT	70848599	66
Huffman Decoding	22431737	21
M/S stereo	1224011	1
PNS	168978	0
Intensity	81388	0
Others	12751453	12

Table 4.1 Profile of AAC decoding on C64x DSP

4.2 Optimizing C/C++ Code

In this section, we will describe several schemes that we can optimize our C/C++ code and reduce DSP execution time on the C64x DSP. These techniques include the use of fixed-point coding, intrinsic functions, packet data processing, loop unrolling and software pipelining, using linear assembly and the assembly.



4.2.1 Fixed-point Coding

The C64x DSP is a fixed-point processor, so it can do fixed-point processing only. Although the C64x DSP can simulate floating-point processing, it takes a lot of extra clock cycle to do the same job. Table 4.2 is the test results of C64x DSP processing time of assembly instructions “add” and “mul” for different datatypes. It is the processing time without data transfer between external memory and register. The “char”, “short”, “int” and “long” are the fixed-point datatypes, and the “float” and “double” are the floating-point datatypes. We can see clearly that floating-point datatypes need more than 10 times longer time than fixed-point datatypes in computation time. To optimize our code on the C64x DSP, we need to convert the datatypes from floating-point to fixed-point first. But this modification has to quantize the input data, so it may lose some accuracy. We need to do the quantization noise analysis when we want to do the datatype conversion.

Assembly Instruction	Char 8-bit	short 16-bit	int 32-bit	long 40-bit	float 32-bit	double 64-bit
add	1	1	1	2	77	146
mul	2	2	6	8	54	69

Table 4.2 Processing time on the C64x DSP with different datatypes

4.2.2 Using Intrinsic Functions

TI provides many intrinsic functions to increase the efficiency of code on the C6000 series DSP. The intrinsic functions are optimized code by the knowledge and technique of DSP architecture, and it can be recognize by TI CCS compiler only. So if the C/C++ instructions or functions have corresponding intrinsic functions, we can replace them by intrinsic functions directly. The modification can make our code more efficient substantially. Fig 4.1 shows a part of the intrinsic functions for the C6000 series DSP, and some intrinsic functions are only in the specific DSP.



C Compiler Intrinsic	Assembly Instruction	Description	Device
<code>int _abs(int src2);</code> <code>int _labs(long src2);</code>	ABS	Returns the saturated absolute value of src2.	
<code>int _abs2 (int src2);</code>	ABS2	Calculates the absolute value for each 16-bit value.	'C64x
<code>int _add2(int src1, int src2);</code>	ADD2	Adds the upper and lower halves of src1 to the upper and lower halves of src2 and returns the result. Any overflow from the lower half add will not affect the upper half add.	
<code>int _add4 (int src1, int src2);</code>	ADD4	Performs 2s-complement addition to pairs of packed 8-bit numbers.	'C64x

Fig 4.1 Intrinsic functions of the TI C6000 series DSP (Part.) [6]

4.2.3 Packet Data Processing

The C64x DSP is a 32-bit fixed-point processor, which suit to 32-bit data operation. Although it can do 8-bit, or 16-bit data operations, it will waste some processor resource. So if we can place four 8-bit data or two 16-bit data in a 32-bit space, we can do four or two

operations in one clock cycle. It can improve the code efficiency substantially. One another thing should be mentioned that some of the intrinsic functions have similar way to enhance the efficiency.

4.2.4 Loop Unrolling and Software pipelining

Software pipelining is a scheme to generate efficient assembly code by the compiler so that most of the functional units are utilized within one cycle. For the TI CCS compiler, we can enable the software pipelining function operate or not. If our codes have conditional instructions, sometimes the compiler may not be sure that the branch will be happen or not. It may waste some clock cycles to wait for the decision of branch operation. So if we can unroll the loop, it will avoid some of the overhead for branching. Then the software pipelining will have more efficient result. Besides, we can add some compiler constrains, which tell the compiler that the branch will taken or not, or the loop will run a number of times at least.

4.2.5 Linear Assembly and Assembly

When we are not satisfied with the efficiency of assembly codes which generated by the TI CCS compiler, we can convert some function into linear assembly or optimize the assembly directly. The linear assembly is the input of TI CCS assembly optimizer, and it does not need to specify the parallel instructions, pipeline latency, register usage, and which functional units is being used.

Generally speaking, this scheme is too detail and too time consumption in practice. If we consider project development time, we may skip this scheme. Unless we have strict constrains in processor performance and we have no other algorithm selection, we will do this scheme at last.

4.3 Huffman Decoding

Generally speaking, the architecture of Huffman decoder can be classified into the sequential model and the parallel model [12]. The sequential model reads in one bit in one clock cycle, so it has a fixed input rate. The parallel model outputs one codeword in one clock cycle, so it has a fixed output rate. Fig. 4.2 and 4.3 show the block diagrams of these two models.

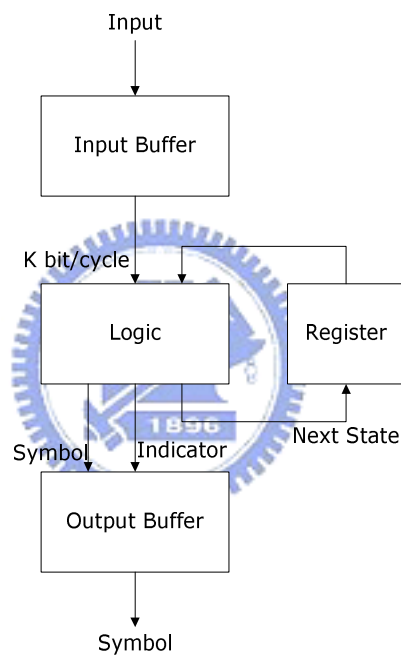


Fig 4.2 Sequential model of Huffman decoder [12]

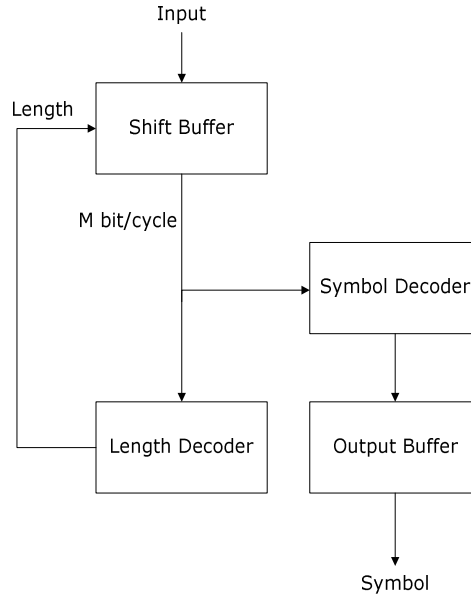


Fig 4.3 Parallel model of Huffman decoder [12]

Because the Huffman code is a variable length code, it means that the codeword is not fixed length for each symbol. Hence the DSP can not know the number of bits in the each codeword in advance. The DSP has to fetch one bit in one clock cycle and compare it with the stored patterns. If there is no matched pattern, the DSP has to fetch the next bit in the next clock cycle and compare with the patterns again. It will take many clock cycles to do the job. The Huffman decoder is restricted by the DSP processing structure, so it belongs to sequential model. We do not find an efficient algorithm for the DSP Huffman decoding scheme, so we plan to implement the Huffman decoding in the FPGA to enhance the performance of total system.

4.4 IMDCT

IMDCT takes the most part of the DSP processing time in an AAC decoder, so we want to optimize this part to improve total system performance. At first, we will describe the efficient way to use $N/4$ -point IFFT to replace the IMDCT. And then we will discuss the architecture of IFFT. At last, we will describe the implementation and optimization of IMDCT on DSP.

4.4.1 N/4-point FFT Algorithm for MDCT

We will discuss N/4-point FFT algorithm for MDCT. Since the processing of $Y_{i,k}$ and $x_{i,n}$ requires a very heavy computational load, we want to find the faster algorithm to replace the original equation. For the fast MDCT algorithm, P. Duhamel had suggested a fast algorithm which uses N/4-point complex FFT (Fast Fourier Transform) to replace MDCT [14]. The key point is that Duhamel found the relationship between N-point MDCT and N/4-point complex FFT. We can thus use the efficient FFT algorithm to enhance the performance of IMDCT. The relationship is valid for N-point IMDCT and N/4-point IFFT.

We will describe the forward operation steps here, and the derivation of this algorithm can be found in Appendix A.

1. Compute $z_n = (x_{i,2n} - x_{i,N/2-1-2n}) + j(x_{i,N-1-2n} + x_{i,N/2+2n})$

2. Multiply the pre-twiddle: $z'_n = z_n W_{4N}^{-(4n+1)}$, $n = 0, 1, \dots, N/4 - 1$

Where $W_{4N} = \cos(2\pi/4N) - j \sin(2\pi/4N)$

3. Do N/4-point complex FFT: $Z'_k = FFT\{z'_n\}$

4. Multiply the post-twiddle: $Z_k = ((-1)^{k+1} W_8^{-1} W_N^{-k}) Z'_k$, $k = 0, 1, \dots, N/4 - 1$

5. The coefficients $Y_{i,2k}$ are found in the imaginary part of Z_k , and the coefficients $Y_{i,2k+N/2}$ are found in the real part of Z_k . The odd part coefficients can be obtained from $Y_{i,k-1} = -Y_{i,N-k}$

We summarize the fast MDCT algorithm by the flow diagram shown in Fig 4.4.

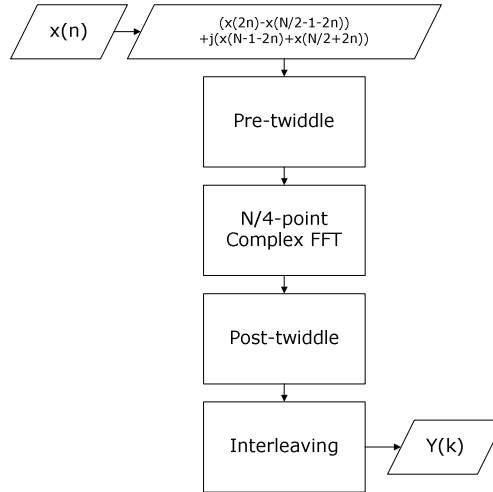


Fig 4.4 Fast MDCT algorithm

The inverse operation steps are in a similar way.

1. Compute $Z_k = -Y_{i,2k} + jY_{i,N/2-1-2k}$
2. Multiply the pre-twiddle: $Z'_k = ((-1)^{k+1} W_8^{-1} W_N^k) Z_k, \quad k = 0, 1, \dots, N/4 - 1$
3. Do N/4-point complex IFFT: $z_n = IFFT\{Z'_k\}$
4. Multiply the post-twiddle: $z'_n = z_n W_{4N}^{(4n+1)}, \quad n = 0, 1, \dots, N/4 - 1$
5. In the range of n from 1 to N/4, the coefficients $x_{i,3N/4-1-2n}$ are found in the imaginary part of z_n , and the coefficients $x_{i,N/4+2n}$ are found in the real part of z_n . In the range of n from 1 to N/8, the coefficients $x_{i,3N/4+2n}$ are found in the imaginary part of z_n , and the coefficients $x_{i,N/4-1-2n}$ are found in the negative of real part of z_n . At last, in the range of n from N/8 to N/4, the coefficients $x_{i,2n-N/4}$ are found in the negative of imaginary part of z_n , and the coefficients $x_{i,5N/4-1-2n}$ are found in the real part of z_n .

We summarize the fast IMDCT algorithm by the flow diagram shown in Fig 4.5.

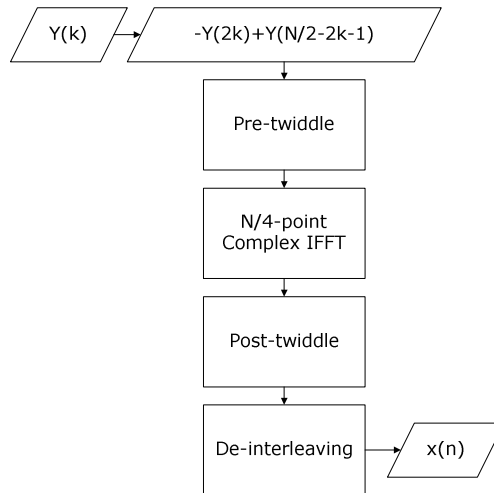


Fig 4.5 Fast IMDCT algorithm

4.4.2 Radix-2³ FFT

There are many FFT algorithms which have been derived in recent years [18]. The radix-2 FFT has the best accuracy, but requires most computations, and the split-radix FFT has fewer computations, but requires irregular butterfly architecture [15]. S. He suggested an FFT algorithm called radix-2² in 1996. It combined radix-2/4 FFT and radix-2 FFT in a processing element (PE), so it has a more regular butterfly architecture than the split-radix FFT and needs fewer computations than radix-2 FFT. But the radix-2² FFT is suit to the 4N-point only, and our requirement for IFFT is 512-point for long window and 64-point for short window. So we can use radix-2³ FFT which derived form radix-2² FFT is suit to 8N-point only.

Fig. 4.6 shows the butterfly of 8-point radix-2 FFT and Fig. 4.7 shows the butterflies of a radix-2³ PE. We can see the number of twiddle factor multiplication is decreased in the data flow graphs. Fig. 4.8 shows the combined split-radix FFT in a radix-2³ PE. We can see the regular architecture of butterflies than split-radix. Table 4.3 shows the computational complexity of radix-2 and radix-2³ FFT algorithms.

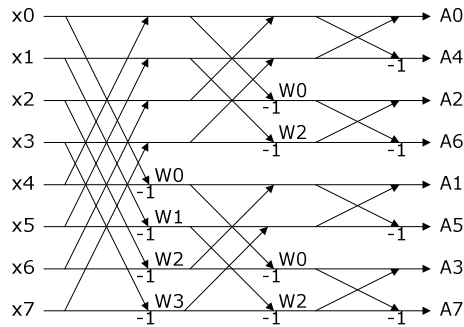


Fig. 4.6 Butterflies for 8-point radix-2 FFT

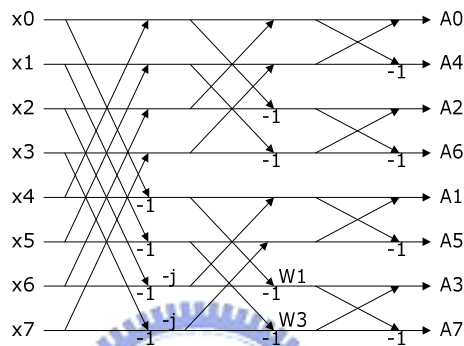


Fig. 4.7 Butterflies for a radix- 2^3 FFT PE

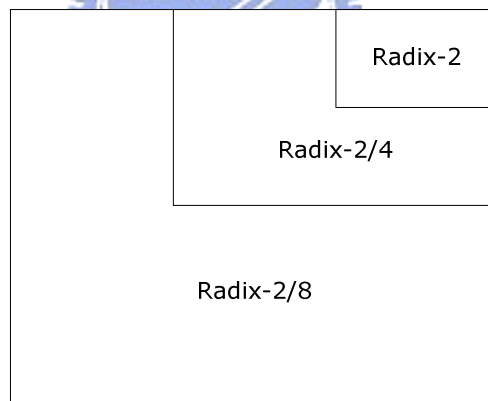


Fig. 4.8 Simplified data flow graph for 8-point radix- 2^3 FFT

N	Radix-2		Radix- 2^3	
	Complex multiplication	Complex addition	Complex multiplication	Complex Addition
8	8	24	2	24
64	160	384	96	384
512	2048	4608	1408	4608
4096	22528	49152	16384	49152

Table 4.3 Comparison of computational load of FFT

4.4.3 Implementation of IMDCT with Radix-2 IFFT

We first code the 512-point IMDCT with radix-2 IFFT architecture in double datatype to ensure the function is correct for the reasonable input data range. After the function is verified, we modified the datatype from floating-point to fixed-point and calculate the data precision loss in SNR (signal-to-noise ratio). In the fixed-point edition, we multiply a factor of 4096 to all twiddle factors.

	Code Size	Clock Cycle
Double	9972	6344125
Int	4848	2070390
Short	4248	2078066

Table 4.4 DSP implementation result of different datatypes

	SNR
Double	278.49dB
Int	101.42dB
Short	100.31dB

Table 4.5 SNR of IMDCT of different datatypes

4.4.4 Implementation of IMDCT with Radix-2³ IFFT

Then we code the 512-point IMDCT with the radix-2³ IFFT architecture in double datatype to ensure the function is correct in the reasonable input data range. Then we modified the register datatype from floating-point to fixed-point. The data precision loss is the same with the radix-2 FFT. In the fixed-point edition, we multiply a factor of 4096 to all twiddle factors, and multiply a factor of 256 to the $\sqrt{2}/2$ in the radix-2³ PE. The original floating-point datatype edition is slower than radix-2 IFFT might influenced by the coding style of the two architectures.

	Code Size	Clock Cycle
Double	8616	12889084
Int	7716	243074
Short	7120	225485

Table 4.6 DSP implementation result of different datatypes

	SNR
Double	278.44dB
Int	83.55dB
Short	83.45dB

Table 4.7 SNR of IMDCT of different datatypes

4.4.5 Modifying the Data Calculation Order

We want to the data in the register can be used twice after they are fetch from memory. So we modified the C/C++ code for the data calculation order in each stage. The original calculation order is from the top to the down in the data flow graph. We calculate the first butterfly's two output data, and then calculate the next butterfly's two output data. Fig. 4.9 shows the calculation order of the modified edition. The number in the parentheses is the calculation order. In this way, the compiler generates the assembly code which can use the data more efficiency.

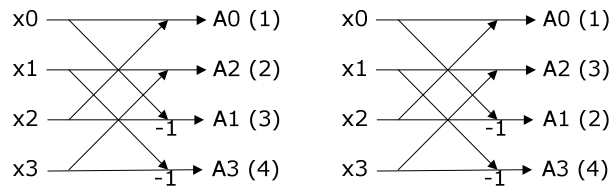


Fig. 4.9 Comparison of the old (left) and new (right) data calculation order

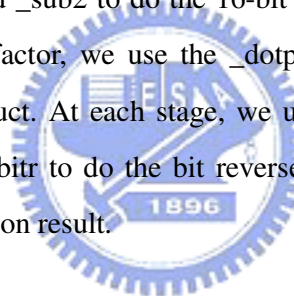
	Code Size	Clock Cycle
Original	7120	225485
Optimized	7852	77547

Table 4.8 DSP implementation results of the modified data calculation order

4.4.6 Using Intrinsic Functions

Since we use the “short” datatype to represent the data in the IMDCT, we may put two 16-bit data in a 32-bit register to improve the performance as packet data processing. At first, we try to use shift the first 16-bit data than add the second 16-bit data into a 32-bit data space. Use one intrinsic function to process these data, and then put the result into two 16-bit data. But the result of this modification is slower than the original version because the data transfer takes too many clock cycles.

So we modify the total IFFT architecture. Put the real part into the 16-bit MSB (maximum significant bit) of 32-bit space, and the imaginary part into the 16-bit LSB (least significant bit). Then use intrinsic functions to do all data process in the IFFT. Fig. 4.10 shows the intrinsic functions we use. At first, we use `_pack2` to put two 16-bit data into a 32-bit space. Then we use `_add2` and `_sub2` to do the 16-bit addition or subtraction. When the data needs to multiply a twiddle factor, we use the `_dotp2` or `_doptn2` to calculate the sum of product or difference of product. At each stage, we use the `_shr2` to divide the data by the factor of 2. At last, we use `_bitr` to do the bit reverse and put the output data in sequence. Table 4.9 shows the modification result.



C Compiler Intrinsic	Assembly Instruction	Description	Device
<code>int _add2(int src1, int src2);</code>	ADD2	Adds the upper and lower halves of src1 to the upper and lower halves of src2 and returns the result. Any overflow from the lower half add will not affect the upper half add.	
<code>int _sub2(int src1, int src2);</code>	SUB2	Subtracts the upper and lower halves of src2 from the upper and lower halves of src1, and returns the result. Any borrowing from the lower half subtract does not affect the upper half subtract.	
<code>int _dotp2 (int src1, int src2);</code> <code>double _ldotp2 (int src1, int src2);</code>	DOTP2 LDOTP2	The product of signed lower 16-bit values of src1 and src2 is added to the product of signed upper 16-bit values of src1 and src2.	'C64x
<code>int _dotpn2 (int src1, int src2);</code>	DOTPN2	The product of signed lower 16-bit values of src1 and src2 is subtracted from the product of signed upper 16-bit values of src1 and src2.	'C64x
<code>unsigned _pack2 (uint src1, uint src2);</code> <code>unsigned _packh2 (uint src1, uint src2);</code>	PACK2 PACKH2	The lower/upper half-words of src1 and src2 are placed in the return value.	'C64x
<code>int _shr2 (int src2, uint src1);</code> <code>unsigned _shru2 (uint src2, uint src1);</code>	SHR2 SHRU2	For each 16-bit quantity in src2, the quantity is arithmetically or logically shifted right by src1 number of bits. src2 can contain signed or unsigned values.	'C64x
<code>unsigned _bitr (uint src2);</code>	BITR	Reverses the order of the bits.	'C64x

Fig. 4.10 Intrinsic functions we used [6]

	Code Size	Clock Cycle
Original	7852	77547
Optimized	8480	24307

Table 4.9 DSP implementation results of using intrinsic functions

4.4.7 IMDCT Implementation Results

We have implemented and optimized the MPEG-AAC IMDCT on DSP. Table 4.10 shows the final optimized results. If the sampling rate is 44.1 kHz, it has to process 43 frames in one second for real time decoding. The final optimized IMDCT can process about 24684 frames

in one second on C64x DSP. It is about 530 times faster than the original version.

	Code Size	Clock Cycle
Original	8616	12889084
Optimized	8480	24307

Table 4.10 DSP implementation results of IMDCT

Function	void DSP_ifft32x32(const int * restrict w, int nx, int * restrict x, int * restrict y)	
Arguments	w[2*nx]	Pointer to complex 32-bit FFT coefficients.
	nx	Length of FFT in complex samples. Must be power of 2 or 4, and $16 \leq nx \leq 32768$.
	x[2*nx]	Pointer to complex 32-bit data input.
	y[2*nx]	Pointer to complex 32-bit data output.

Fig. 4.11 TI IFFT library [7]

Then we compare the modification IMDCT to the IMDCT with TI IFFT library as shown in Fig. 4.11. Table 4.11 shows the comparison of the modification IMDCT and the IMDCT using TI IFFT library. The performance has reached about 81% of the IMDCT with TI IFFT library.

	Code Size	Clock Cycle
WithTI IFFT	1492	19897
Optimized	8480	24307

Table 4.11 Comparison of modification IMDCT and IMDCT with TI IFFT library

4.5 Implementation on DSP

We has implemented and optimized MPEG-4 AAC on TI C64x DSP. The optimized result has been shown in Table 4.12. Using the ITU-R BS.1387 PEAQ (perceptual evaluation of audio quality) defined ODG (objective difference grade), we test some sequences on the modified MPEG-4 AAC decoder. The first test sequence is “guitar”; it has sounds variations and is more complex. The second test sequence is “eddie_rabbitt”; it is a pop music with human voice. The test result is shown in Table 4.13 and 4.14. The notation (a) is the original

floating point version, and (b) is the modified integer version. It seems acceptable in the data rate from 32 kbps to 96 kbps. Finally, the overall speed is 2.73 times faster than the original architecture. Note that the IMDCT part is 1/14 of the original in computation, and the result is shown in table 4.14.

	IMDCT	Total	Performance Ratio
Original	70848599	107506166	1
Optimized	5631239	39365730	2.7310

Table 4.12 Comparison of original and the optimized performance

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	160 kbps	196 kbps	256 kbps
(a)	-3.53	-3.37	-0.99	-0.38	-0.26	-0.05	-0.01	-0.01
(b)	-3.67	-3.36	-1.07	-0.52	-0.38	-0.36	-0.39	-0.44

Table 4.13 The ODG of test sequence “guitar”

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	160 kbps	196 kbps	256 kbps
(a)	-3.78	-3.40	-0.87	-0.27	-0.11	-0.17	-0.00	-0.00
(b)	-3.77	-3.33	-0.95	-0.41	-0.34	-0.30	-0.29	-0.30

Table 4.14 The ODG of test sequence “eddie_rabbitt”

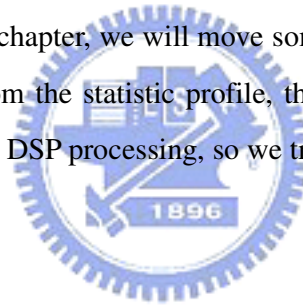
Chapter 5

MPEG-4 AAC Decoder

Implementation and

Optimization on DSP/FPGA

In the last chapter, we describe the implementation and optimization of the MPEG-AAC decoder on DSP. Also, in this chapter, we will move some of MPEG-4 AAC tools to FPGA to enhance the performance. From the statistic profile, the Huffman decoding and the IMDCT are the heaviest work tools for DSP processing, so we try to implementation these tools on the FPGA.



5.1 Huffman Decoding

In this section, we describe the implementation and optimization of the Huffman decoding on FPGA. We will implement two different architectures of Huffman decoder and compare the results.

5.1.1 Integration Consideration

In the MPEG-4 AAC decoder, the Huffman decoder receives a series of bits ranging from 1 bit to 19 bits from the input bitstream. It uses these bits to search for the matched pattern in the code index table. Then it returns a code index and length. The code index is ranging from

0 to 120, and we will take this value to find the codeword from the codeword table. Fig. 5.1 shows the flow diagram of the MPEG-4 AAC Huffman decoding process.

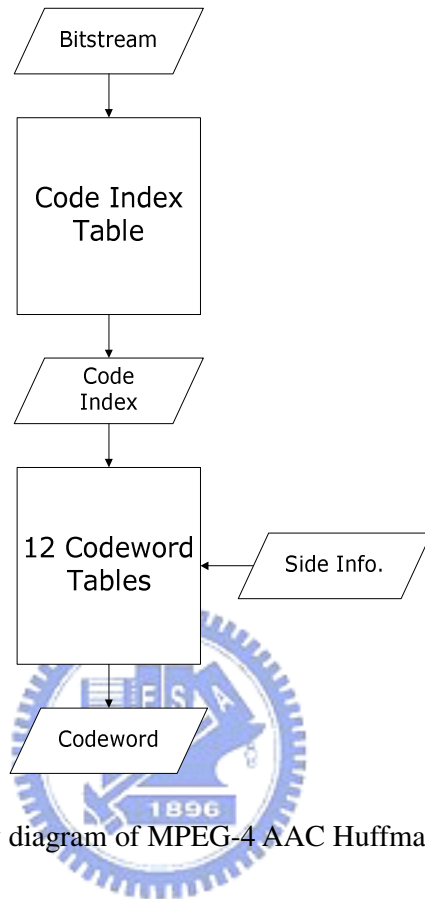


Fig. 5.1 Flow diagram of MPEG-4 AAC Huffman decoding

As we can see, the length of a symbol in the bitstream varies from 1 bit to 19 bits. The range of the code index in the table is 0 to 120, and its length is fixed to 7 bits. DSP is not suitable to do the variable length data processing, because it needs many extra clock cycles to find the correct length. Hence, we map out the MPEG-4 AAC Huffman decoder on DSP/FPGA. The patterns in the code index table are variable length, so we put it on FPGA; and the patterns in the codeword table are fixed length, so we put it on DSP. Fig. 5.2 shows the scheme of the DSP/FPGA integrated Huffman decoding.

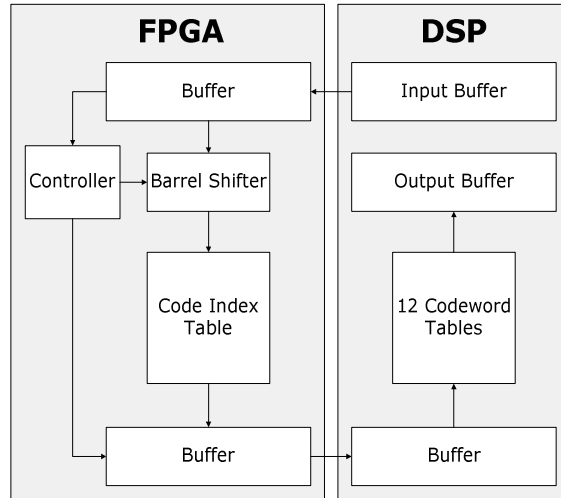


Fig. 5.2 Block diagram of DSP/FPGA integrated Huffman decoding

5.1.2 Fixed-output-rate Architecture

We put the code index table on FPGA. Also we want to implement the fixed-output-rate Huffman decoder architecture on FPGA. If we want to enhance the Huffman decoding performance substantially, we have to implement the parallel model on FPGA. This architecture outputs one code index in one clock cycle continuously.

We designed the code index table with the necessary control signals, Fig. 5.3 shows the block diagram. Because the code index range is from 0 to 120, we use 7-bit to represent the data. Allowing DSP fetch the code index easily, we put one bit “0” between two adjacent code indices in the output buffer. Fig 5.4 shows the output buffer diagram. In this way, the DSP can fetch the code index in “char” datatype easily.

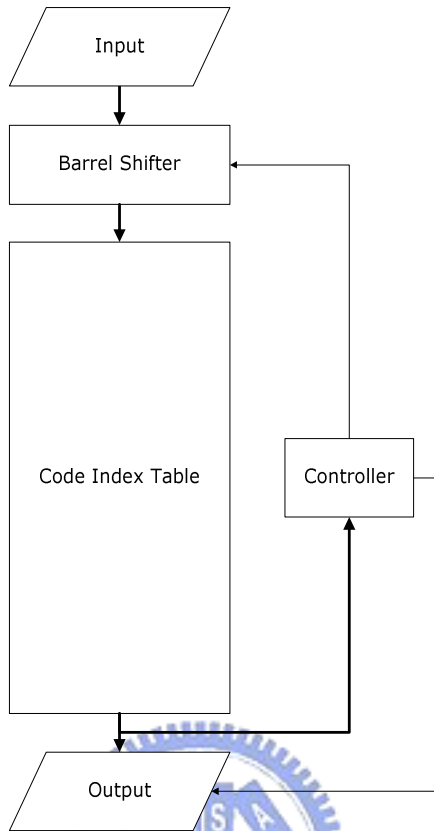


Fig. 5.3 Block diagram of fixed-output-rate architecture

0	Code Index 0	0	Code Index 1	0	Code Index 2	0	Code Index 3
---	--------------	---	--------------	---	--------------	---	--------------

Fig. 5.4 Output Buffer of code index table

The architecture needs some control signals between DSP and FPGA. When the DSP sends the “input_valid” signal to FPGA, it means the “input_data” is valid now. When the FPGA receives the “input_valid” signal and the FPGA is not busy, it would send a response of “input_res” signal to DSP, means the FPGA has received the input data successfully. But when the FPGA is busy, it would not send the “input_res” signal, meaning the FPGA has not received the input data successfully, and the DSP has to send the same data again. When the FPGA finishes the code index processing, it sends the “output_valid” signal to DSP, meaning the “output_data” is ready. Fig.5.5 shows the waveform of these signals, and each “output_data” contains ten code indexes. The architecture needs a clock cycle latency for the input register.

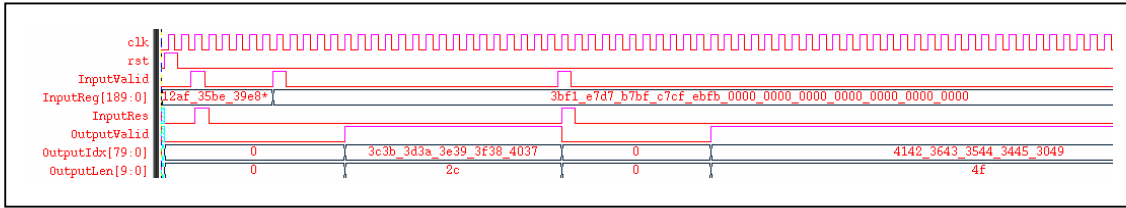


Fig 5.5 Waveform of the fixed-output-rate architecture

5.1.3 Fixed-output-rate Architecture

Implementation Result

Fig. 5.6 and Fig. 5.7 show the Xilinx ISE 6.1 synthesis and the P&R (place & route) reports. The P&R report shows that the clock cycle can reach 5.800 ns (172.4 MHz). It needs one clock cycle latency for the input register, meaning that we can retrieve about 156.7 M code indices in one second. We use a test sequence of 38 frames and it contains 13188 code indices. The comparison of DSP implementation and the FPGA implementation is shown in the Table 5.1.

Timing Summary:				
Speed Grade: -6				
Minimum period: 9.181ns (Maximum Frequency: 108.918MHz)				
Minimum input arrival time before clock: 4.812ns				
Maximum output required time after clock: 4.795ns				
Maximum combinational path delay: No path found				
Device utilization summary:				
Selected Device : 2v2000ff896-6				
Number of Slices:	820	out of	10752	7%
Number of Slice Flip Flops:	379	out of	21504	1%
Number of 4 input LUTs:	1558	out of	21504	7%
Number of bonded IOBs:	284	out of	624	45%
Number of GCLKs:	1	out of	16	6%

Fig 5.6 Synthesis report of the fixed-output-rate architecture

Timing Summary:				
Speed Grade: -6				
Clock to Setup on destination clock clk				
-----+-----+-----+-----+-----+				
	Src:Rise	Src:Fall	Src:Rise	Src:Fall
Source Clock	Dest:Rise	Dest:Rise	Dest:Fall	Dest:Fall
-----+-----+-----+-----+-----+				
clk	5.800			
-----+-----+-----+-----+-----+				
Device utilization summary:				
Number of External IOBs	285	out of	624	45%
Number of LOCed External IOBs	0	out of	285	0%
Number of SLICES	830	out of	10752	7%
Number of BUFGMUXs	1	out of	16	6%

Fig 5.7 P&R report of the fixed-output-rate architecture

	Time	Performance Ratio
DSP Implementation	4.7414×10^{-3}	1
FPGA Implementation	8.4161×10^{-5}	56.33

Table 5.1 The performance Comparison of DSP and FPGA implementation

5.1.4 Variable-output-rate Architecture

The fixed output rate Huffman decoder is limited by the speed of searching for the matched pattern [12]. We can further split the code index table into several small tables to reduce the comparison operations in one clock cycle. In this way, we can use shorten the time of processing short symbol, and it needs more than one clock cycle time to process the long symbols, which occurs is less frequently than the short symbol. But the cost is the more complex control signals. Fig. 5.8 shows the block diagram of the modified architecture.

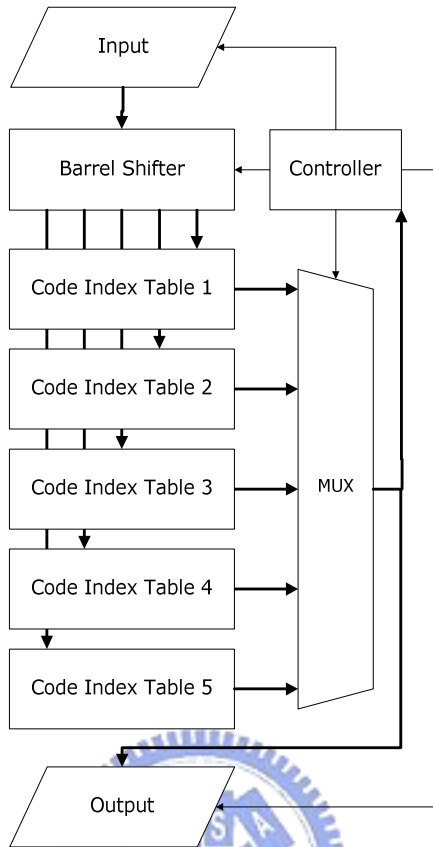


Fig. 5.8 Block diagram of the variable-output-rate architecture

Fig. 5.9 shows that the waveform and the external control signals between DSP/FPGA are the same for the fixed output rate architecture. The difference between the fixed-output-rate and the variable-output-rate architectures is the internal control signal of the variable-output-rate architecture is more complex, and the variable output rate architecture may need more clock cycle to produce the result.

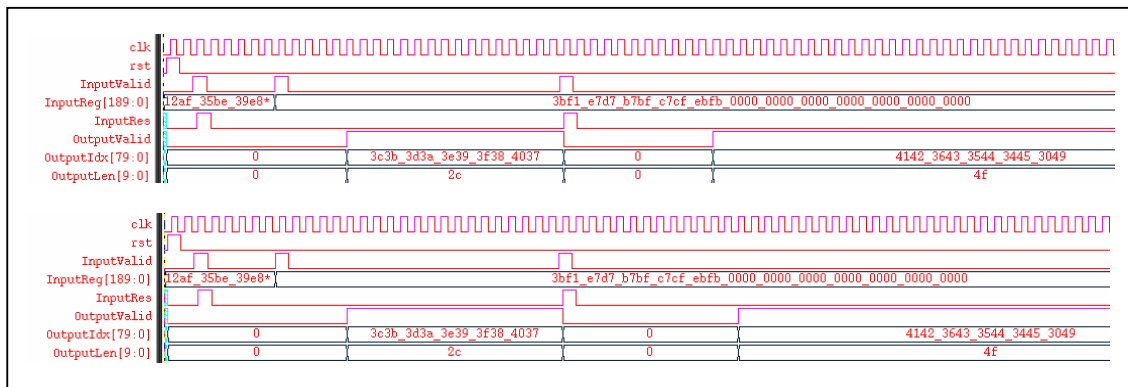


Fig 5.9 Comparison of the waveform of the two architectures

5.1.5 Variable-output-rate Architecture

Implementation Result

Fig. 5.10 and Fig. 5.11 show the synthesis report and the P&R report. Its clock rate is slower than that of the fixed-output-rate architecture. The implementation of the control signals may be constrained by the FPGA cell. When the control signals of FPGA design are too complex, the controller may become the FPGA system operating bottleneck.

Timing Summary:				
Speed Grade: -6				
Minimum period: 10.132ns (Maximum Frequency: 98.700MHz)				
Minimum input arrival time before clock: 4.829ns				
Maximum output required time after clock: 4.575ns				
Maximum combinational path delay: No path found				
Selected Device : 2v2000ff896-6				
Number of Slices:	945	out of	10752	8%
Number of Slice Flip Flops:	402	out of	21504	1%
Number of 4 input LUTs:	1785	out of	21504	8%
Number of bonded IOBs:	284	out of	624	45%
Number of GCLKs:	1	out of	16	6%

Fig 5.10 Synthesis report for the variable-output-rate architecture

Timing Summary:				
Speed Grade: -6				
Clock to Setup on destination clock clk				
-----+-----+-----+-----+-----+				
	Src:Rise	Src:Fall	Src:Rise	Src:Fall
Source Clock	Dest:Rise	Dest:Rise	Dest:Fall	Dest:Fall
-----+-----+-----+-----+-----+				
clk	6.131			
-----+-----+-----+-----+-----+				
Device utilization summary:				
Number of External IOBs	285	out of	624	45%
Number of LOCed External IOBs	0	out of	285	0%
Number of SLICES	989	out of	10752	9%
Number of BUFGMUXs	1	out of	16	6%

Fig 5.11 P&R report for the variable-output-rate architecture



5.2 IFFT

Continuing the discussion in chapter 4, we implement IFFT on FPGA to enhance performance of the IMDCT.

5.2.1 IFFT Architecture

We can compare several FFT hardware architectures [18], shown in Table 5.2. The SDF (single-path delay feedback) means to input one complex data in one clock cycle, then put the input data into a series of DFF (delay flip/flop) to wait for the appropriate time. Then we process the input data which are the source data from the same butterfly in the data flow diagram. The MDC (multi-path delay commutator) means to input two complex data which is the source of the same butterfly in the data flow diagram in one clock cycle. These two data can be processed in one cycle, but it needs more hardware resources. To summary, the SDF

architecture demands fewer registers and arithmetic function units, but the MDC architecture has less latency. We will use the radix-2³ SDF architecture of our IFFT.

	Number of Complex Multipliers	Number of Complex Adders	Number of Complex Registers
Radix-2 SDF	$\log_2 N - 2$	$2 \log_2 N$	$N - 1$
Radix-4 SDF	$1/2 \log_2 N - 1$	$4 \log_2 N$	$N - 1$
Radix-8 SDF	$1/3 \log_2 N - 1$	$(8 + 2t/3) \log_2 N$	$N - 1$
Radix-2 ² SDF	$1/2 \log_2 N - 1$	$2 \log_2 N$	$N - 1$
Radix-2 ³ SDF	$1/3 \log_2 N - 1$	$(2 + t/3) \log_2 N$	$N - 1$
Radix-2 MDC	$\log_2 N - 2$	$2 \log_2 N$	$1.5 N - 2$
Radix-4 MDC	$3/2 \log_2 N - 3$	$4 \log_2 N$	$2.5 N - 4$
Radix-8 MDC	$7/3 \log_2 N - 7$	$(8 + 2t/3) \log_2 N$	$4.5 N - 8$
Radix-2 ² MDC	$\log_2 N - 2$	$2 \log_2 N$	$1.5 N - 2$
Radix-2 ³ MDC	$2/3 \log_2 N - 2$	$(2 + t/3) \log_2 N$	$1.5 N - 2$

Table 5.2 Comparison of hardware requirements [18]

Because the data in PE is always multiplier a factor of $\sqrt{2}/2$, so we can use several shifters and adders to replace the multiplier. At first, we can see the binary representation of the

$$\sqrt{2}/2 = 0.7071 = 0.10110101,$$

If we set the “twiddle multiply factor” be 256, then the binary representation can be represented in fixed-point datatype by “10110101.” Then we can use five shifters and five adders to replace one multiplier as Fig 5.12 shows the block diagram. In the Table 5.2, the “t” represent that the “1” of the simplified multiplier used.

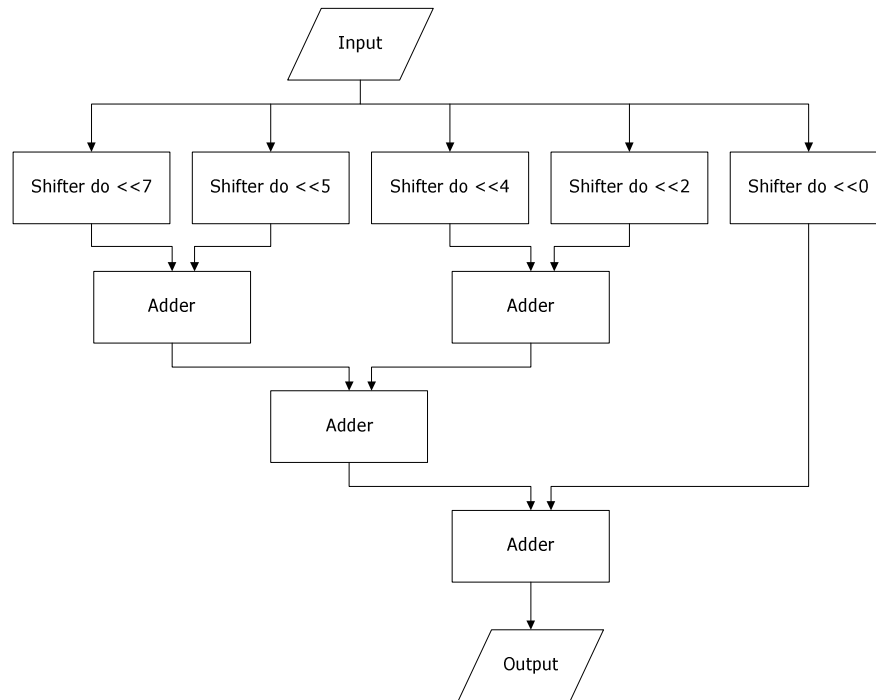


Fig. 5.12 Block diagram of shifter-adder multiplier

5.2.2 Quantization Noise Analysis

First, we want to analyze the quantization noise due to transforming the datatype from floating-point to fixed-point. The original range of the twiddle factor is from -1 to 1 , so we need to scalar up the “twiddle multiplier” for integer representation. Also, we need to scalar up the input data to the “scaling multiplier.” At the end, we generate 1000 sets of random input data in the range from -5000 to 5000 , and compute the output SNR for the IFFT. If an overflow occurs, the SNR would drop down drastically. Therefore, we do not label the SNR for the overflow codes.

There are two main differences between the FFT and the IFFT. The first one is the twiddle factor is conjugate, and the second is the IFFT has to multiply a $1/N$ factor but the FFT does not. If we multiply the $1/N$ factor at the last stage, the SNR would be better, but the effective bit in the output data would be less. So we split the $1/N$ factor into multiple stage, and each stage is only a multiplication of a factor of $1/2$. Fig. 5.13 and 5.14 show the comparison of the noise analysis. As the result, we choose “twiddle multiplier” to be 256, and the “scaling multiplier” to be 1.

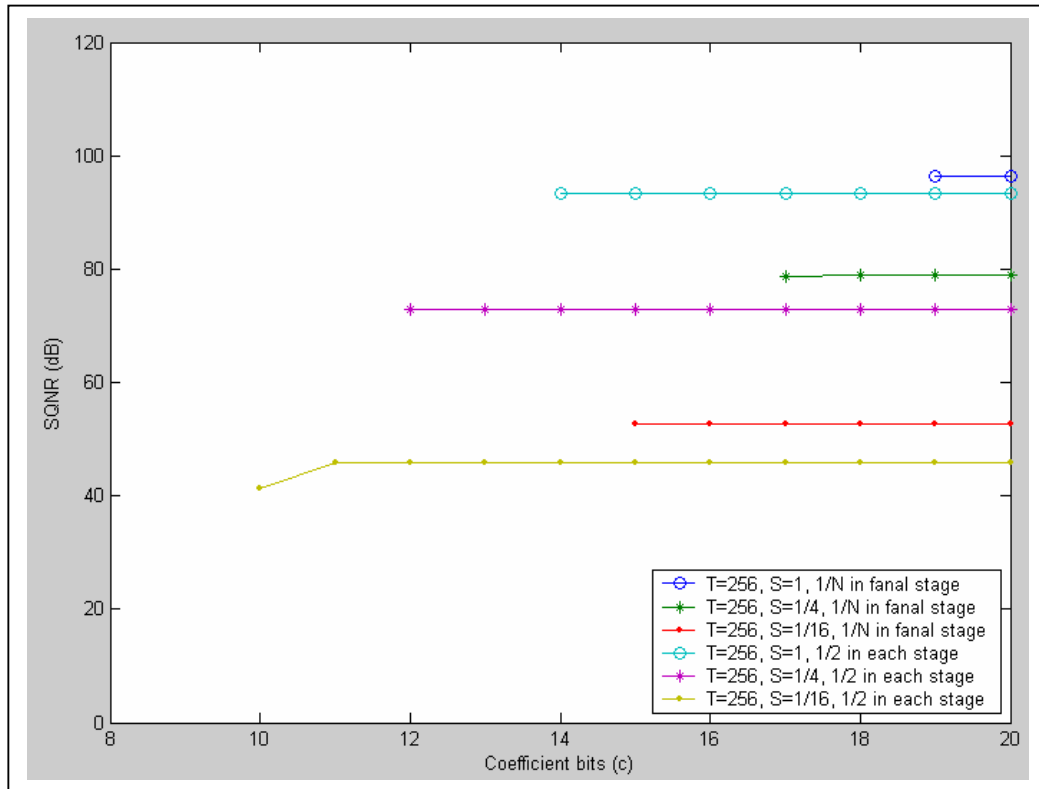


Fig. 5.13 Quantization noise analysis for twiddle multiplier with scaling of 256

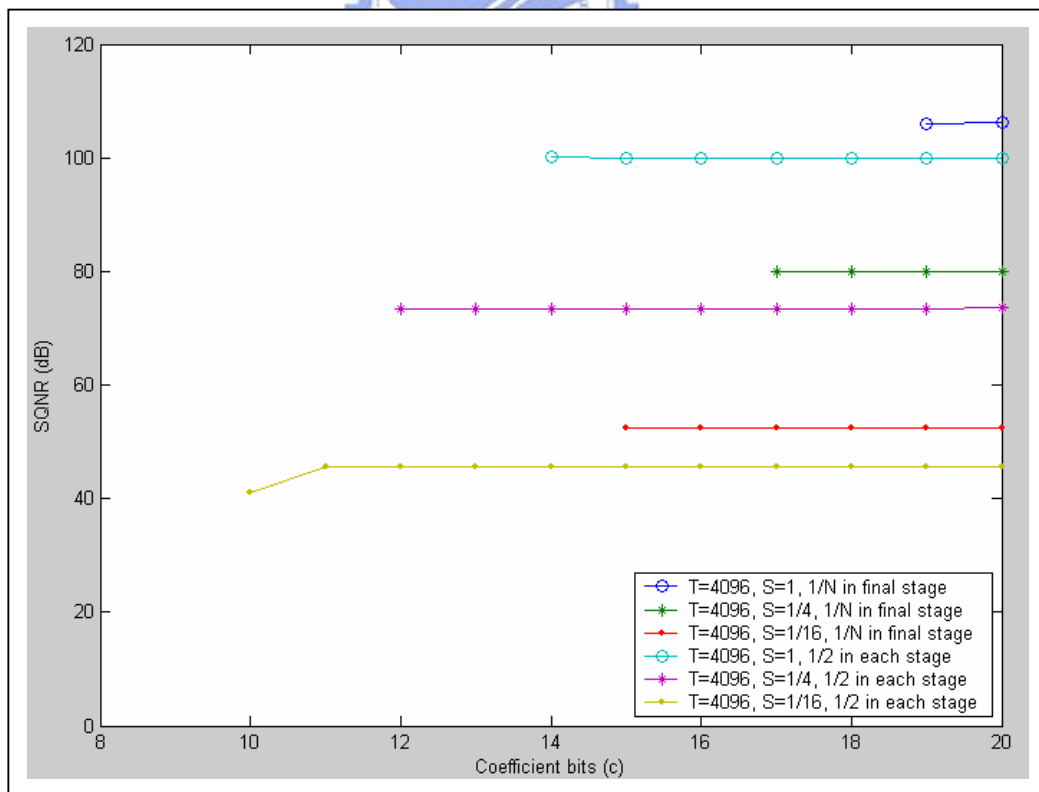


Fig. 5.14 Quantization noise analysis for twiddle multiplier with scaling of 4096

5.2.3 Radix-2³ SDF IFFT Architecture

We use the radix-2³ SDF 512-point IFFT pipelined architecture as Fig. 5.15 shows. The input data from the first one to the last one are put into the IFFT sequentially. Fig. 5.16 shows the computational work for each PE.

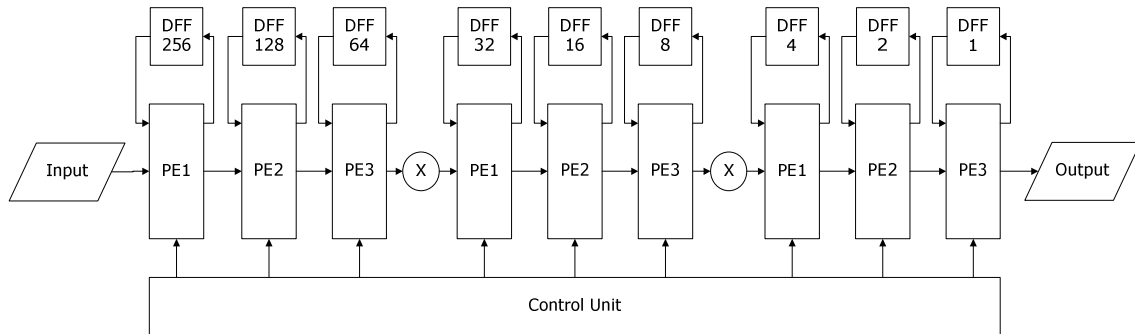


Fig. 5.15 Block diagram of radix2³ SDF 512-point IFFT pipelined architecture

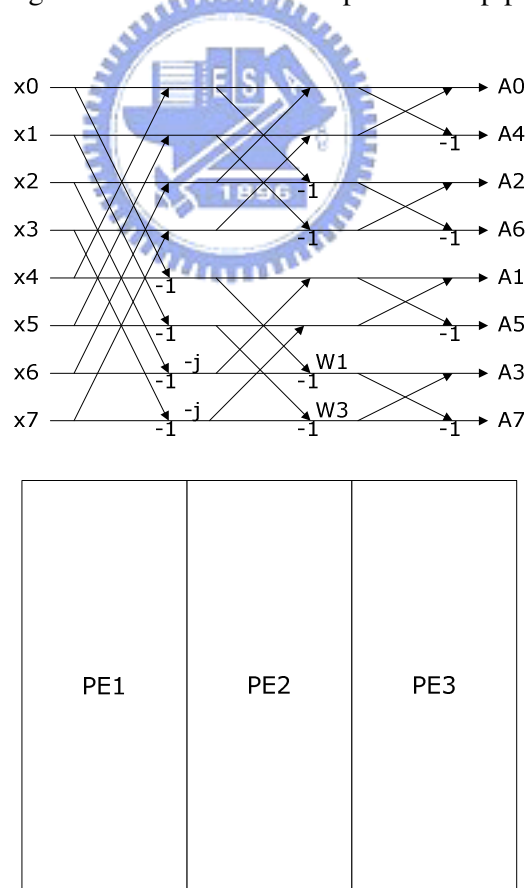


Fig. 5.16 Simplified data flow graph for each PE

PE1 has the architecture as Fig 5.17 shows. At the first $N/4$ clock cycle, PE1 puts the DFF output data to the PE1 output and put the input data to the DFF input. The next $N/4$ clock cycle, PE1 multiply the DFF output data by j then put to the PE1 output and put the input data to the DFF input. We can replace the multiplication by exchange the real part and the imaginary part of data. At the last $N/2$ clock cycle, PE1 add the DFF output data to the input data to the PE1 output, and subtract the DFF data from the input data to the DFF input.

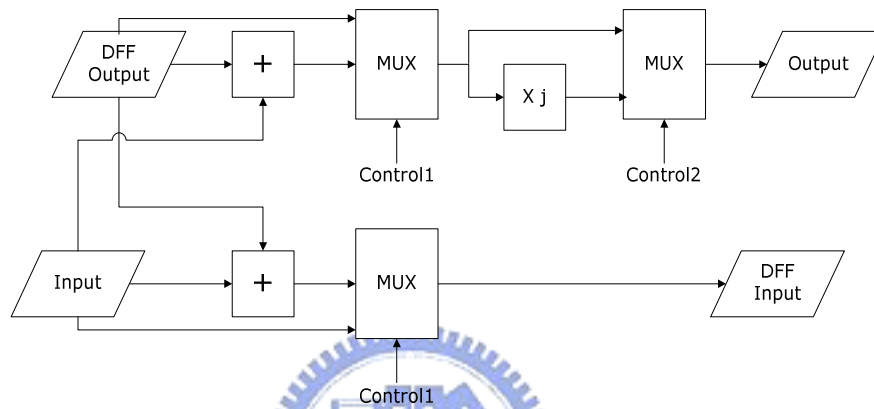


Fig 5.17 Block diagram of the PE1

PE2 has the architecture as Fig 5.18 shows. At the first $N/8$ clock cycle, the PE2 put the DFF output data to the PE2 output and put the input data to the DFF input. The next $N/8$ clock cycle, PE2 multiply DFF output data by j then put to the PE2 output and put the input data to the DFF input. We can replace the multiplication by exchange the real part and the imaginary part of data. At the third $N/8$ clock cycle, PE2 add the DFF output data to the input data to the PE2 output, and subtract the DFF output data from the input data to the DFF input. At the fourth $N/8$ clock cycle, PE2 add the DFF output data and the input data, then multiply $\frac{\sqrt{2}}{2}(1+j)$ to the PE2 output, and subtract the DFF output data from the input data to the DFF input. At the fifth $N/8$ clock cycle, PE2 put the DFF output data to the output and put the input data to the DFF input. At the sixth $N/8$ clock cycle, PE2 multiply the DFF output data by $-\frac{\sqrt{2}}{2}(1-j)$ to the output, and put input data to the DFF input. At the last $N/4$ clock cycle, PE2 add the DFF output data to the input data to the PE2 output, and subtract the DFF output data from the input data to the DFF input.

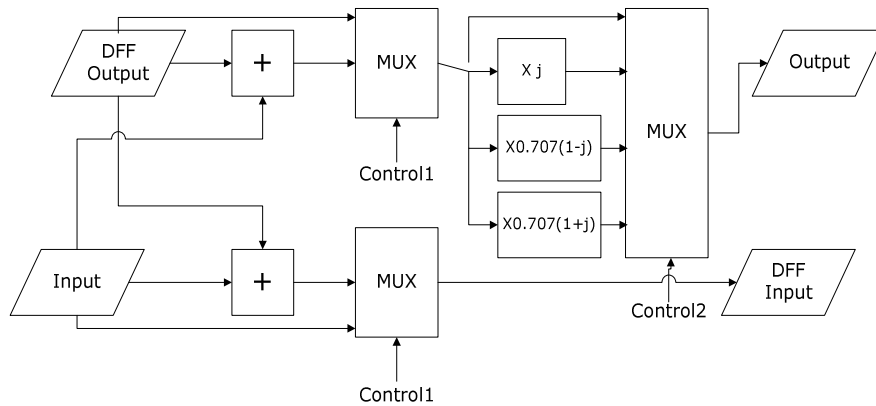


Fig 5.18 Block diagram of the PE2

PE3 has the architecture as Fig 5.19 shows. At the first $N/8$ clock cycle, the PE3 put the DFF output data to the PE3 output and put the input data to the DFF input. At the next $N/8$ clock cycle, add the DFF output data to the input data to the PE3 output, and subtract the DFF output data from the input data to the DFF input.

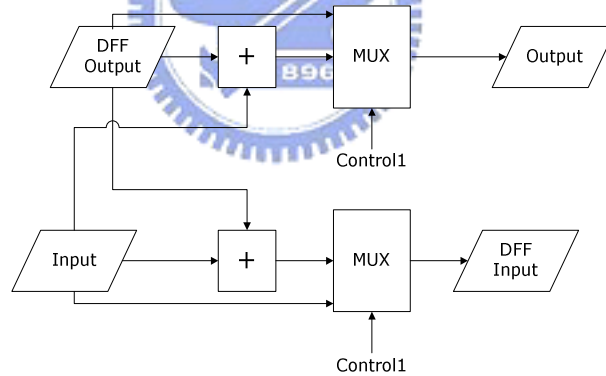


Fig 5.19 Block diagram of the PE3

In the beginning, we use a big MUX and control signals to select the twiddle factor. In the Huffman decoding section in this chapter, we found that the complex control signal would slow down the clock. In the IFFT, the complex control signals might not be synthesized in the FPGA. So we try a simple way to implement the twiddle multiplier which does not to use the complex control signals. We put the twiddle factor in a circular shift register in the order and then access the first one at each clock cycle. Fig. 5.20 shows the circular shift register of twiddle factor multiplier. In this way, we can avoid to use complex control signals.

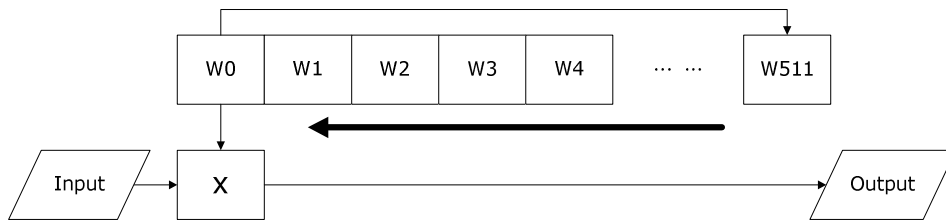


Fig 5.20 Block diagram of the twiddle factor multiplier

The Fig. 5.21 shows the signal waveform of the IFFT. When the DSP sends a “input_valid” signal to FPGA, it means the input data will start to transfer sequentially. The FPGA sends the “output_valid” signal to DSP meaning the output data will start to transfer in sequentially.

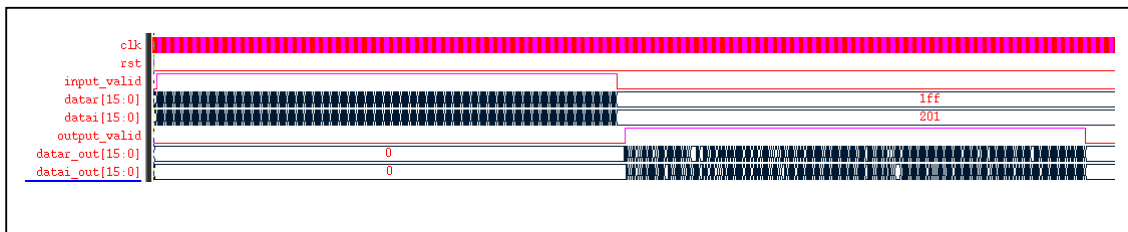


Fig 5.21 Waveform of the radix-2³ 512-point IFFT

5.2.4 IFFT Implementation Result

The Fig. 5.22 and 5.23 show the synthesis report and the P&R report of the IFFT. The clock frequency on P&R can reach 93.14 MHz. It means it can process 95.9k long window data in one second. We use a test sequence with 12 long window data. The comparison of DSP implementation and FPGA implementation is shown in Table 5.3.

Timing Summary:				
Speed Grade: -6				
Minimum period: 11.941ns (Maximum Frequency: 83.745MHz)				
Minimum input arrival time before clock: 2.099ns				
Maximum output required time after clock: 4.994ns				
Maximum combinational path delay: No path found				
Selected Device : 2v6000ff1152-6				
Number of Slices:	17045	out of	33792	50%
Number of Slice Flip Flops:	28295	out of	67584	41%
Number of 4 input LUTs:	2503	out of	67584	3%
Number of bonded IOBs:	67	out of	824	8%
Number of MULT18X18s:	54	out of	144	37%
Number of GCLKs:	1	out of	16	6%

Fig 5.22 Synthesis report of radix-2³ 512-point IFFT



Timing Summary:				
Speed Grade: -6				
Clock to Setup on destination clock clk				
-----+-----+-----+-----+-----+				
	Src:Rise	Src:Fall	Src:Rise	Src:Fall
Source Clock	Dest:Rise	Dest:Rise	Dest:Fall	Dest:Fall
-----+-----+-----+-----+-----+				
clk	10.736			
-----+-----+-----+-----+-----+				
Design Summary				
Logic Utilization:				
Number of Slice Flip Flops:	28,267	out of	67,584	41%
Number of 4 input LUTs:	2,420	out of	67,584	3%
Logic Distribution:				
Number of occupied Slices:	15,231	out of	33,792	45%
Number of Slices containing only related logic:	15,231	out of	15,231	100%
Number of Slices containing unrelated logic:	0	out of	15,231	0%
Total Number 4 input LUTs:	2,568	out of	67,584	3%
Number used as logic:	2,420			
Number used as a route-thru:	148			
Number of bonded IOBs:	68	out of	824	8%
IOB Flip Flops:	28			
Number of MULT18X18s:	54	out of	144	37%
Number of GCLKs:	1	out of	16	6%
Total equivalent gate count for design: 464,785				

Fig 5.23 P&R report of radix-2³ 512-point IFFT

	Time	Performance Ratio
DSP Implementation	4.0486 x 10 ⁻⁵	1
FPGA Implementation	1.0428 x 10 ⁻⁵	3.8825

Table 5.3 The performance comparison of DSP and FPGA implementation

5.3 Implementation on DSP/FPGA

We have implemented and optimized MPEG-4 AAC on TI C64x DSP and Xilinx Virtex-II FPGA. The optimized result has been shown in Table 5.4. We use a 0.95 second test sequence to compare the performance of the DSP implementation and the DSP/FPGA implementation. The overall speed is 8.17 times faster than the original version, and the DSP/FPGA version can process 48-second audio data of in 1 second.

	Huffman Decoding	IMDCT	Total	Performance Ratio
Original	0.03738622	0.11808099	0.17917694	1
DSP Modified	0.03738622	0.00938539	0.06560996	2.73
DSP/FPGA	0.00066370	0.00241736	0.02192054	8.17

Table 5.4 Comparison of DSP and DSP/FPGA implementation





Chapter 6

Conclusion and Future Work

We have implemented the MPEG-4 AAC decoder on DSP and FPGA together. In this project, we speed up the IMDCT implementation on DSP implementation, and the modified version is 503 times faster than the original version. And then we implement the Huffman decoding and IFFT on FPGA. The implementation and optimized results are faster than the DSP version as expected.

For the IMDCT calculation, we use radix-2³ FFT algorithm in DSP implementation. Then, we use fixed-point data type to present the input data. In addition, we rearrange the data calculation order in IFFT. Furthermore, we use intrinsic functions to speed up the IFFT. The test result is 503 times faster than the original version. The details of our design and results can be found in chapter 4.

We use FPGA to implement the fixed-output-rate Huffman decoder. Also, we modify this architecture to a more efficient variable-output-rate architecture. But the latter is in fact slower than the former due to the complexity of the control signals, which create slow paths on FPGA. The FPGA implementation is about 56 times faster than the DSP implementation.

We also use FPGA to implement IFFT. Similar to the DSP implementation, we use radix-2³ FFT algorithm for IFFT. The 512-point IFFT has a heavy computational load. Therefore, we use three types of PE to perform these computations in order to reduce the chip area. The FPGA implementation of IFFT is about 4 times faster than the fastest DSP version. The details of our design and results can be found in chapter 5.

Due to the board hardware defect and/or system software bug, we are unable to run and test our implementations on the DSP/FPGA baseboard yet. Thus, there are two important targets in the future. First, the DSP implementation should be executed on the DSP baseboard, and the streaming interface is needed to connect to the Host PC in real time execution. The Host PC reads in the source data from the file in the memory, and then it transfers the data to

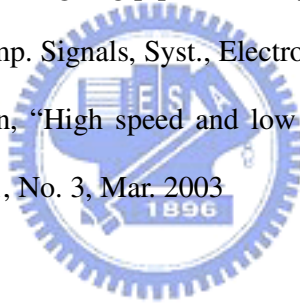
DSP through the streaming interface. After DSP has processed data, it transfers data back to the Host PC. The second target is to integrate the FPGA implementation together with DSP to demonstrate the overall system. DSP does the pre-processing and then it transfers the data to FPGA through the streaming interface. After FPGA has processed the data, it transfers data back to DSP.



Bibliography

- [1] ISO/IEC JTC/SC29/WG11 MPEG, International Standard ISO/IEC 13818-7 “Advanced Audio Coding”, 1997
- [2] ISO/IEC JTC/SC29/WG11 MPEG, International Standard ISO/IEC 14496-3 “Advanced Audio Coding”, 1999
- [3] M. Bosi and et al., “ISO/IEC MPEG-2 Advanced Audio Coding”, JAES, Vol.45, No.10 Oct. 1997
- [4] M. Wolters and et al., “A closer look into MPEG-4 High Efficiency AAC”, AES 115th Convention Paper, 2003
- [5] Innovative Integration, “Quixote User’s Manual”, Dec. 2003
- [6] Texas Instruments, “TMS320C6000 Programmer’s Guide”, SPRU198F, Feb. 2001
- [7] Texas Instruments, “TMS320C6000 CPU and Instruction Set Reference Guide”, SPRU189F, Jan. 2000
- [8] Texas Instruments, “TMS320C6000 Peripherals Reference Guide”, SPRU190D, Mar. 2001
- [9] Texas Instruments, “TMS320C64x Technical Overview”, SPRU395B, Jan. 2001
- [10] Xilinx, “Virtex-II Platform FPGA User Guide”, UG002(v1.7) Feb. 2004
- [11] K. S. Lee and et al., “A VLSI implementation of MPEG-2 AAC decoder system,” ASICs, 1999 AP-ASIC '99. The First IEEE Asia Pacific Conf., pp. 139-142, 23-25 Aug. 1999
- [12] M. K. Rudberg and L. Wanhammer, “New approaches to high speed Huffman decoding”, IEEE Int. Symp., Vol. 2, pp. 149-152, 12-15 May 1996

- [13] M. K. Rudberg and L. Wanhammar, "High speed pipelined parallel Huffman decoding," IEEE Proc. Int. Symp., Vol. 3, pp.2080-2083, 9-12 Jun. 1997
- [14] P. Duhamel and et al., "A fast algorithm for the implementation of filter banks based on 'time domain aliasing cancellation'", IEEE Trans. Acous., Speech, Signal Processing, ICASSP, Vol. 3, pp. 2209-2212, Apr. 1991
- [15] P. Duhamel and H. Hollmann, "Split-radix FFT algorithm for complex, real, and real symmetric data," IEEE Trans. Acous., Speech, Signal Processing, ICASSP, Vol. 10, pp. 784-787, Apr. 1985
- [16] S. He and M. Torkelson, "A new approach to pipeline FFT processor", IEEE Proc. 10th Int. Parallel Processing Symp., IPPS, Apr. 1996
- [17] S. He and M. Torkelson, "Designing pipeline FFT processor for OFDM (de)modulation", IEEE Proc. URSI Int. Symp. Signals, Syst., Electron., pp. 257-262, Oct. 1998
- [18] W. C. Yeh and C. W. Jen, "High speed and low power split-radix FFT," IEEE Trans. Signal Processing, Vol. 51, No. 3, Mar. 2003

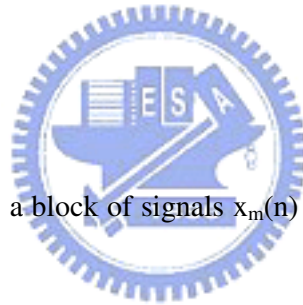


Appendix A

N/4-point FFT Algorithm for MDCT

We will describe the N/4-point complex FFT in detail in this appendix. We will show the mathematical derivation to the algorithm. The details can be found in [14].

A.1 MDCT



The MDCT can be seen as a block of signals $x_m(n)$ project on a set of cosine functions as follow

$$Y_m(k) = \sum_{n=0}^{N-1} x_m(n)h(N-1-n)\cos((2\pi(2k+1)/2)(n+n_0)/N), \quad (\text{A.1})$$

where $h(n)$ is a weighting function, N is the block size, and n_0 is a phase shift. It can be seen that this transform is not invertible, since

$$Y_m(k-1) = -Y_m(N-k), \quad (\text{A.2})$$

only $N/2$ output points are linearly independent.

However, if two adjacent block $x_m(n)$ and $x_{m+1}(n)$ overlap by $N/2$, the set of values $x_m(n)$ can be removed from two successive output sets $Y_{m-1}(n)$ and $Y_m(n)$. Let

$$X_m(n) = \sum_{k=0}^{N-1} Y_m(k)\cos((2\pi(2k+1)/2)(n+n_0)/N) \text{ for blocks } m-1 \text{ and } m. \quad (\text{A.3})$$

Then, $x_m(n)$ can be shown to be equal to

$$x_m(n) = g(n+N/2)X_{m-1}(n+N/2) + g(n)X_m(n) \quad (\text{A.4})$$

this reconstruction is perfect when the windows are symmetric and identical, thus $g(n)=h(n)$.

A.2 N/4-Point FFT

The antisymmetry of the FFT output coefficients allows that we only compute half the input signals. In order to obtain a formula which is easy to handle, we have chosen to keep the even coefficients. The odd ones are reduced by Eq. (A.2). Hence Eq. (A.1) is equivalent to

$$Y_{2k} = \sum_{n=0}^{N-1} y_n \cos(2\pi(2n+1)(4k+1)/4N + (4k+1)\pi/4), \quad (\text{A.5})$$

which can be rewritten as

$$Y_{2k} = (-1)^k \sqrt{2}/2 \sum_{n=0}^{N-1} y_n ((\cos(2\pi(2n+1)(4k+1)/4N) - \sin(2\pi(2n+1)(4k+1)/4N)) \quad (\text{A.6})$$

A symmetrical function in n and k can be obtained by performing the following permutation, which is typical in the DCT case

$$y'_n = y_{2n}, \quad n = 0, \dots, N/2 - 1 \quad (\text{A.7})$$

$$y''_n = y_{N-2n-1}, \quad n = 0, \dots, N/2 - 1 \quad (\text{A.8})$$

Here we will use two symbols:

$$c = \cos(2\pi(4n+1)(4k+1)/4N) \quad (\text{A.9})$$

$$s = \sin(2\pi(4n+1)(4k+1)/4N) \quad (\text{A.10})$$

It can be shown that

$$Y_{2k} = (-1)^k \sqrt{2}/2 \sum_{n=0}^{N/2-1} y'_n (c-s) + y''_n (-c-s) \quad (\text{A.11})$$

$$Y_{2(k+N/4)} = (-1)^k \sqrt{2}/2 \sum_{n=0}^{N/2-1} y'_n (-c-s) + y''_n (-c+s) \quad (\text{A.12})$$

If we define W_{4N} as the $4N$ th root of unity,

$$W_{4N} = \cos(2\pi/4N) + j \sin(2\pi/4N) \quad (\text{A.13})$$

Then Eq (A.11) and Eq. (A.12) can be grouped into a complex formula:

$$Y_{2k+N/4} + jY_{2k} = (-1)^{k+1} W_8^{-1} \sum_{n=0}^{N/4-1} ((y'_n - y''_{n+N/4}) + j((y''_n + y'_{n+N/4}))) W_{4N}^{(4k+1)(4n+1)} \quad k = 0, \dots, N/4 - 1 \quad (\text{A.14})$$

which is in the form of a modified complex DFT.

Appendix B

Radix-2² and Radix-2³ FFT

We will describe the radix-2² and radix-2³ FFT in detail in this appendix. We will discuss the mathematical derivation to the algorithm. The details can be found in [16] and [17].

B.1 Radix-2² FFT

At first, we will see the analytical expression for the FFT is

$$X_k = \sum_{n=0}^{N-1} x_n \cdot W_N^{kn}, \quad k = 0, 1, \dots, N-1, \quad \text{Eq. B.1}$$

and the analytical expression for the IFFT is

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot W_N^{-kn}, \quad n = 0, 1, \dots, N-1, \quad \text{Eq. B.2}$$

The derivation of the radix-2² FFT algorithm starts with a substitution with a 3-dimensional index map. The index n and k in Eq. B.1 can be expressed as

$$n = \left(\frac{N}{2} n_1 + \frac{N}{4} n_2 + n_3 \right)_N \quad \text{Eq. B.3}$$

$$k = (k_1 + 2k_2 + 4k_3)_N \quad \text{Eq. B.4}$$

When the above substitutions are applied to DFT definition, the definition can be rewritten as

$$\begin{aligned} X(k_1 + 2k_2 + 4k_3) &= \sum_{n_3=0}^{\frac{N}{4}-1} \sum_{n_2=0}^1 \sum_{n_1=0}^1 x\left(\frac{N}{2} n_1 + \frac{N}{4} n_2 + n_3\right) \cdot W_N^{\left(\frac{N}{2} n_1 + \frac{N}{4} n_2 + n_3\right)(k_1 + 2k_2 + 4k_3)} \\ &= \sum_{n_3=0}^{\frac{N}{4}-1} \sum_{n_2=0}^1 \left\{ B_N^{k_1} \left(\frac{N}{4} n_2 + n_3 \right) \cdot W_N^{\left(\frac{N}{4} n_2 + n_3\right) k_1} \right\} \cdot W_N^{\left(\frac{N}{4} n_2 + n_3\right)(2k_2 + 4k_3)} \quad \text{Eq. B.5} \end{aligned}$$

where

$$B_{\frac{N}{2}}^{k_1} \left(\frac{N}{4} n_2 + n_3 \right) = x \left(\frac{N}{4} n_2 + n_3 \right) + (-1)^{k_1} \cdot x \left(\frac{N}{4} n_2 + n_3 + \frac{N}{2} \right) \quad \text{Eq. B.6}$$

which is a general radix-2 butterfly

Now, the two twiddle factor in Eq. B.6 can be rewritten as

$$\begin{aligned} W_N^{\left(\frac{N}{4} n_2 + n_3\right)(k_1 + 2k_2 + 4k_3)} &= W_N^{N n_2 k_3} W_N^{\frac{N}{4} n_2 (k_1 + 2k_2)} W_N^{n_3 (k_1 + 2k_2)} W_N^{4 n_3 k_3} \\ &= (-j)^{n_2 (k_1 + 2k_2)} W_N^{n_3 (k_1 + 2k_2)} W_N^{4 n_3 k_3} \end{aligned} \quad \text{Eq. B.7}$$

Observe that the last twiddle factor in the above Eq. B.5 can be rewritten.

$$W_N^{4 n_3 k_3} = e^{\frac{-j2\pi}{N} 4 n_3 k_3} = e^{\frac{-j2\pi}{4N} n_3 k_3} = W_{\frac{N}{4}}^{n_3 k_3} \quad \text{Eq. B.8}$$

Insert Eq. B.8 and Eq. B.7 in Eq. B.5, and expand the summation over n_2 . The result is a DFT definition with four times shorter.

$$X(k_1 + 2k_2 + 4k_3) = \sum_{n_3=0}^{\frac{N}{4}-1} [H(n_3, k_1, k_2) W_N^{n_3 (k_1 + 2k_2)}] W_{\frac{N}{4}}^{n_3 k_3} \quad \text{Eq. B.9}$$

The result is that the butterflies have the following structure. The PE2 butterfly takes the input from two PE1 butterflies.

$$H(n_3, k_1, k_2) = \left[x(n_3) + (-1)^{k_1} x\left(n_3 + \frac{N}{2}\right) \right] + (-j)^{(k_1 + 2k_2)} \left[x\left(n_3 + \frac{N}{4}\right) + (-1)^{k_1} x\left(n_3 + \frac{3N}{4}\right) \right] \quad \text{Eq. B.10}$$

These calculations are for first radix-2² butterfly, or components the PE1 and PE2 butterflies. The PE1 is the one represented by the formulas in brackets in Eq. B.10 and PE2 is the outer computation in the same equation. The complete radix-2² algorithm is derived by applying this procedure recursively.

B.2 Radix-2³ FFT

Like radix-2² FFT algorithm, the derivation of the radix-2³ FFT algorithm starts with a substitution with a 4-dimensional index map. The index n and k in Eq. B.1 can be expressed as

$$n = \left(\frac{N}{2}n_1 + \frac{N}{4}n_2 + \frac{N}{8}n_3 + n_4\right)_N \quad \text{Eq. B.11}$$

$$k = (k_1 + 2k_2 + 4k_3 + 8k_4)_N \quad \text{Eq. B.12}$$

When the above substitutions are applied to DFT definition, the definition can be rewritten as

$$X(k_1 + 2k_2 + 4k_3 + 8k_4) = \sum_{n_4=0}^{\frac{N}{8}-1} \sum_{n_3=0}^1 \sum_{n_2=0}^1 \sum_{n_1=0}^1 x\left(\frac{N}{2}n_1 + \frac{N}{4}n_2 + \frac{N}{8}n_3 + n_4\right) \cdot W_N^{nk} \quad \text{Eq. B.13}$$

is a general radix-2 butterfly

Now, the two twiddle factor in Eq. B.13 can be rewritten as

$$\begin{aligned} W_N^{nk} &= W_N^{\left(\frac{N}{2}n_1 + \frac{N}{4}n_2 + \frac{N}{8}n_3 + n_4\right) \cdot (k_1 + 2k_2 + 4k_3 + 8k_4)} \\ &= W_N^{\frac{N}{2}n_1 k_1} W_N^{\frac{N}{4}(k_1 + 2k_2)n_2} W_N^{\frac{N}{8}n_3(k_1 + 2k_2 + 4k_3)} W_N^{n_4(k_1 + 2k_2 + 4k_3 + 8k_4)} \\ &= (-1)^{n_1 k_1} (-j)^{n_2(k_1 + 2k_2)} W_N^{\frac{N}{8}n_3(k_1 + 2k_2 + 4k_3)} W_N^{n_4(k_1 + 2k_2 + 4k_3)} W_N^{8n_4 k_4} \end{aligned} \quad \text{Eq. B.14}$$

Substitute Eq. B.14 into Eq. B.13, and expand the summation with regard to index n_1 , n_2 and n_3 . After simplification we have a set of 8 DFT of length $N/8$.

$$X(k_1 + 2k_2 + 4k_3 + 8k_4) = \sum_{n_4=0}^{\frac{N}{8}-1} [T(n_4, k_1, k_2, k_3) W_N^{n_4(k_1 + 2k_2 + 4k_3)}] W_N^{\frac{n_4 k_4}{8}} \quad \text{Eq. B.15}$$

There a third butterfly structure has the expression of

$$T(n_4, k_1, k_2, k_3) = H_{\frac{N}{8}}(n_4, k_1, k_2) + W_N^{\frac{N}{8}(k_1 + 2k_2 + 4k_3)} H_{\frac{N}{4}}\left(n_4 + \frac{N}{8}, k_1, k_2\right) \quad \text{Eq. B.16}$$

As in the Radix- 2^2 FFT algorithm, Eq. B.6 and Eq. B.10 represent the first two columns of butterflies with only trivial multiplications in the Radix-23 FFT algorithm. The third butterfly contains a special twiddle factor

$$W_N^{\frac{N}{8}(k_1 + 2k_2 + 4k_3)} = \left(\frac{\sqrt{2}}{2}(1-j)\right)^{k_1} (-j)^{(k_2 + 2k_3)} \quad \text{Eq. B.17}$$



作者簡歷

曾建統，民國六十七年出生於新竹市。民國九十一年六月畢業於國立交通大學電子工程學系，同年九月進入國立交通大學電子所就讀，從事多媒體訊號處理系統設計與實現之相關研究。民國九十三年六月取得碩士學位，碩士論文題目為『MPEG-4 先進音訊編碼在 DSP/FPGA 平台上的實現與最佳化』。研究範圍與興趣包括：多媒體訊號處理，軟硬體整合實現與最佳化。

