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 先進音訊編解碼器之增速及其在 DSP 平台上的實現

MPEG-4 AAC Codec Acceleration and DSP Implementation

研 究 生：王盈閔

指導教授：杭學鳴 博士

中 華 民 國 九 十 四 年 六 月

MPEG-4 先進音訊編解碼器之增速及其在 DSP 平台上的實現

MPEG-4 AAC Codec Acceleration and DSP Implementation

研 究 生：王盈閔　　Student: Yin-Ming Wang

指導教授：杭學鳴 博士　　Advisor: Dr. Hsueh-Ming Hang

國 立 交 通 大 學

電子工程學系　電子研究所碩士班

碩 士 論 文

A Thesis

Submitted to Institute of Electronics

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of Requirements

for the Degree of

Master of Science

in

Electronics Engineering

June 2005

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 四 年 六 月

MPEG-4 先進音訊編解碼器之增速及其在 DSP 平台上的實現

學生：王盈閔　　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)是非常有效率的音訊壓縮編碼技術。它是由 ISO/IEC MPEG 所制定的一套標準。

在本篇論文當中，我們首先分析 MPEG-4 先進音訊編碼器在 DSP 上的執行計算複雜度。發現心理聲學模式(psychoacoustic model)和量化及位元編碼(bit allocation)所花的執行時脈週期為最多，因此針對它們，我們在 DSP 上的實現利用比較快速的演算法主要加速之。

在 DSP 實現方面，為了加速先進音訊編碼器，我們針對 DSP 的架構使用了一些程式技巧，包括定點式資料型態、TI DSP 的特殊指定群等等。除此之外，我們也參考了一些快速運行的演算法，並套用在原來的音訊編碼器之心理聲學模式及量化位元編碼上。經由這些的程式修改，最後的編碼器版本在 DSP 上的執行速度比原來的有了 77.89%的改善幅度。並且我們也成功的把先進音訊編碼器及解碼器兩者實現在 II(Innovative Integration) 所提供的 Quixote DSP 平台上。而在主端及客端的傳輸介面，我們採用了緩衝之區塊傳輸模式，此模式讓我們容易實現整個架構。最後經由我們的加速及系統實現，此先進音訊編碼及解碼器各自都可達到即時編解碼的效果。

MPEG-4 AAC Codec Acceleration and DSP Implementation

Student: Yin-Ming Wang 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 ISO/IEC (International Standard Organization) MPEG (Moving Pictures Experts Groups) committee.

In this thesis, we first analyze the computational complexity of the MPEG-4 AAC encoder program. We find that the PAM (psychoacoustic model) and the quantization and bit allocation module require the most execution cycles on DSP. Hence, we mainly propose methods to accelerate them on DSP.

In order to speed up the AAC encoder on DSP, we use several DSP codes acceleration techniques including fixed-point data types, TI (Texas Instruments) DSP intrinsic functions and others. In addition, we accelerate the PAM and the quantization and bit allocation modules by fast algorithms for DSP implementation. Through these modifications, the final AAC encoder version has about 77.89 percent improvement. Furthermore, we also successfully implement both the AAC decoder and encoder on the II's (Innovative Integration) Quixote DSP board. We adopt the burst block transmission mechanism for communication between the host and the target side. Finally, the speed of the AAC encoder and decoder on DSP implementation can achieve real-time operation.

誌謝

能夠完成這篇論文，最感謝的是我的指導教授－杭學鳴老師，在這兩年的研究生涯，老師不但給予我專業的知識與指導，並且培養了我認真踏實的研究態度，讓我獲益良多。

實驗室完善的設備及認真的環境，也幫助我能夠順利的完成這篇論文，我要感謝實驗室的學長、同學及學弟，特別是楊政瀚、陳繼大、吳俊榮學長給予我很多研究的建議，我也要感謝曾建統學長，他在音訊編碼方面做了很多的研究，讓我更加容易的實現於硬體平台上，還有實驗室一同奮鬥的志楹、景中、昱昇、朝雄、漢光等同學們，讓我在研究過程中遇到困難時，能夠互相討論和砥礪。另外我也要感謝我的朋友給予我生活上的支持及勉勵，使我充滿信心。

最後，我要感謝我的父母及家人，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，沒辦法一一詳列，在此，謹以這篇論文，獻給在研究生涯中所有關心以及幫助我的人，謝謝你們。

王盈閔

民國九十四年六月

List of Figures

List of Tables

Contents

Chapter 1

Introduction

MPEG stands for ISO “Moving Pictures Experts Groups.” It is a group work 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 Advanced Audio Coding (AAC) is an efficient audio algorithm standardized by ISO/IEC MPEG committee. The AAC can achieve indistinguishable quality at 128 kbits/s for stereo signals, and at 320 kbits/s for 5.1 multichannel audio. Hence, it can compress audio data at high quality with high compression efficiency. The MPEG-4 AAC mainly inherits MPEG-2 AAC (13818-7) and adds several tools to enhance the coding performance, such as temporal noise shaping (TNS), perceptual noise substitution (PNS), long time prediction (LTP), spectral band replication (SBR) and others.

In this thesis, our aim is to implement the MPEG-4 AAC encoder and decoder on the DSP processor. Hence, we adopt the DSP board made by Innovative Integration's Quixote to implement our program. The board houses a Texas Instruments' TMS320C6416 DSP and a Xilinx Virtex-II FPGA. The TI TMS320C6416 fixed-point processor has a rather good performance. Its instruction cycle frequency is 600MHz. It adopts the advanced VelociTI very long instruction word (VLIW) architecture that can execute eight instructions in parallel. In addition, we accelerate the MPEG-4 AAC encoder by some DSP coding techniques and several efficient algorithms.

Our contributions are the acceleration of the AAC encoder and the implementation of the AAC encoder and decoder. Through some DSP codes acceleration techniques and the fast algorithms of the PAM (psychoacoustic model) and the quantization and bit allocation modules in AAC encoder, the final AAC encoder version has about 77.89

percent improvement. Furthermore, the speed of the AAC encoder and decoder on DSP implementation can achieve real-time operation.

This thesis is organized as follows. In chapter 2, we describe operations of MPEG-2 AAC and MPEG-4 AAC. In chapter 3, we describe the DSP development environment and the communication interface provided by the DSP platform. In chapter 4, we speed up the AAC encoder program on DSP. In chapter 5, we successfully implement the AAC encoder and decoder on DSP platform. Finally, we give a conclusion and future work of our system.

Chapter 2

MPEG-2/4 Advanced Audio Coding

In this chapter, we will briefly introduce several basic concepts and major modules of the MPEG-2/4 AAC (Advanced Audio Coding) system. Details can be found in [1] and [2] respectively.

2.1 MPEG-2 AAC

In 1994, the MPEG-2 audio standardization committee defined a high quality multi-channel standard. It was the first-step of the development of “MPEG-2 AAC”. In 1997 April, the MPEG-2 AAC (ISO/IEC 13818-7) was standardized by the MPEG (Moving Pictures Expert Group). The aim of MPEG-2 AAC was to reach “indistinguishable” audio quality at the 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 results showed that MPEG-2 AAC needs 320 kbps to achieve the ITU-R quality requirements. This result showed that MPEG-2 AAC satisfied the ITU-R specifications.

The MPEG-2 AAC provides the transparent audio quality at the cost of discarding MPEG-1 backward-compatibility. The MPEG-2 AAC algorithm combines the coding efficiency of a high-resolution filter bank, prediction techniques, Huffman coding and other tools to achieve the audio quality at low data rates. And like most audio coding schemes, the MPEG-2 AAC algorithm compresses 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 use the masking properties of human hearing system to remove irrelevant audio signals. Besides, the MPEG-2 AAC system offers three profiles to fulfill the demand of different tradeoffs between audio quality, memory requirement and system complexity. For this purpose, the three profiles are defined as main profile, low-complexity (LC) profile and scalable sampling rate (SSR) profile. The main profile is intended for use when the processing power, and especially the memory, is not better. The LC profile is intended to use when the computing cycles and memory use are constrained, and the SSR profile is in use when a scalable decoder is required.

Next, we will briefly introduce each tool in this section. Fig 2.1 gives an overview of the MPEG-2 AAC encoder block diagram.

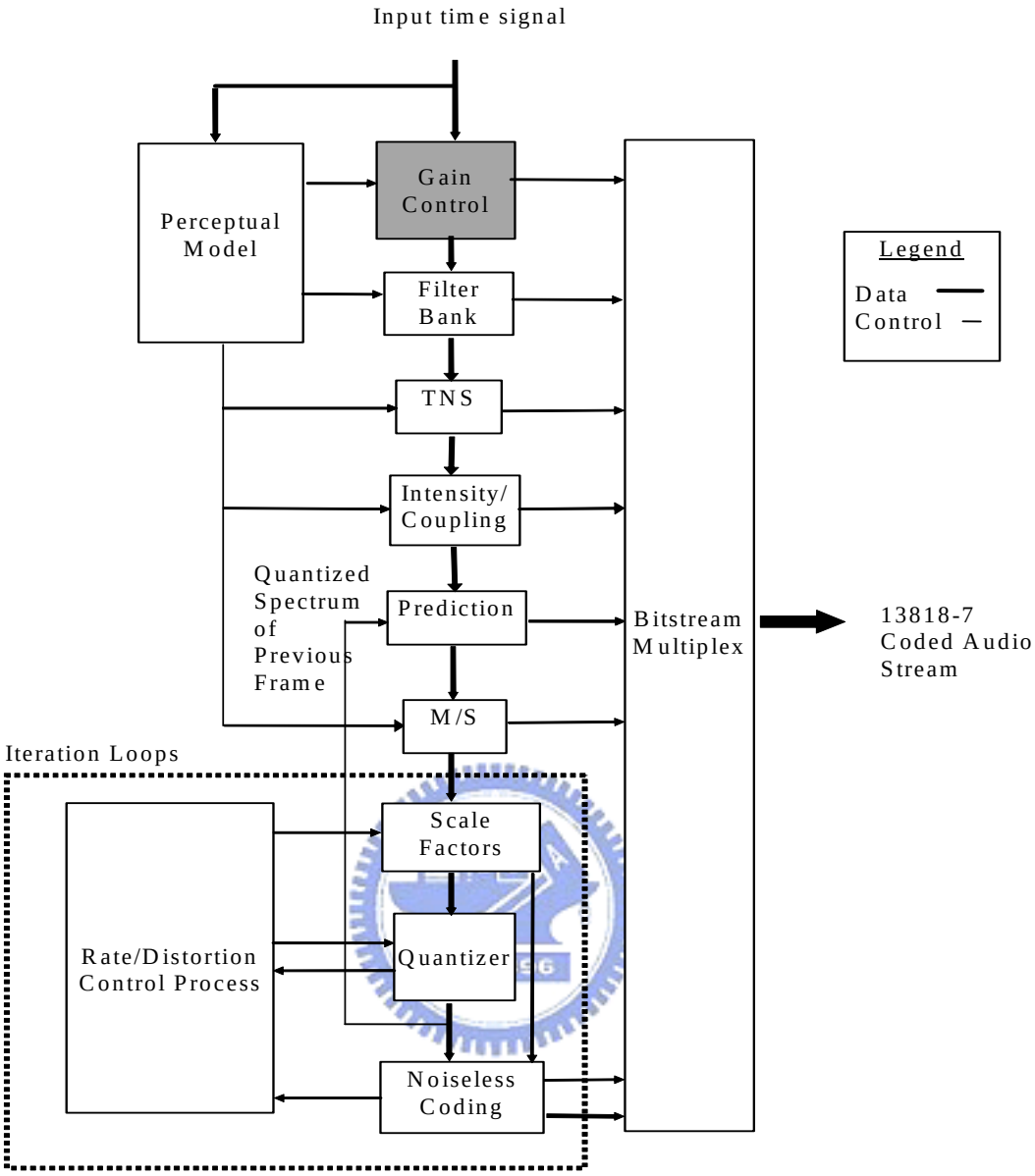


Fig 2.1 Block diagram for MPEG-2 AAC encoder [1]

2.1.1 Psychoacoustic Model

The psychoacoustic model is an essential component of the AAC encoder that enables its high performance. The job of the psychoacoustic model is to analyze the input audio signal and determine where the spectrum quantization noise can be allowed and to what extent. Then, the encoder uses this information to decide how to represent the input audio signal in the most way with the given limited number of code bits. In this process, the psychoacoustic model calculates the maximum distortion energy value which can be

masked by the signal. And this energy is called threshold. The threshold generation process has three inputs. They are:

1. The shift length for the threshold calculation process is called *iblen*. This *iblen* must remain constant over any particular application of the threshold calculation process. For long FFT *iblen* = 1024, for short FFT *iblen* = 128.
2. For each FFT type, the newest *iblen* samples of the signal, with the samples delayed (either in the filterbank or psychoacoustic calculation) such that the window of the psychoacoustic calculation is centered in the time-window of the codec time/frequency transform.
3. The sampling rate. There are sets of tables that will be used in the calculation process, and the tables are provided for the standard sampling rates. Sampling rate must necessarily remain constant over one implementation of the threshold calculation process.

The outputs of the psychoacoustic model are:

1. a set of Signal-to-Mask Ratios and thresholds, which are to be used by the encoder.
2. the delayed time domain data (PCM samples), which are to be used by MDCT.
3. the block type for the MDCT.
4. an estimate of the amount of bits should be used for encoding in addition to the average available bits.

Fig 2.2 [2] shows the block diagram for the psychoacoustic model in the MPEG-2 AAC encoder. Unlike the psychoacoustic model 1, this model does not make a dichotomous distinction between tonal and non-tonal components. Instead the spectral data is transformed to a “partition” domain and the fractions of the tonal and non-tonal components are estimated in each partition. This fraction ultimately determines the amount of masking.

For more detailed procedures for calculation, please see [2].

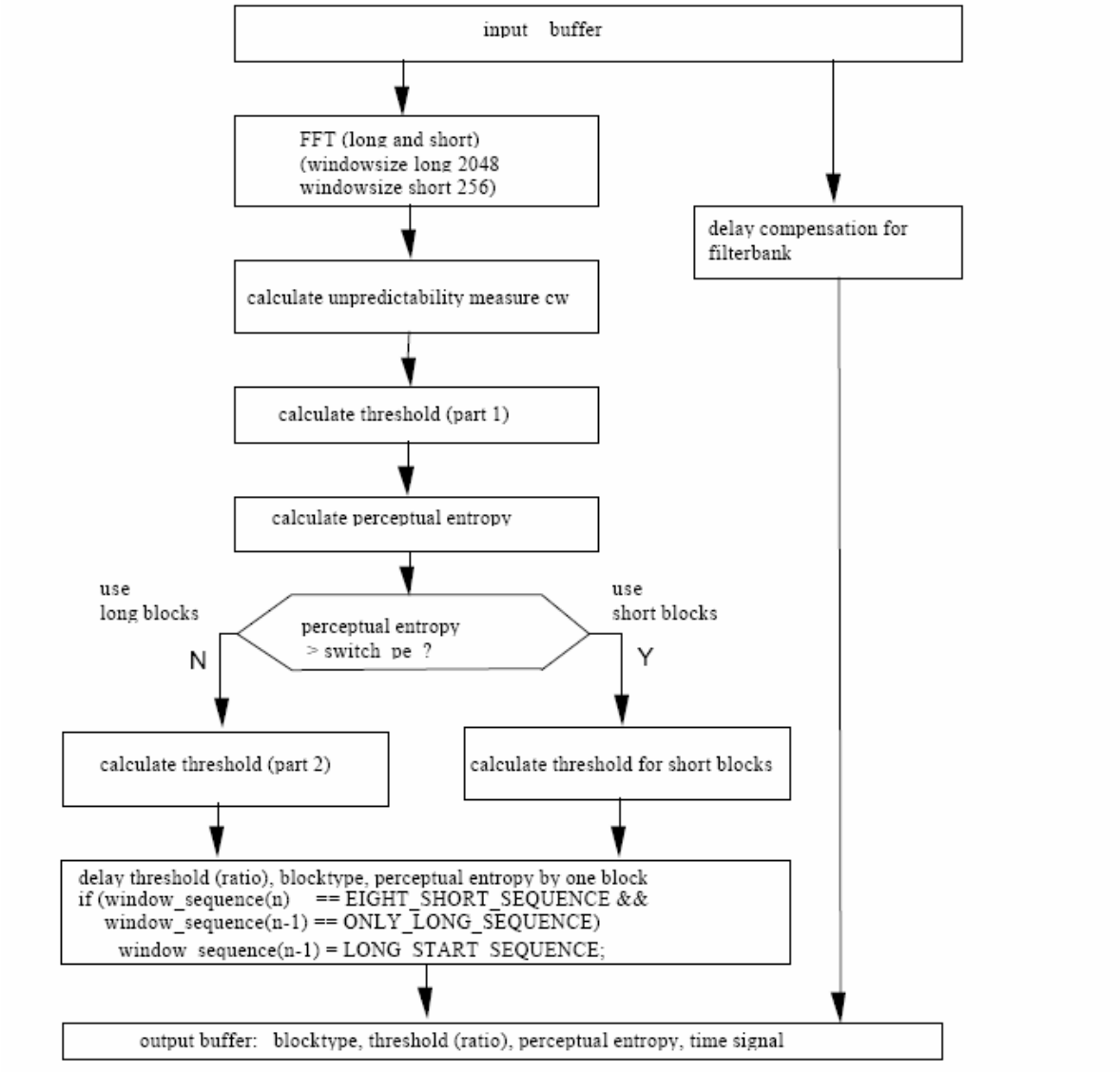


Fig 2.2 Block diagram of psychoacoustic model [2]

2.1.2 Gain Control

The gain control tool receives the input time-domain signals, and then ouputs gain_control_data and a gain controlled signal whose length is equal to the length of the MDCT window. The tool consists of a PQF (Polyphase Quadrature Filter), gain detectors and gain modifiers. The PQF divides the input signals into four equal width frequency bands. The gain detectors produce gain control data, which satisfies the MPEG bitstream syntax. They consist of the number of gain changes, the index of gain change positions and the index of gain change level. The gain modifier for each PQF band controls the

gain of each signal band. And the gain control tool can be applied to each of four bands independently. The block diagram for the gain control tool is shown in Fig 2.3.

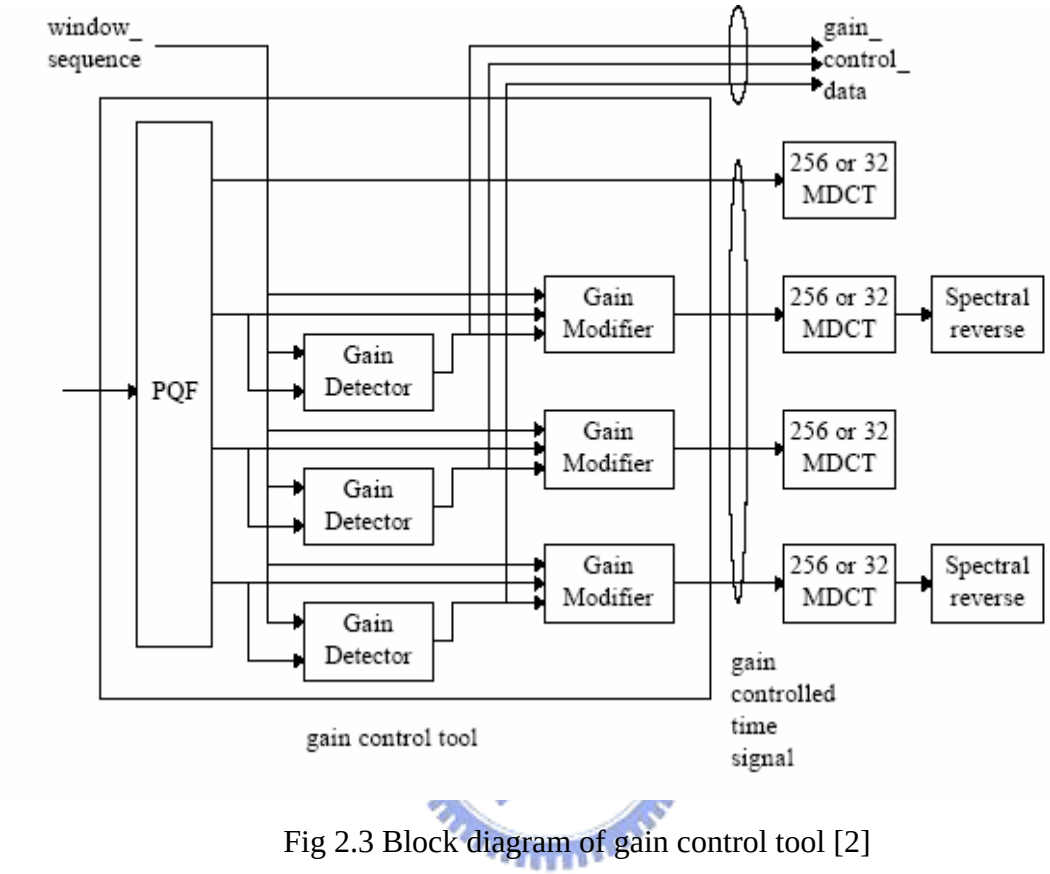


Fig 2.3 Block diagram of gain control tool [2]

2.1.3 Filterbank

The filterbank maps the signal samples into a spectral representation using a modified discrete cosine transformation (MDCT) with critical subsampling and overlapping subsequent windows. The MDCT employs TDAC (time-domain aliasing cancellation) technique.

In the encoder, the filterbank takes in the appropriate block of time samples, modulates them by an appropriate window function, and performs the MDCT. 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.

The mathematical expression of the MDCT is

$$X_{i,k} = 2\sum_{n=0}^{N-1} x_{i,n} \cos\left[\frac{2\pi}{N}(n+n_0)\left(k+\frac{1}{2}\right)\right], \qquad k = 0,1,\ldots,\frac{N}{2}-1 \tag{2.1}$$

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 match to the input signal characteristics. There are two resolutions in AAC, one with 1024 spectral coefficients (one long window) and one with eight sets of 128 coefficients (eight short windows) and the switching between them is supported through the use of transition windows. The encoder also selects the optimal shape for each of these windows between the Kaiser-Bessel-derived window (KBD) with improved far-off rejection and the sine window with a wider main lobe.

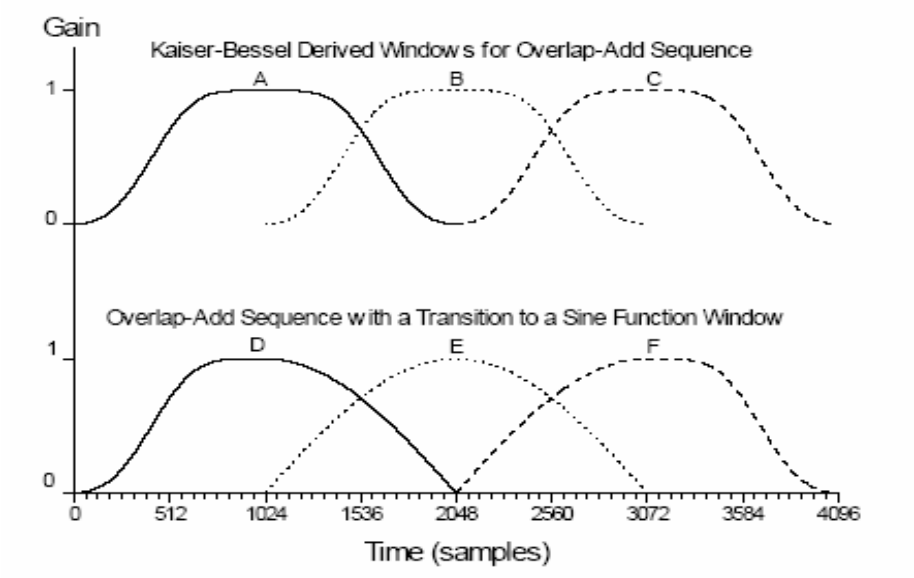


Fig 2.4 Window shape adaptation process [2]

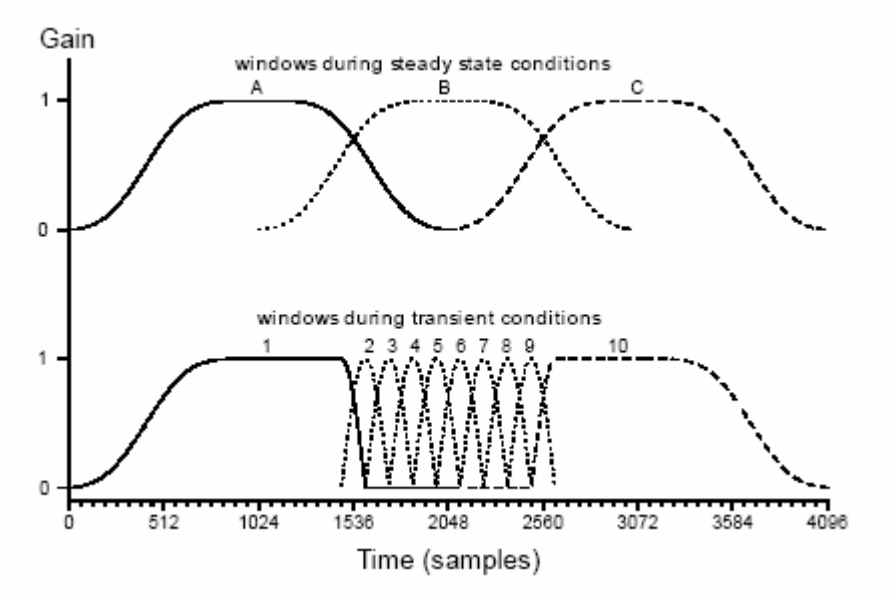


Fig 2.5 Block switching during transient signal conditions [2]

2.1.4 Prediction

Prediction is used for an improved redundancy reduction and is very effective in the stationary parts of a signal. The current spectral coefficient is estimated by the predictor based on the corresponding spectral coefficients of the preceding two frames and only the prediction errors need to be transmitted.

For each channel prediction is applied to the spectral components resulting from the filterbank. For each spectral component, there is one corresponding predictor resulting in a bank of predictors. Each predictor exploits the auto-correlation between the spectral component values of consecutive frames. The predictor coefficients are calculated from preceding quantized spectral components in the encoder. A second order backward-adaptive lattice structure predictor is working on the spectral component values of the 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 the prediction is activated, the quantizer is fed with the prediction error. Fig 2.6 shows the block diagram of prediction unit for one scalefactor band.

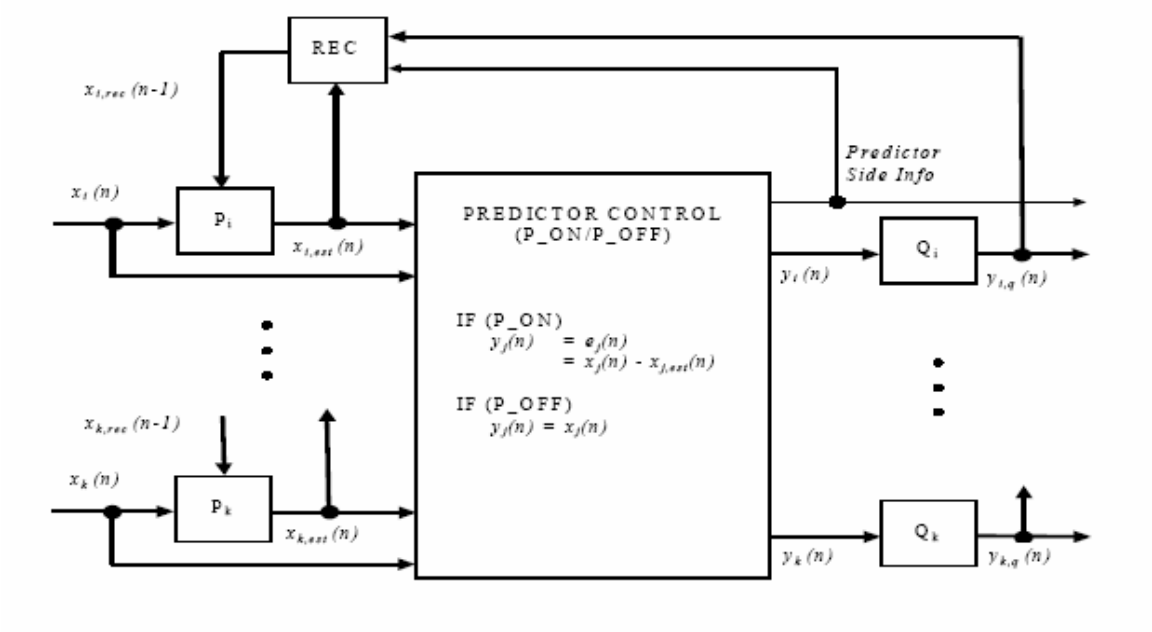


Fig 2.6 Prediction tool for one scalefactor band [2]

2.1.5 Temporal Noise Shaping (TNS)

The Temporal Noise Shaping tool is used to control the temporal shape of the quantization noise within each window of the transform, which is needed for transient and pitched signals. This is done by applying a filtering process to parts of the spectral data of each channel. The tool can provide considerable enhancement to the audio quality for the speech and transient signals.

2.1.6 Joint Stereo Coding

AAC joint stereo coding reduces the needed bitrate for stereo or multichannel signals more efficiently than separate coding of several channels. There are two different joint stereo methods that can be selected for coding of different frequency bands to optimize the resulting bitrate: M/S stereo coding and intensity stereo coding.

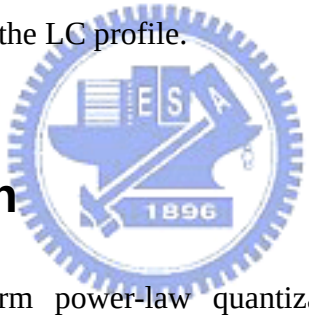
1. M/S stereo coding:

The decision to code left and right coefficients as either left/right (L/R) or mid/side

(M/S) is made on a noiseless coding band by noiseless coding band basis for all spectral coefficients in the current block. M/S stereo coding is very efficient for near monophonic signals, because it use a sum (M) and a difference (S) channel instead of left and right channels and the difference signals is very small in this case. If the high correlated left and right signals could be summed, the require bits to code this signals will be less. Therefore, when the left and right signals' correlation is higher than a threshold, the M/S stereo coding tool will operate on transforming the L/R signals to M/S signals.

2. Intensity stereo coding:

The intensity stereo coding 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. In this AAC system, the intensity stereo coding mechanism is implemented in the LC profile.



2.1.7 Quantization

AAC uses the nonuniform power-law quantization, where smaller values are quantized finer, so that quantization noise is stronger at larger values and is easier masked. Scalefactors are used to scale the spectral coefficients before the quantization to be able to control the power of the introduced quantization noise.

$$ix(i) = \mathrm{sgn}(xr(i)) \cdot NINT\left[\frac{|xr(i)|}{\sqrt[4]{2^{quantizer\_stepsize}}})^{0.75} - 0.0946 \right] \tag{2.2}$$

The AAC quantization module consists of three levels. The top level calls a subroutine named "outer iteration loop", which calls the subroutine "inner iteration loop". The outer iteration loop (distortion control loop) controls the quantization noise which is produced by the quantization of the frequency domain lines within the inner iteration loop to maintain perceptual performance. The inner interation loop (rate control loop)

calculates the actual quantization of the frequency domain data to maintain bit rate.

2.1.8 Noiseless Coding

In the AAC encoder the input to the noiseless coding module is the set of 1024 quantized spectral coefficients. Since the noiseless coding is done inside the quantizer inner loop, it is part of an iterative process that converges when the total bit count is within some interval surrounding the allocated bit count. The noiseless coding stage in AAC uses sectioning and Huffman coding (entropy coding) and exploits statistical redundancy to efficiently encode the 1024 coefficients without further loss of information.

The noiseless coding segments the set of 1024 quantized spectral coefficients, such that a single Huffman codebook is used to code each section. The Huffman coding is used to represent n-tuples of quantized coefficients, with 12 codebooks can be used. The spectral coefficients within n-tuples are ordered and the n-tuple size is two or four coefficients. Each codebook specifies the maximum absolute value that it can represent and the n-tuple size.

2.2 MPEG-4 AAC Version1

MPEG-4 is formal as its ISO/IEC designation “ISO/IEC 14496”, and it includes the major parts: Systems, Audio, Video and DMIF. Specially, compared to previous MPEG standard, MPEG-4 has the following concepts: universality, scalability, object-based representation, content-based interactivity and natural and synthetic representations.

MPEG-4 AAC Version 1 was finalized in October 1998 and became an International Standard in the first months of 1999. It is fully backward compatible with MPEG-2 AAC, and includes some 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 PNS tool and the LTP tool are available to enhance the

coding performance for the noise-like and very tonal signals, respectively. The TwinVQ tool is provided to cover very low bitrates. This new scheme which combined AAC with TwinVQ is officially called "General Audio (GA)." Next, we will briefly introduce these new tools.

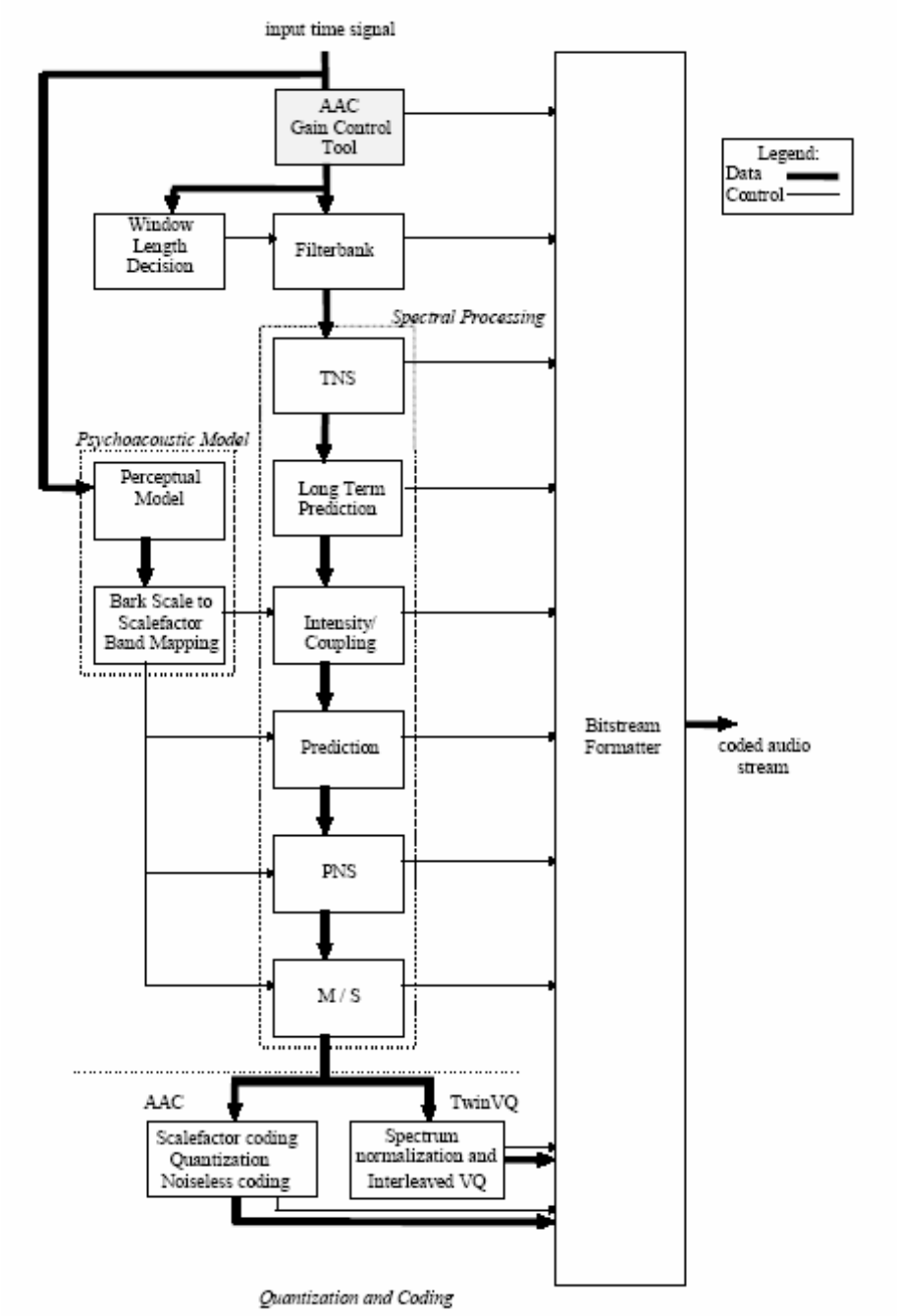


Fig 2.7 Block diagram of MPEG-4 GA encoder [2]

2.2.1 Long Term Prediction (LTP)

The LTP tool is well-known from speech coding and is used to exploit redundancy in the speech signal which is related to the signal periodicity as expressed by the speech pitch. The LTP tool has been integrated into the audio coder where quantization and coding is performed on the input signal. Fig 2.8 shows the combined LTP and coding system.

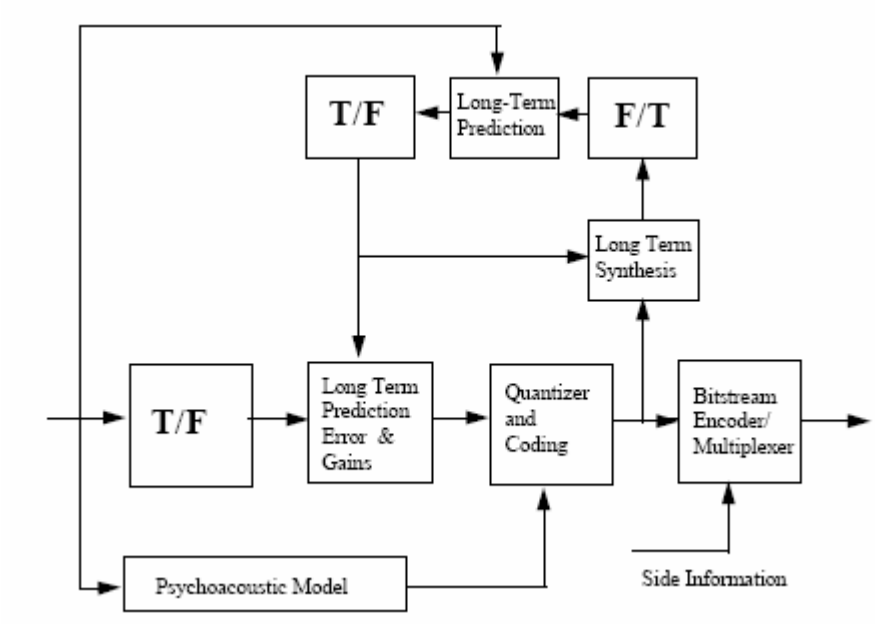


Fig 2.8 LTP in the MPEG-4 General Audio encoder [2]

The LTP is used to predict the input signal based on the quantized values of the preceding frames which were transformed back to a time domain representation by the inverse filterbank and the associated inverse TNS operation. Comparing this decoded signal to the input signal, the optimum pitch lag and gain factor is determined. Then, the difference between the predicted signal and the original signal 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. This is achieved by means of the “frequency selective switch” (FSS).

The LTP tool provides considerable coding gain for stationary harmonic signals and some gain for non-harmonic tonal signals. Besides, the computational complexity of the

LTP tool is much less than original prediction tool.

2.2.2 Perceptual Noise Substitution (PNS)

The PNS tool allows for a very compact representation of noise-like signal components because only the signaling and the energy information is transmitted once for a scalefactor band instead of the set of quantized and coded spectral coefficients. Therefore, it increases compression efficiency for certain types of input signals. Fig 2.9 shows the PNS concept.

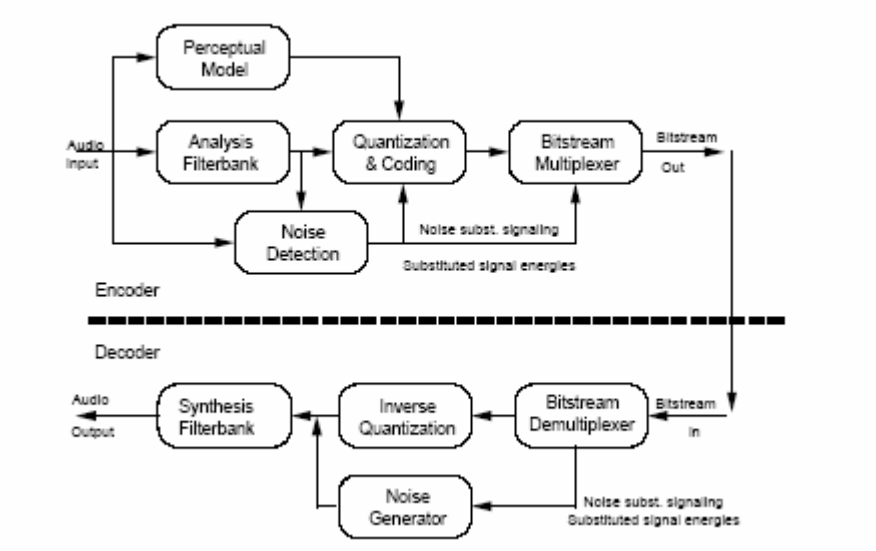


Fig 2.9 Principle of Perceptual Noise Substitution [2]

2.2.3 TwinVQ

The TwinVQ tool is an alternative VQ-based coding kernel. It can provide good coding performance at very low bitrates (at or below 16kbps).

When it performs the quantization of the spectral coefficients, the spectral coefficients will first be normalized to a specified target range and then be quantized by using the weighted vector quantization (VQ) process. The Fig 2.10 shows the TwinVQ tool module.

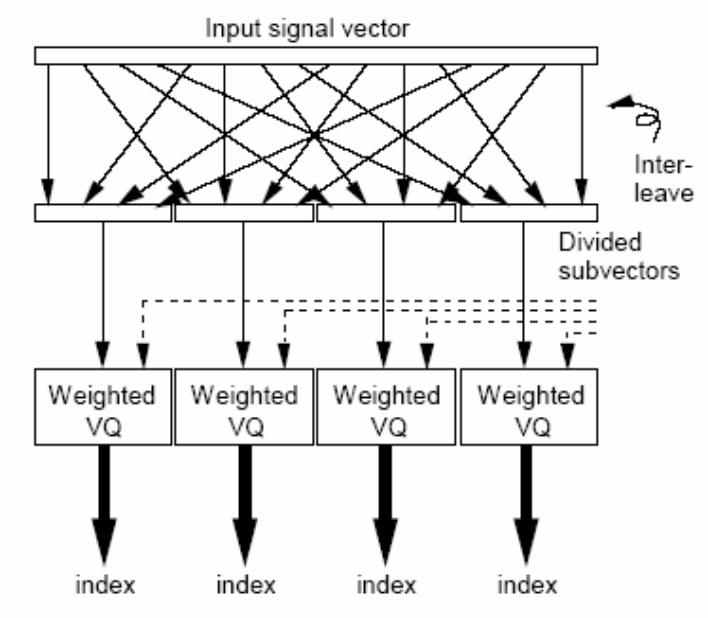


Fig 2.10 TwinVQ quantization scheme [2]

2.3 MPEG-4 AAC Version2

MPEG-4 AAC Version 2 was finalized in 1999. Compared to MPEG-4 AAC version 1, it adds some new tools without replacing any existing tools of version 1. So, it is fully backward compatible to version 1. The version 2 provides the following new functionalities: Error Robustness, Low-Delay Audio Coding, Fine grain scalability and so on. Next, we will briefly introduce these new tools in this section.

2.3.1 Error Rubustness

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 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.

2.3.3 Fine Grain Scalability

Bitrate scalability, also known as embedded coding, is a very desirable functionality. In order to provide efficient small step scalability for the AAC, the Bit-Sliced Arithmetic Coding (BSAC) tool is available in version 2. This tool is used in combination with the AAC coding tools and replaces the noiseless coding of the quantized spectral data and the scalfactors. BSAC provides scalability in steps of 1kbps per audio channel, which means 2kbps steps for a stereo signal. One base layer bitstream and many small enhancement layer bitstreams are used.

2.3.4 Parametric Audio Coding

The Parametric Audio Coding tools combine very low bitrate coding of general audio signals with the possibility of modifying the playback speed or pitch during decoding without the need for an effects processing unit. In combination with the speech and audio coding tools of version 1, improved overall coding efficiency is expected for applications of object based coding allowing selection and switching between different coding techniques.

2.3.5 CELP Silence Compression

The silence compression tool reduces the average bitrate because of a lower-bitrate compression for silence. In the encoder, a voice activity detector is used to distinguish between regions with normal speech activity and those with silence or background noise. During normal speech activity, the CELP coding as in version 1 is used. Otherwise a Silence Insertion Descriptor (SID) is transmitted at a lower bitrate. This SID enables a Comfort Noise Generator (CNG) in the decoder. The amplitude and spectral shape of this comfort noise is specified by energy and LPC parameters similar as in a normal CELP frame. These parameters are an optional part of the SID and thus can be updated as

required.

2.3.6 Extended HVXC

The variable bitrate mode of 4.0 kbps maximum is additionally supported in version 2 HVXC. In the version 1 HVXC, variable bitrate mode of 2.0 kbps maximum is supported as well as 2.0 and 4.0 kbps fixed bitrate mode. In version 2, the operation of the variable bitrate mode is extended to work with 4.0 kbps mode. In the variable bit-rate mode, non-speech part is detected from unvoiced signals, and smaller number of bits are used for non-speech part to reduce the average bitrate. When the variable bit-rate mode of 4.0 kbps maximum is used, the average bit rate goes down to approximately 3.0 kbps with typical speech items.

Chapter 3

DSP Implementation Environment

In our project, we choose digital signal processor (DSP) platform to implement MPEG-4 AAC encoder and decoder. The DSP baseboard we use is made by Innovative Integration's (II's) Quixote, which houses Texas Instruments' TMS320C6416 DSP chip and Xilinx Virtex-II FPGA. In this chapter, we will introduce the DSP baseboard and DSP chip. At the end, the data transmission process from the host PC to the target DSP and vice versa is also introduced.

3.1 DSP Baseboard

The Quixote DSP Baseboard card is shown in Fig. 3.1 and the architecture is shown in Fig. 3.2 [5]. Quixote combines one TMS320C6416 600MHz 32-bit fixed-point DSP with one two- or six-million-gate Virtex-II FPGA on the DSP baseboard, utilizing the signal processing technology to provide extreme processing flexibility and 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. The SDRAM provides a large, fast external memory pool for DSP data and code. The SBSRAM is configured as independent banks for fast data processing storage, directly attached to the FPGA. Developers can build complex signal processing systems by integrating these reusable logic designs with their specific application logic.

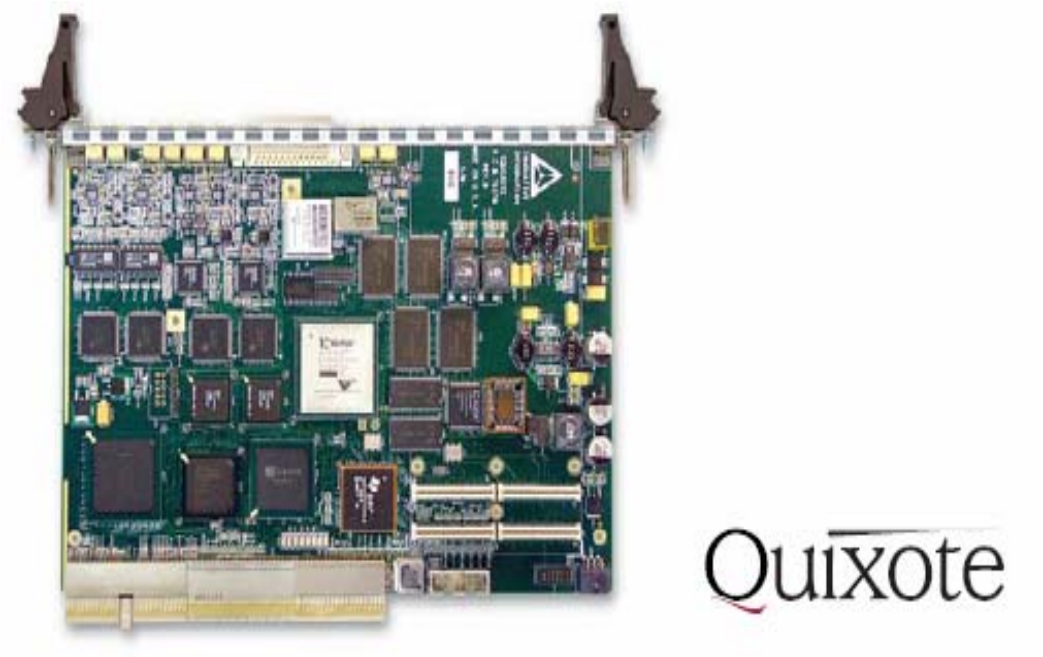


Fig. 3.1 Innovative Integration's Quixote DSP Baseboard Card [5]

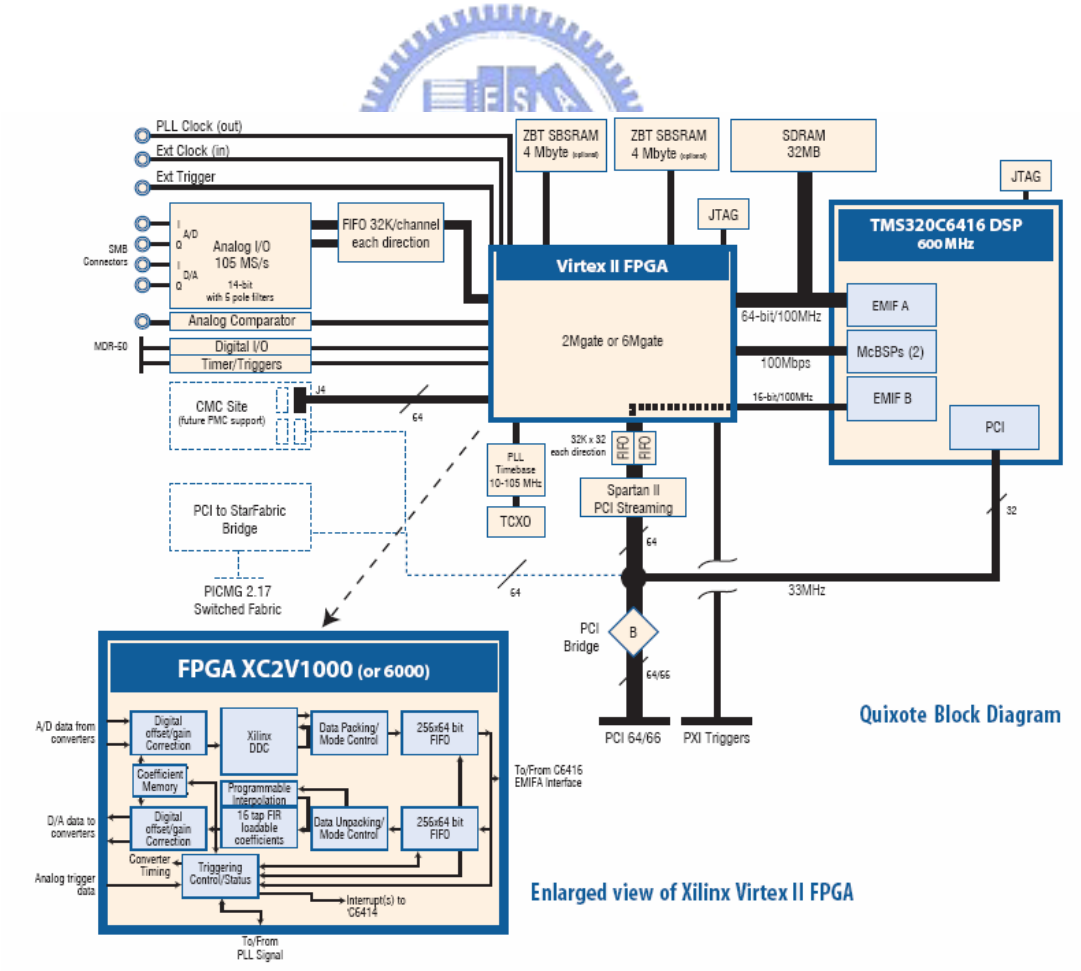


Fig. 3.2 Block Diagram of Quixote [5]

3.2 DSP Chip

The TMS320C64x fixed-point DSP is using the VelociTI architecture. The VelociTI is a high-performance, advanced VLIW (very long instruction word) architecture, making it an excellent choice for multichannel, multifunctional, and performance-driven applications. VLIW can achieve high performance through increased instruction-level parallelism, performing multiple instructions during a single cycle. Because parallelism takes the DSP well beyond the performance capabilities of traditional superscalar systems, it is the key to high performance.

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

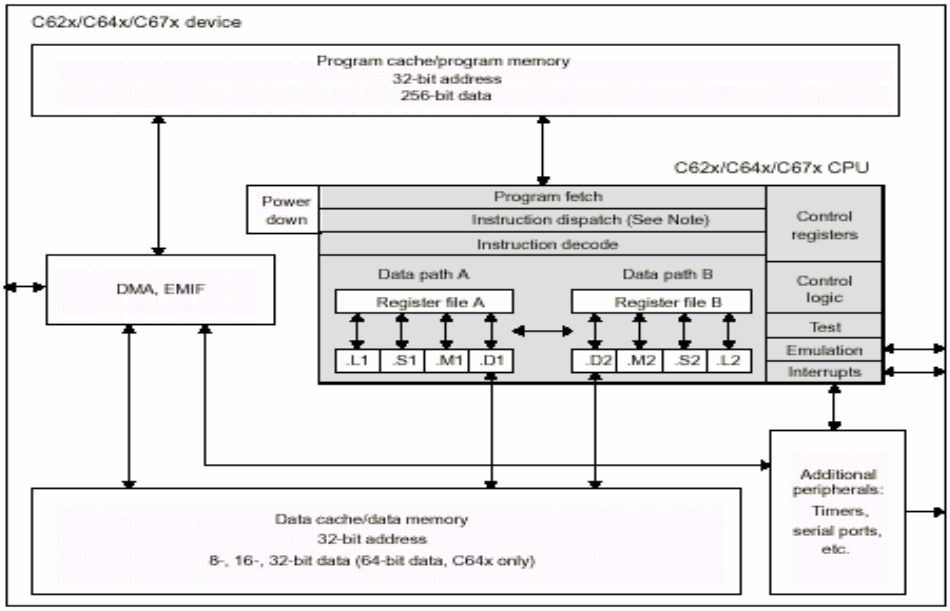


Fig 3.3 Block diagram of TMS320C6x DSP [6]

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. Peripherals such as a direct memory access (DMA) controller, power-down logic, and external memory interface (EMIF) usually come with the CPU, while peripherals such as serial ports and host ports are on only certain devices.

In the following subsections, we will introduce several important parts of the TMS320C64x DSP Chip.

3.2.1 Central Processing Unit (CPU)

The TMS320C6416 CPU contains of eight independent functional units, sixty-four general purpose registers and control registers. Besides above, it also has the program fetch unit, instruction dispatch unit (attached with advanced instruction packing), instruction decode unit, two data path (A and B, each with four functional units), test unit, emulation unit, interrupt logic and two register files (A and B with respect to the two data paths). The architecture is illustrated in Fig. 3.3 and Fig. 3.4.

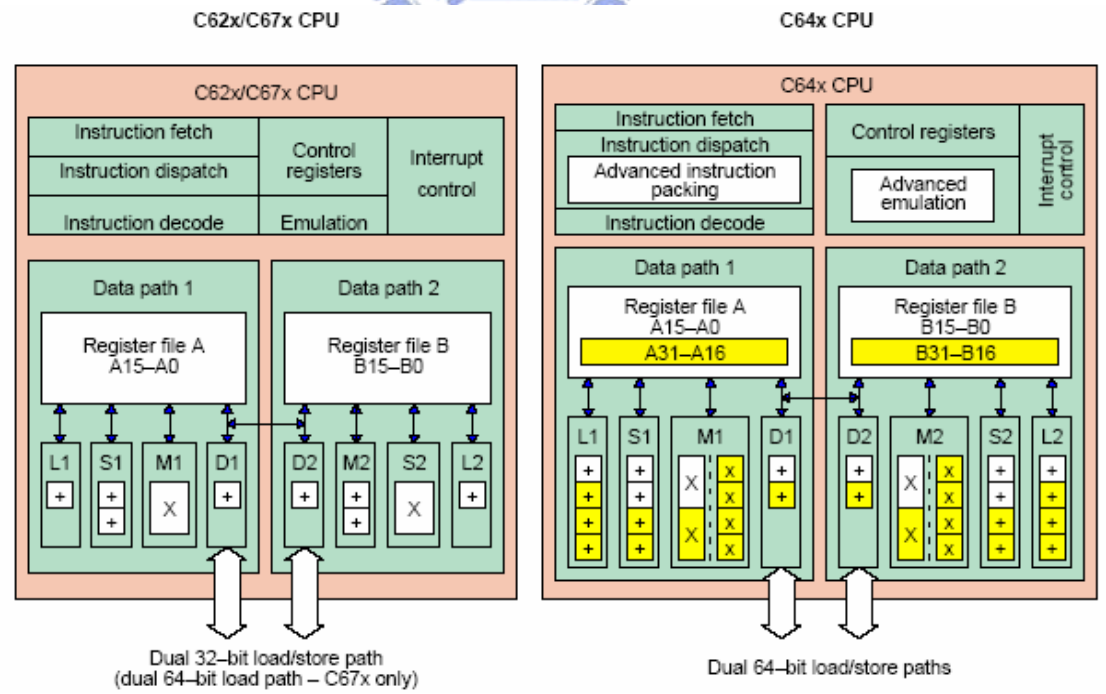


Fig. 3.4 The TMS320C64x DSP Chip Architecture and Comparison with Ancient TMS320C62x/C67x Chip.

The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units during one 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.

The program pipelining is also the important feature to get parallel instructions working properly. There are three stages of pipelining: program fetch, decode, and execute. In the fetch stage, the program address is generated in the CPU, and then the program address is sent to memory. After a memory read occurs, the fetch packet is received at the CPU. In the decode stage, the instructions in execute packet are assigned to the appropriate functional units. And then, the source registers, destimation registers, and associated paths are decoded for the execution of the instructions in the functional units. The execute stage is composed of five phases, and instructions are executed in the stage. Different types of instructions require different numbers of phases to complete the execution.

3.2.2 Data Path

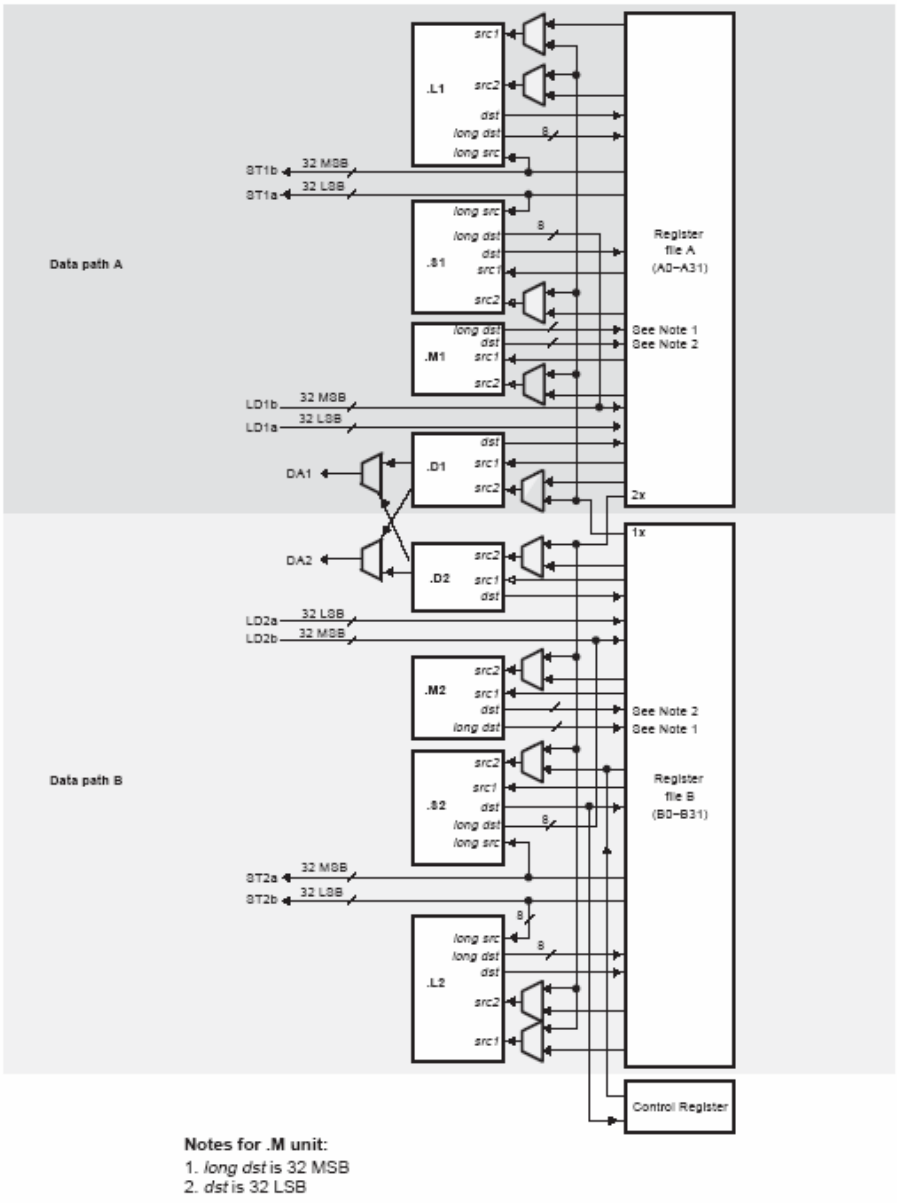


Fig 3.5 TMS320C64x CPU Data Path [6]

There are two general-purpose register files (A and B) in the C6000 data paths as shown in Fig 3.5. The C64x DSP register is double the number of general-purpose registers that are in the C62x/C67x cores, with 32 32-bit registers.

The C64x architecture has eight functional units that could be further divided into two data paths A and B. Each path has one unit for multiplication operations (.M), one for logical and arithmetic operations (.L), one for branch, bit manipulation, and arithmetic operations (.S), and one for loading/storing, address calculation and arithmetic operations

(.D). 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. There can be a maximum of two cross-path source reads per cycle. Fig 3.6 and Fig 3.7 show the functional units 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. 3.6 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. 3.7 Functional Units and Operations Performed (Cont.) [7]

3.2.3 Memory

3.2.3.1 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 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 internal port to access internal program memory, with an instruction-fetch width of 256 bits.

3.2.3.2 External Memory and Peripheral Options

The external memory and peripheral options of C6416 contain

- ☐ Large on-chip RAM, up to 7M bits
- ☐ Program cache

- 2-level caches
- 32-bit external memory interface supports SDRAM, SBSRAM, SRAM, and other asynchronous memories for a broad range of external memory requirements and maximum system performance.
- DMA Controller transfers data between address ranges in the memory map without intervention by the CPU. The DMA controller has four programmable channels and a fifth auxiliary channel.
- EDMA Controller performs the same functions as the DMA controller. The EDMA has 16 programmable channels, as well as a RAM space to hold multiple configurations for future transfers.
- HPI is a parallel port through which a host processor can directly access the CPU's memory space. The host device has ease of access because it is the master of the interface. The host and the CPU can exchange information via internal or external memory.
- McBSP (multichannel buffered serial port) is based on the standard serial port interface found on the TMS320C2000 and C5000 platform devices. Besides, the port can buffer serial samples in memory automatically with the aid of the DMA/EDMA controller. It also has multichannel capability compatible with the T1, E1, SCSA, and MVIP networking standards.

3.3 Data Transmission Mechanism

Many applications of the Quixote baseboard involve communication with the host CPU in some manner. All applications at a minimum must be reset and downloaded from the host, even if they are isolated from the host after that. The simplest method supported is a mapping of Standard C++ I/O to the Uniterminal applet that allows console-type I/O on the host. This allows simple data input and control and the sending of text strings to the user. The next level of support is given by the Packetized Message Interface. This allows more complicated medium rate transfer of commands and information between the host and target. It requires more software support on the host than the Standard I/O

does. For full rate data transfers Quixote supports the creation of data streaming to the host, for the maximum ability to move data between the target and host. On Quixote baseboards, a second type of busmaster communication between target and host is available for use, the CPU Busmaster interface.

3.3.1 DSP Streaming Interface

The DSP Streaming interface is bi-directional. Two streams can run simultaneously, one running from the analog peripherals through the DSP into the application. This is called the “Incoming Stream”. The other stream runs out to the analog peripherals. This is the “Outgoing Stream”. In both cases, the DSP needs to act as a mediator, since there is no direct access to analog peripherals from the host. This arrangement allows the DSP to process the streams as they move from the application to the hardware.

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

3.3.2 Burst Block Transmission

The 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 subsequence data accumulates to allow for an efficient transfer.

The interface uses a different model: it transfers discrete blocks between the source and destination. Each data buffer is transferred completely to the destination in a single operation. The data buffers transferred can be of different sizes. At the destination, the destination buffer is re-sized to allow the incoming data to fit.

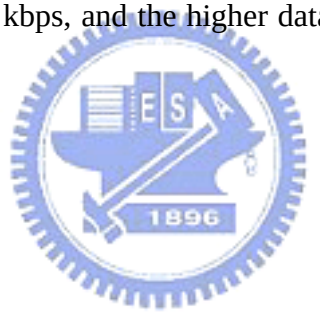
In this simple blocking interface, there are sending and receiving functions can be

used. The sending function will not return until the transfer has completed and the buffer is ready for reuse. Similarly, the receiving function waits until data has arrived from the data source and transferred into the data buffer before returning.

3.3.3 Message Exchange

Besides the above interfaces, the DSP and host have a lower bandwidth communications link for sending commands or out-of-band information between target and host. Software is provided to build a packet-based message system between the target and the host. These packets can provide a simple yet powerful means of sending commands and information across the link.

A set of sixteen mailboxes in each direction to and from host are shared with the 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.



Chapter 4

MPEG-4 AAC Encoder Acceleration on DSP

In this chapter, we will describe the MPEG-4 AAC code acceleration on DSP. We will first introduce TI's code development environment, describe how to optimize the C/C++ code for DSP architecture, and then discuss how to optimize the functions for DSP execution.

4.1 TI's Code Development Environment

In this section, we will briefly introduce the CCS (Code Composer Studio) tool for DSP, and describe how to develop C/C++ code for the given DSP architecture.

4.1.1 The Code Composer Studio

The Code Composer Studio (CCS) is a helpful tool for developing the DSP codes. We briefly describe some of its features related to our implementation below. The details can be found in [6].

- Compiles code and generates Common Object File Format (COFF) output file.
- Provides debug options such as step over, step in, step out, run free, and so on.
- Watches any memory sections when the DSP halts.
- Probes a PC file stream into or from the target memory location.

☐ Counts the instruction cycles between successive profile-points.

We mainly use the CCS tool for debugging, refining, optimizing, and implementing our C codes on DSP. The profile-points help us to evaluate if our changes to the codes are better or not. Besides, we must write the host code and target code with the burst block transmission in order to implement our system on the DSP platform.

4.1.2 Code Development Flow

The DSP code development can be divided into three steps.

- Step1：Develop the C code like standard ANSI C code without any regard to the particular structure of the C64x. Then, use the debugger to profile the code to identify any inefficient areas in the code. If the performance is not satisfactory, go to step2.
- Step2：Use DSP intrinsics and optimization techniques for code generation to improve the C codes. Refine the C code procedures such as compiler options, intrinsics, statement, data type modifiers, and code transformations. If the code efficiency is still not sufficient, proceed to step3.
- Step3：Extract the most time-critical areas and replace the C code with linear assembly, then use assembly optimizer to optimize the code, such as resource allocation

Generally, we do not go to step3 because the linear assembly is too detail. Doing assembly programming is difficult and assembly codes are hard to maintain. The recommended code development flow involves utilizing the C6000 code generation tools to aid optimization rather than forcing the programmer to code by hand in assembly. These advantages allow the compiler to do the instruction selection, parallelizing, pipelining, and register allocation. Figure 4.1 shows the steps of the software development flow [6].

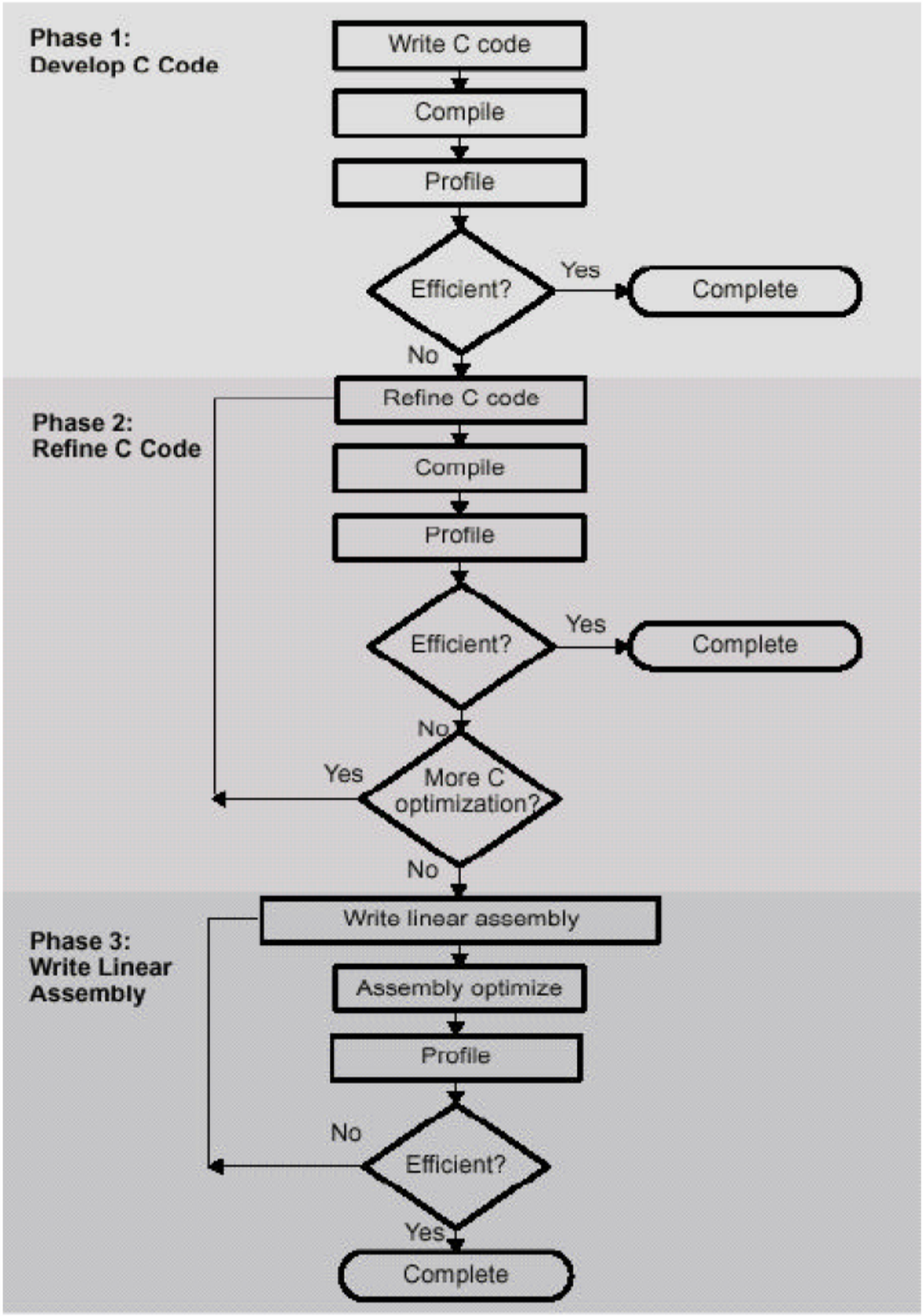


Fig. 4.1 Code development flow of C6000

4.2 Profile of AAC on DSP

We do the essential modifications on the MPEG-4 AAC source C code, and then implement the modified C code on DSP. In order to identify the computational intensive parts of the MPEG-4 AAC encoder, we first use TI CCS profiler to analyze it. Our test sequence is “guitar”, and the data length is about 0.1 second. Table 4.1 shows the profile results at 64k bit rate. We can see clearly that the psycho-acoustic model and the quantization and bit-allocation module are two major computational parts of the AAC encoder. Therefore, we should accelerate these two parts.

Function	Execution cycles	Percent (%)
Total	2,126,810,017	100
Psycho-acoustics	980,246,737	46.09
Filterbank	137,817,289	6.48
Quantization and Bit-allocation	1,000,238,751	47.03
Others	8,507,240	0.4

Table 4.1 Profile of AAC encoder on C64x DSP

4.3 DSP Code Acceleration Methods

Improving the execution cycles of the AAC encoder is the main task of our system implementation on DSP. In this section, we will describe several methods that can accelerate our code and reduce the execution time on the C64x DSP. Some of these methods are supported by the features of C64x.

4.3.1 Setting of Compiler Options

The Code Composer Studio (CCS) is a useful GUI tool that helps programmers in

developing DSP codes. Its compiler offers a complicated optimization process that includes several advanced techniques and it takes advantages of the features of the C6000 architecture. Hence, we can configure some setting of the compiler options to optimize our DSP codes efficiently. Table 4.2 shows the compiler options for performance enhancement and Table 4.3 shows those to avoid.

Option	Description
–o3†	Represents the highest level of optimization available. Various loop optimizations are performed, such as software pipelining, unrolling, and SIMD. Various file level characteristics are also used to improve performance.
–oi0	Disables all automatic size–controlled inlining, (which is enabled by –o3). User specified inlining of functions is still allowed.
–pm‡	Combines source files to perform program–level optimization by allowing visibility to the entire application source.

Table 4.2 Compiler Options for Performance Enhancement [6]

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

Table 4.3 Compiler Options to Avoid on Performance Enhancement [6]

4.3.2 Fixed-point Coding

The C6000 compiler defines a size for each data type:

- ☐ char 8bits
- ☐ short 16bits
- ☐ int 32bits
- ☐ long 40bits
- ☐ float 32bits
- ☐ double 64bits

The C64x DSP is a fixed-point processor, so it can only perform fixed-point operations. Although the C64x DSP can simulate floating-point processing, it takes a lot of extra clock cycles to do the same job. The “char”, “short”, “int” and “long” are the fixed-point data types, and the “float” and “double” are the floating-point data types. We test C64x DSP processing time of the assembly instructions “add” and “mul” for different data types. Table 4.4 shows the results. We can clearly see that the floating-point data types need more computation time than the fixed-point data types. Hence, we can accelerate our DSP codes in computation time efficiently by converting the data types from floating-point to fixed-point.

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 4.4 Processing time on the C64x DSP for different data types

4.3.3 Loop Unrolling

Loop unrolling expands the loops so that all iterations of the loop appear in the code.

It often increases the number of instructions available to execute in parallel. When our codes have conditional instructions, sometimes the compiler may not be sure that the branch will occur or not. It needs more waiting time for the decision of branch operation. If we do loop unrolling, some of the overhead for branching instruction will be reduced. Example 4.1 is the loop unrolling and table 4.5 shows the result.

(a)	(b)

/*Before unrolling*/
 	
int i,a=0,b=0;
for (i=0;i<10;i++)
{
a+=i;
b+=i;
}
	/*After unrolling*/
	
	int i=0,a=0,b=0;
	a+=i; b+=i; i++;
	a+=i; b+=i; i++;
	a+=i; b+=i; i++;
	a+=i; b+=i; i++;
	a+=i; b+=i; i++;
	a+=i; b+=i; i++;
	a+=i; b+=i; i++;
	a+=i; b+=i; i++;
	a+=i; b+=i; i++;
	a+=i; b+=i; i++;

Example 4.1 loop unrolling

	(a)	(b)
Execution cycles	436	206
Code size	116	476

Table 4.5 Comparison between with unrolling and without unrolling

We can see clearly that the clock cycle decreases after loop unrolling, but the code size is larger than the original. So generally speaking, if one iteration can execute many instructions, the code size is larger, but it runs faster.

4.3.4 Using Intrinsics

The TI C6000 compiler provides many special functions that map C codes directly to inlined C64x instructions, to increase C code efficiently. These special functions are called intrinsics. So if the instructions have equivalent intrinsic functions, we can replace them by intrinsic functions directly. The execution time will be decreased because of using intrinsics. Fig 4.2 shows some examples of the intrinsic functions for the C6000 DSP. The entire list of intrinsics for the C6000 DSP can be found in [6].

C Compiler Intrinsic	Assembly Instruction	Description	Device
int _**abs**(int *src2*); int_**labs**(long *src2*);	**ABS**	Returns the saturated absolute value of src2.	
int _**abs2** (int src2);	**ABS2**	Calculates the absolute value for each 16–bit value.	'C64x
int _**add2**(int *src1*, int *src2*);	**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.	
int _**add4** (int *src1*, int *src2*);	**ADD4**	Performs 2s–complement addition to pairs of packed 8–bit numbers.	'C64x

Fig 4.2 Intrinsic functions of the TI C6000 series DSP (partial list) [6]

4.3.5 Packet Data Processing

We often use a single load or store instruction to access multiple data consecutively located in memory to maximize data throughput. For example, 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. This method can improve the code efficiency substantially. In addition, some of the intrinsic functions enhance the efficiency in a similar way.

4.3.6 Register and Memory

When the accessed data are located in the external memory, we need more clock cycles in transfering data time. So we can use registers to store data in order to reduce transfer time in operation. In DSP code, the pointer, malloc function and so on will locate data in memory. Therefore, sometimes we can adequately modify code to avoid frequently accessing data from/to memory so that the execution time will be decreased.

4.3.7 Using Macros

Because the software-pipelined loop can not contain function calls, it takes more clock cycles to complete the function call. Hence, we can change the functions to the “define” macros under some conditions. In addition, replacing the function with the macro can cut down the code for initial function definition and reduce the number of branches. However, macros are expanded each time they are called if the function has a number of instructions, it is not efficient in memory usage.

4.3.8 Linear Assembly

When we are not satisfied with the efficiency of assembly codes which are generated by the TI CCS compiler, we can convert parts of the C codes into linear assembly and then optimize the assembly directly. But this process generally is too detail and very time consumption in practice. Hence, we will do this process at last if we have strict constrains in processor performance and we have no other algorithms selection.

4.4 Psychoacoustic Model

From AAC encoding profile in table 4.1, we can see clearly that the psychoacoustic model plays an important role in execution time. To improve the performance, we replace the psychoacoustic model with a new algorithm that was proposed by [9]. Next, we will briefly describe the new algorithm and show the simulation results after improving its performance.

4.4.1 Optimization of PAM

Firstly, we briefly describe the original psychoacoustic model of AAC encoder. Fig 4.3 [9] is this block diagram of PAM (psychoacoustic model). In steps 1-2, the auditory spectrum is computed using the FFT. Then, the real-part spectrums lead to the calculation of partitioned energy, and the imaginary-part spectrums result in the calculation of the unpredictability measure c(w). The unpredictablility measure is first weighted with the energy in each partition, deriving a partitioned unpredictability measure. Then in step 5, both partitioned energy and unpredictability are convolved with the spreading function in order to estimate the effects across the partitioned bands. For each partition, the ratio of the convolved partitioned unpredictability over the convolved partitioned energy spectrum is determined. Then, the tonality index is derived from the logarithm of this ratio in step 6 to indicate if a signal is tonal-like. SNR (Signal-to-Noise Ratio) is computed from the tonality *tb(b)* in step 7 and then the masking partitioned energy threshold nb(b) is estimated in steps 8-10 and thus the masking curve is estimated. PE (Perceptual Entropy) is calculated for each block type from the ratio e(b)/nb(b) in steps 11-12 to determine the block type. Finally, SMR (Signal-to-Mask Ratio) is computed in step 13 as the output. These SMRs are then sent to the bit allocation routine to determine the number of bits allocated to each subband.

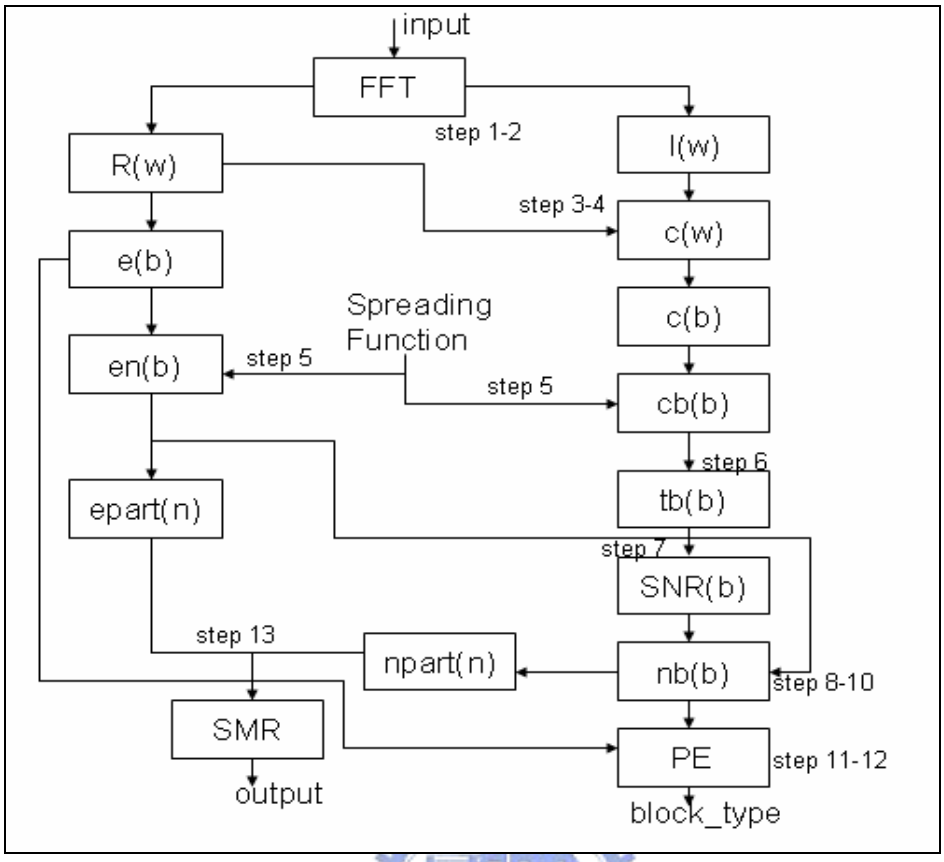


Fig 4.3 Block diagram of original PAM [9]

The step 2 and step 4 have high computational complexity because they include sophisticated mathematical functions. The step 5 includes spreading functions calculations and convolutions, so it also has high computational complexity. Next, the algorithm proposed by [9] that can reduce computational complexity of above mentioned steps will be described. It consists of two points:

■ Reduce calculations of spreading functions as fixed-coefficients

The calculations of spreading functions in step 5 are a series of complex functions such as comparisons, square roots, power of tens, squares, and divisions. They are calculated at the square number of partitioned bands and repeatedly estimated every frame. Besides, the spreading functions are only determined by sampling rate and block types. Therefore, we can reduce the calculations by replacing them with fixed-coefficients.

■ MDCT-based PAM

We know that there is one main filterbank MDCT outside the PAM and there is

another filterbank FFT inside the PAM transforming input samples into spectrums in similar ways. Therefore, we replace FFT by MDCT so that the FFT could be omitted in order to decrease computational complexity. Steps 2-4 are thus calculated on the MDCT, but step 5 requires some modification that only the partitioned energies are convolved with the spreading functions mentioned above because of the lack of phase information. Step 6 is also modified where Spectral Flatness Measure (SFM) [11] is used to generate the tonality index from the MDCT coefficients. The SFM is defined as the ratio of the geometric mean (Gm) of the power spectrum to the arithmetic mean (Am) of the power spectrum. Then, the SFM is converted to decibels. And the tonality index tb can be computed by this formula: $tb = min\ (SFM_{dB} / 60\ ,\ 1)$.

Finally, Fig 4.4 [9] is the block diagram of proposed PAM described in it. The steps 2-6 are using MDCT and SFM, and the spreading functions in step 5 are computed with fixed coefficients.

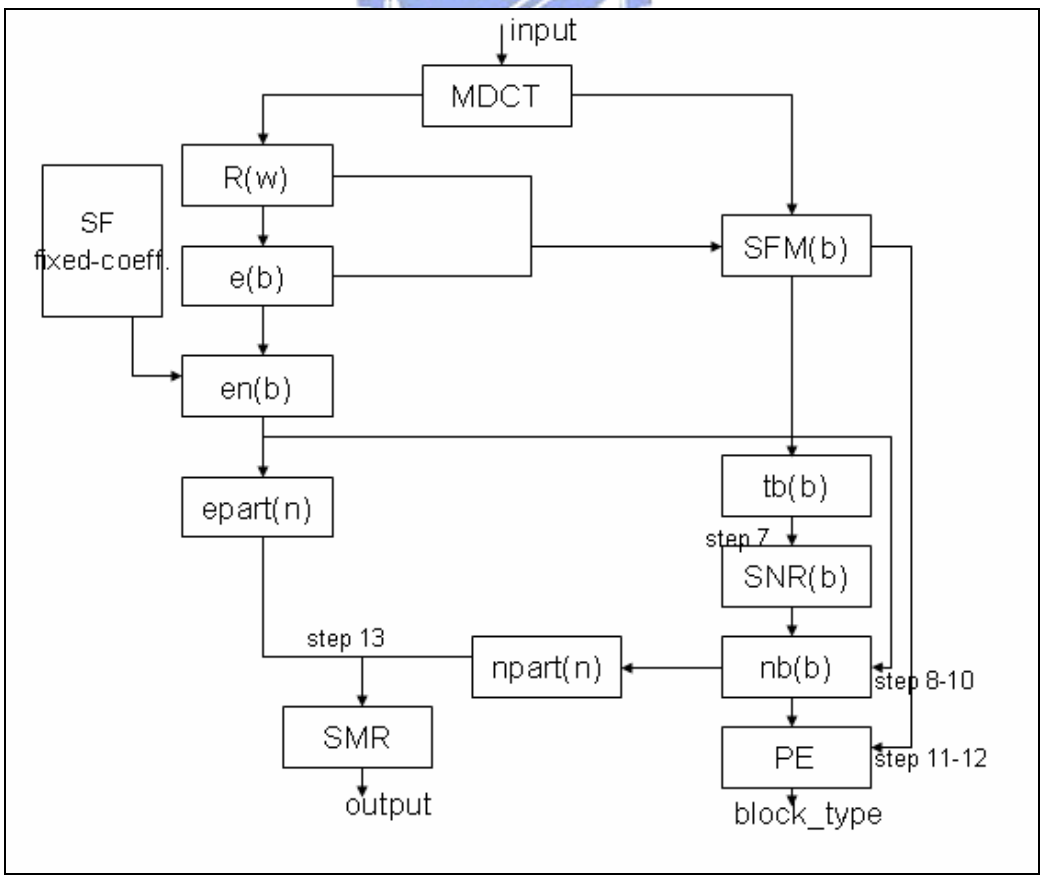


Fig 4.4 Block diagram of proposed PAM [9]

4.4.2 Simulation Results on DSP

In this section, we have simplified the psychoacoustic model by the algorithm in section 4.4.1 and implemented it on the TI C64x DSP. The fast algorithm result is shown in Table 4.6. Our test sequence is “guitar”, and the length is about 0.1 second. In the original AAC encoder program, the PsyInit function calculates the spreading function. The PsyBufferUpdate function contains the FFT calculation. And the PsyCalculate function does the masking threshold calculation. We can clearly see that the acceleration of the PAM is effective by this new algorithm. Also, we have done the sound quality test. Using the ITU-R BS.1387 PEAQ (perceptual evaluation of audio quality) defined ODG (objective difference grade), we examine some sequences using the fast algorithm. The ODG, which is a measure of quality, is calculated as the difference between the quality rating of the reference and the test signal. The quality ratings are measured with a range of [-4;0], where -4 stands for very annoying difference and 0 stands for imperceptible difference between the reference and the test signal. This parameter represents the audio quality well for good quality codecs.

The first test sequence is “guitar” and it has sound variations and is quite complex. The second test sequence is “organ” and it is another instrument music. But its sound is consecutive and delicate. The third test sequence is “eddie_rabbitt” and it is pop music with human voice. The test results are shown in Tables 4.7, 4.8, and 4.9. From these results, the quality test seems acceptable and the acceleration is good. The overall speed-up is around 80 percent, and the ODG drop is less than 0.3 or so.

Original	Code size	Execution cycles	Improvement (%)
PsyInit	7824	81,831,864	
PsyBufferUpdate	2312	21,822,065	
PsyCalculate	548	63,774,150	

Fast algorithm	Code size	Execution cycles	
PsyInit	5336	63,587,461	22.3
PsyBufferUpdate	432	24,004	99.89
PsyCalculate	408	10,630,861	83.33

Table 4.6 The acceleration result of the PAM in the AAC encoder

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.53	-3.37	-0.99	-0.38	-0.26	-0.01	-0.01
Modified	-3.68	-3.62	-1.22	-0.69	-0.57	-0.36	-0.28

Table 4.7 The ODG of test sequence “guitar”

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.89	-3.83	-2.76	-0.41	-0.03	-0.01	-0.01
Modified	-3.76	-3.67	-2.79	-0.29	-0.03	-0.00	-0.01

Table 4.8 The ODG of test sequence “organ”

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.78	-3.40	-0.87	-0.27	-0.11	-0.00	-0.00
Modified	-3.78	-3.79	-1.23	-0.62	-0.46	-0.21	-0.00

Table 4.9 The ODG of test sequence “eddie_rabbitt”

4.5 Quantization and Bit Allocation

The quantization and bit allocation module is essential in AAC. Its operation relies on the information from the psychoacoustic model that provides the best possible listening quality. And from Table 4.1, we can know that they have high computation load. Next,

we use efficient algorithms [12] [13] to replace the original models.

4.5.1 A High Quality Requantization Method

The requantization is used to calculate quantization error in the outer loop of the bit allocation module. The requantization is described by the following formula:

$$x\_invquant = Sign(x\_quant) \cdot |x\_quant|^{4/3}$$

The calucation of $X^{4/3}$ have high computation load, so the table lookup method is adopted. We adopt the high quality requantization algorithm in [12] to improve the speed of our system. The algorithm uses linear interpolation for each range of requantization and reduces the approximated error quite efficiently. With this approach, the codec maintains high quality result.

The X has a wide range, so using the whole range lookup table is not suitable. To reduce memory usage, this approach uses a 256-entry lookup table instead of the whole range table and its basic operation is shown in the following equation:

$$X^{4/3} = \left(\frac{X}{8} \times 8\right)^{4/3} = \left(\frac{X}{8}\right)^{4/3} \times 16$$

This means that a 256-entry lookup table, which stores the values of $X^{4/3}$ from X=1 to 256 respectively, can be used. And it uses a directly linear interpolation for the other two ranges of requantization. The request quantized values are 8191, and there are three ranges with its dedicated operation for the whole range as shown in Fig 4.5 [12]:

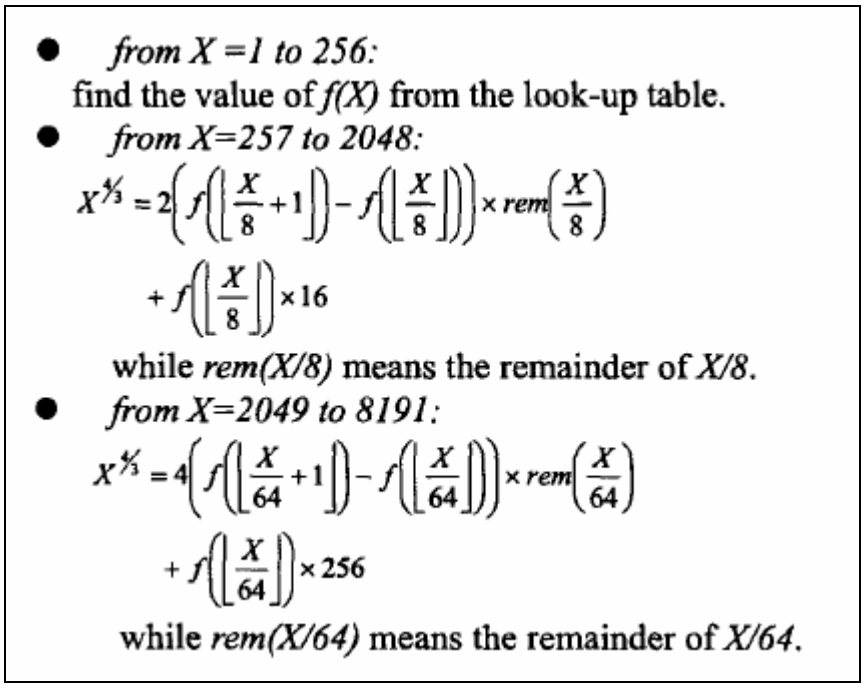


Fig 4.5 Requantization operation with three ranges [12]

4.5.2 Simulation Results on DSP

Using the above mentioned algorithm, the acceleration result of the requantization on the TI C64x DSP is shown in Table 4.10. Our test sequence is the same as above section. In program, the AACQuantizeInit function consists of the table calculation of $X^{4/3}$. From this simulation result, we can clearly see that this algorithm is quite efficient because the acceleration rate achieves 40.63 percent. Besides, the data precision is high enough so that it does not affect the accuracy loss.

AACQuantizeInit	Code size	Execution cycles	Improvement (%)
Original	912	246,364,788	
Fast algorithm	1572	146,274,298	40.627

Table 4.10 The acceleration result of the Requantization in the AAC encoder

4.5.3 Single Loop Distortion Control Algorithm

Firstly, we briefly the bit allocation algorithm adopted at AAC specification. Fig 4.6 [13] shows the bit allocation processes in MPEG-4 AAC encoder. The outer iteration loop controls the quantization noise, which comes from the quantization of the spectral signals within the inner iteration loop. The noise spectrum is computed by multiplying the values within the scalefactor bands with the actual scalefactors before quantization. After quantization, the calculation of the quantization noise is processed band-by-band iteratively. If the noise exceeds the specified threshold, the spectral values of the scalefactor bands are amplified by increasing the scalefactor by one.

In the inner loop, the encoder iteratively employs the nonuniform quantization, which has three major steps including the quantization of the spectral values, the calculation of actual number of bits using Huffman tables, and the computation of the resulting noise. By applying these three steps iteratively for each frame, the bit allocation algorithm provides the actual number of bits needed to encode the source. The quantizer uses the scalefactor and global gain to fit the requirement for the bit allocation, so we can estimate the quantization noise from the scalefactor and global gain.

And it adopts three extra conditions to stop the iterative process as follows:

1. None of the scalefactor bands has more than the allowed distortion.
2. The next iteration would cause the amplification for any of the bands to exceed the maximum allowed distortion value.
3. The next iteration would require all the scale factor bands to be amplified.

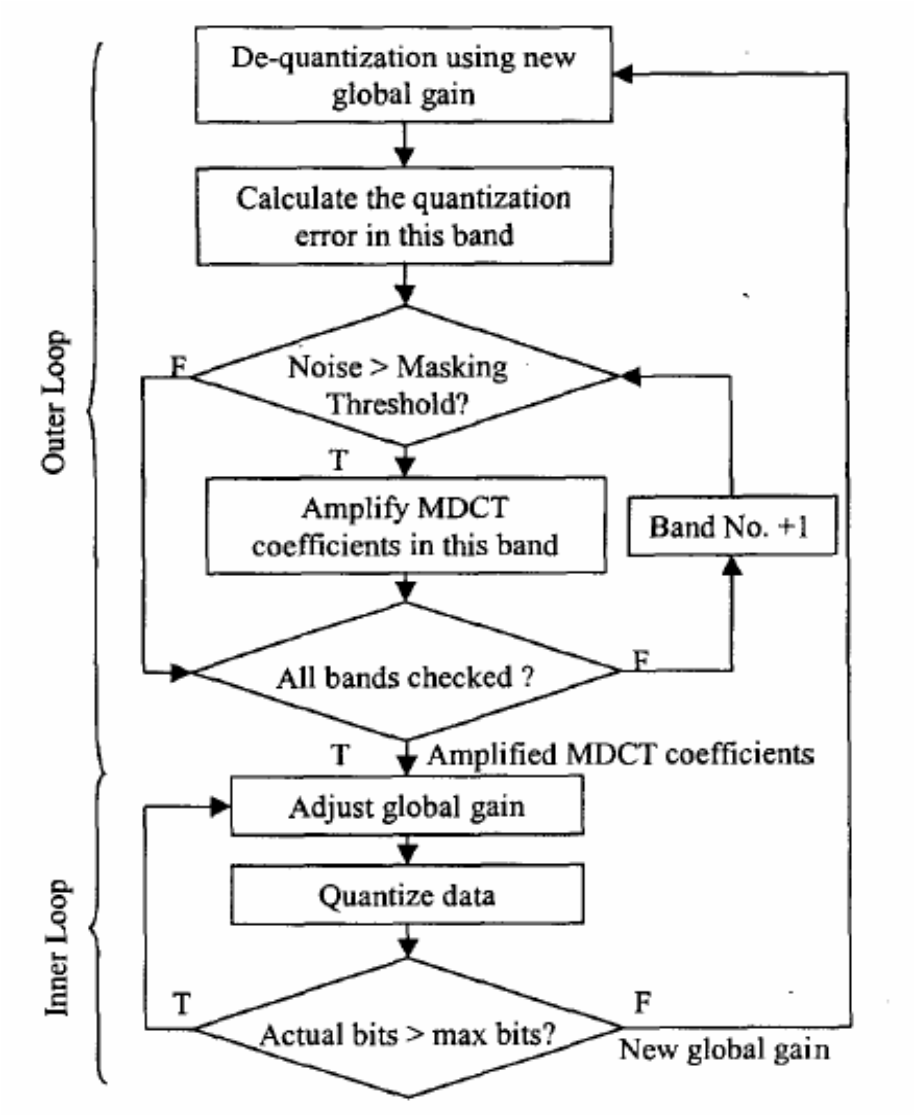


Fig 4.6 Block diagram of bit allocation [13]

Based on the proposed scheme in [13], we replace the iterative process but retain roughly the same listening quality. In the noise shaping scheme, the estimated noise is used to decide the value of scalefactors that can parallelize the quantization noise spectrum with the masking threshold. Hence the algorithm reduces the times of outer loop to one step and thus it provides a significant reduction of execution time. The quantization of the spectral signals $xr_i(j)$ and its approximation can be derived from

$$ix_i(j) = n\text{int}\left(\left(\frac{xr_i(j)\cdot scale_i}{2^{gain}}\right)^{0.75} - 0.0946\right) = \left(\left(xr_i(j)\right)^{0.75} + e_i(j)\right)\cdot\left(\frac{scale_i}{2^{gain}}\right)^{0.75}$$

Where $e_i(j)$ is the estimation error for the quantized spectral signals $ix_i(j)$. The dequantization of $ix_i(j)$ equals to

$$\bar{xr}_i(j) = sign(xr_i(j)) \cdot \left((xr_i(j))^{0.75} + e_i(j)\right)^{\frac{4}{3}}$$

The estimation error using the specified gain value can be measured by the mean square error (MSE) between $xr_i(j)$ and $\bar{xr}_i(j)$. Thus, the distortion function can be represented by

$$N(Noise)_i = E\left(\left(xr_i(j) - \bar{xr}_i(j)\right)^2\right) = \frac{16}{9} E\left(xr_i^{\frac{1}{2}}(j) \cdot e_i^2(j)\right) +$$

$$\frac{16}{27} E\left(xr_i^{\frac{-1}{4}}(j) \cdot e_i^3(j)\right) + \frac{4}{81} E\left(xr_i^{-1}(j) \cdot e_i^4(j)\right)$$

To reduce the order in approximation, it is assumed that and $xr_i(j)$) and $e_i(j)$ are independent and $e_i(j)$ is uniform distributed. So, we can approximately have:

$$N(Noise)_i \approx 0.1640578 \cdot E\left(xr_i^{\frac{1}{2}}(j)\right) \cdot \left(\frac{2^{gain}}{scale_i}\right)^{\frac{3}{2}}$$

In addition, the $N(Noise)_i$ can be derived from the psychoacoustic model by

$$N(Noise)_i = M_i \cdot NMR_i$$

So, we can calculate $scale_i$ by

$$scale_i = 2^{gain} \left(\frac{0.1640578 \cdot E(xr_i(j))}{M_i \cdot NMR_i}\right)^{\frac{2}{3}}$$

To determine the scalefactors, which are used to parallelize the estimated noise spectrum and the masking threshold energy, we let the $scale_i$ equal to unity at the band i. Thus, we can determine the other scalefactors in the remaining bands. After choosing the scalefactor in each band, we can quantize the spectral signals that are amplified by scalefactors $scale_i$ derived from the above equations. Fig 4.7 [13] shows the flow chart of this fast bit allocation algorithm.

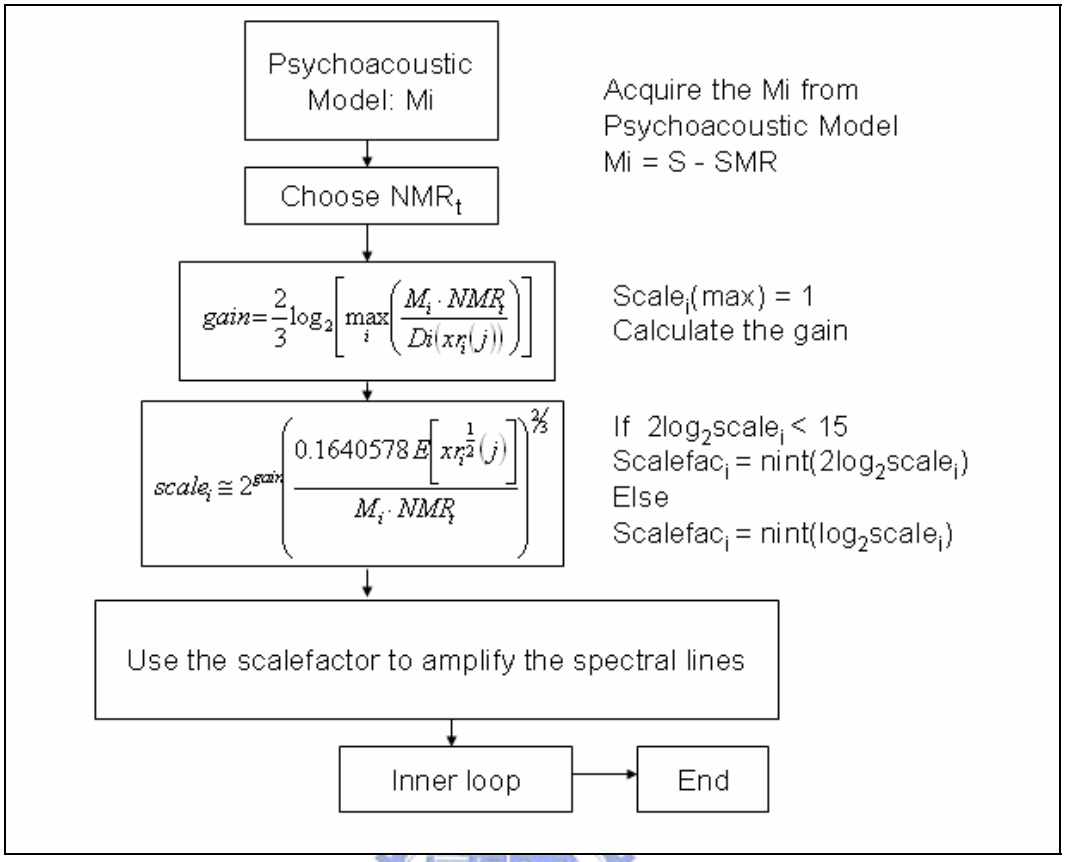


Fig 4.7 Flow chart of the bit allocation algorithm [13]

4.5.4 Simulation Results on DSP

In this section, we have accelerated the outer loop of bit allocation model by a fast algorithm and implemented it on the TI C64x DSP. The result is shown in Table 4.11. Our test sequence is also the same as the above. In the encoder program, the AACQuantize function contains the quantization and bit allocation operations. In addition, the outer loop and the inner loop are also in it. We can clearly see that the acceleration of the outer loop is efficient by this fast algorithm. Because we do not accelerate the inner loop, its clock cycles is not improved. Also, we did the sound quality test. Using the ODG (objective difference grade), we test some sequences on the modified algorithm. These test sequences are the same as the above tests. The test results are shown in Tables 4.12, 4.13, and 4.14. From these results, the outer loop algorithm is efficient because the improvement extremely achieves 92.36 percent. Besides, the sound quality is good as the original algorithm. Because it uses noise estimation to get the adequate scale factors such

that the summation of all NMR (Noise to Masking Ratio) values is the least at each frame. This criterion of the algorithm concerns about the result of psychoacoustics model and the noise is equally audible in different frequency bands.

Original	Code size	Clock cycles	Improvement (%)
AACQuantize	1452	51,841,384	
OuterLoop	1168	36,775,840	
InnerLoop	144	1,078,916	
Fast algorithm	Code size	Clock cycles	
AACQuantize	1448	17,897,864	65.48
OuterLoop	804	2,809,373	92.36
InnerLoop	144	912,279	

Table 4.11 The acceleration result of the bit allocation in the AAC encoder

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.53	-3.37	-0.99	-0.38	-0.26	-0.01	-0.01
Modified	-3.51	-3.14	-0.98	-0.33	-0.28	-0.01	-0.01

Table 4.12 The ODG of test sequence “guitar”

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.89	-3.83	-2.76	-0.41	-0.03	-0.01	-0.01
Modified	-3.89	-3.82	-2.59	-0.23	-0.03	-0.01	-0.01

Table 4.13 The ODG of test sequence “organ”

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.78	-3.40	-0.87	-0.27	-0.11	-0.00	-0.00
Modified	-3.62	-3.52	-0.65	-0.23	-0.12	-0.00	-0.00

Table 4.14 The ODG of test sequence “eddie_rabbitt”

4.6 The Final Simulation and Acceleration Results on TI C64x DSP

After accelerating codes and modifying algorithms, we have efficiently reduced the computation load of the encoder on DSP. Table 4.15 shows the final results. We can clearly see that after codes acceleration the performance improvement achieves 23.5 percent. But the improvement is not fast enough. And after algorithms modification, the final implementation speed is about 22.11 percent of the original execution time. Table 4.16 shows the profile of our final AAC encoder system on TI C64x DSP. We can see that the psychoacoustic model occupies only 22.75 percent in the final system. Table 4.17 shows the improvement of the major functions compared with original version shown in Table 4.1. After using fast algorithms, the speed increase of the psychoacoustic model, quantization, and bit allocation model is drastic compared with the original schemes. The quantization and bit allocation model achieves 73.52 percent and the psychoacoustic model even achieves a higher 89.09 percent than the original models. Also, we test final sound quality. The test sequences are the same as before and we include three additional sequences. The “TS_01” sequence is a piece of the instrument glockenspiel music. The “TS_02” sequence is the instrument guitar music. The “TS_03” sequence is the instrument tongue music. And the three test sequences are retrieved from the European Broadcasting Union (EBU). Tables 4.18 to 4.23 show the quality results. From these tables, the quality test results seem acceptable.

	Total Execution Cycles	Performance Improvement (%)
Original	2,126,810,017	
Code Acceleration	1,627,141,833	23.5
Final	470,273,769	77.89

Table 4.15 The final acceleration result of the AAC encoder

Function	Execution cycles	Percent (%)
Total	470,273,769	100
Psycho-acoustics	106,983,548	22.75
Filterbank	90,203,329	19.18
Quantization and Bit-allocation	264,896,754	56.33
Others	8,182,764	1.74

Table 4.16 Profile of final modified AAC encoder on C64x DSP

Function	Improvement (%)
Total	77.89
Psycho-acoustics	89.09
Filterbank	34.55
Quantization and Bit-allocation	73.52

Table 4.17 Improvement of each part in AAC encoder on C64x DSP

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.53	-3.37	-0.99	-0.38	-0.26	-0.01	-0.01
Modified	-3.65	-3.59	-1.12	-0.64	-0.57	-0.28	-0.25

Table 4.18 The ODG of test sequence “guitar”

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.89	-3.83	-2.76	-0.41	-0.03	-0.01	-0.01
Modified	-3.77	-3.56	-2.59	-0.30	-0.03	-0.01	-0.01

Table 4.19 The ODG of test sequence “organ”

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.78	-3.40	-0.87	-0.27	-0.11	-0.00	-0.00
Modified	-3.73	-3.75	-1.26	-0.65	-0.46	-0.21	-0.00

Table 4.20 The ODG of test sequence “eddie_rabbitt”

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.51	-3.80	-2.12	-0.99	-0.56	-0.14	0.01
Modified	-3.44	-3.70	-2.01	-0.80	-0.52	-0.12	0.01

Table 4.21 The ODG of test sequence “TS_01”

ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.79	-3.82	-2.03	-0.38	-0.15	-0.00	-0.00
Modified	-3.73	-3.80	-1.89	-0.45	-0.20	-0.00	-0.01

Table 4.22 The ODG of test sequence “TS_02”

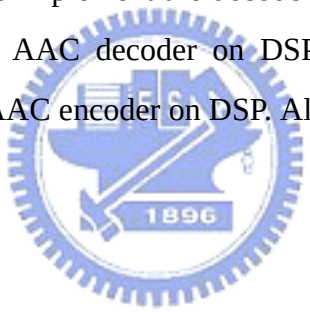
ODG	16 kbps	32 kbps	64 kbps	96 kbps	128 kbps	196 kbps	256 kbps
Original	-3.61	-3.32	-1.42	-0.49	-0.25	-0.01	-0.00
Modified	-3.30	-3.07	-1.15	-0.37	-0.10	-0.01	-0.01

Table 4.23 The ODG of test sequence “TS_03”

Chapter 5

MPEG-4 AAC Codec Implementation on DSP

In the previous chapter, we describe the acceleration of the MPEG-4 AAC encoder on DSP. In addition, we use some efficient algorithms which are derived from several papers to replace time-consuming models. In this chapter, we not only implement the MPEG-4 AAC encoder on DSP, but also implement the decoder on DSP. We will first describe the system structure of MPEG-4 AAC decoder on DSP. Secondly, we will describe the system structure of MPEG-4 AAC encoder on DSP. Also, we give experimental results of implementation at the end.



5.1 AAC Decoder Implementation on DSP

We implement the MPEG-4 AAC decoder on DSP by Quixote DSP board described in chapter 3. As mentioned, the Quixote have efficient hardware to implement our system. The software development environment CCS (Code Composer Studio) helps us in writing C/C++ codes. And we use Visual C++ as host program development environment. The transmission mechanism between PC and DSP adopts the burst block interface, which has been described in section 3.3, because this mechanism is relatively easy to implement as comparing to the data streaming mode.

5.1.1 Structure of AAC Decoder Implementation

We use the burst block transmission to implement our AAC decoder structure. Hence, we must create transmitting and receiving buffers at the host and target sides respectively. The use of the base buffer class allows integer, character, and float data types, but the receiving buffer at the host side must use the character data type. Our AAC decoder structure is shown in Fig 5.1.

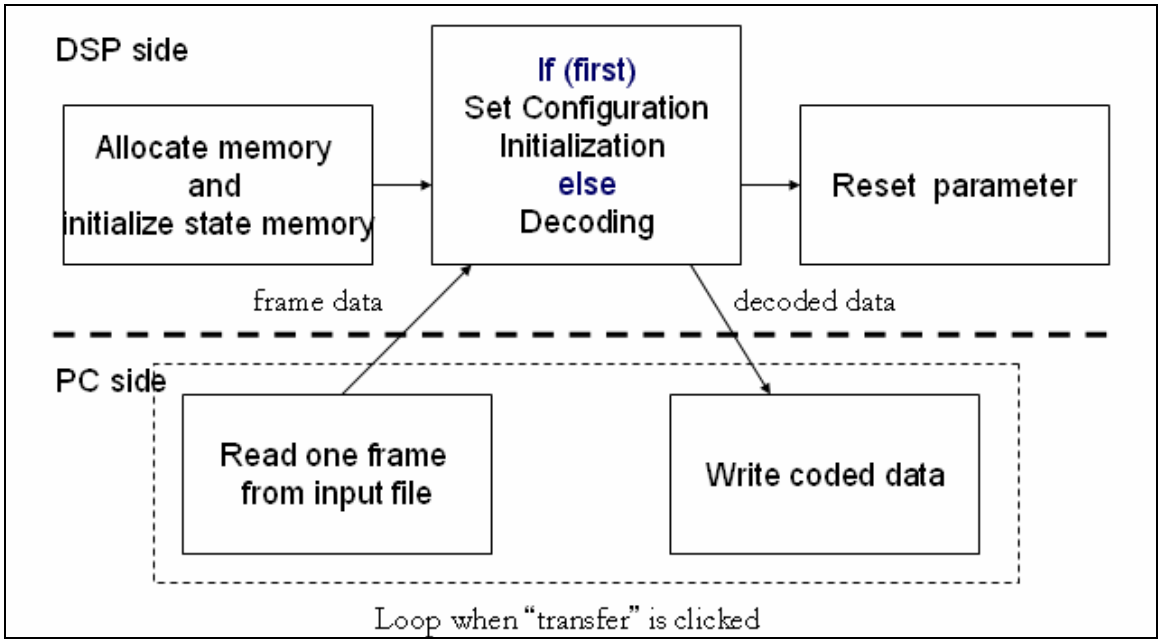


Fig. 5.1 Structure of AAC decoder implementation on DSP

The host side handles input file read and output file write, and allocates buffers to store data before and after processing. The transmitting buffer at the host side stores every frame data from the input file, and the receiving buffer stores decoded data from the target side. The target side stores the frame data from the host side, and then it does decoding process. After decoding, the decoded data stored in the buffer will be transmitted to the host side. And before running the decoding process at the target side, the DSP board performs some preprocessing and initialization work, including the memory allocation, state memory initialization and so on. In our program, there is a loop that does file read, file write, buffer transmission and buffer reception at the host side.

And at the target side, the DSP platform does the decoding one frame work, buffer transmission and buffer reception for every frame. But for the first time, the receiving buffer at the target side stores the initialization data instead of the decoding frame data. So, the DSP board will do default object type and samplerate setting, and the aac file format read. Also note that we must use the character buffer as the receiving buffer at the host side of our system as mentioned earlier. Therefore, we must convert character data into integer data from the receiving buffer at the host side and then write them into the output file.

5.1.2 Implementation Results of AAC Decoder

We have implemented the AAC decoder on DSP, and our test sequence is “guitar”. The sampling rate is 44.1 kHz. Table 5.1 shows the implementation result. We measure the average execution time of decoding one frame. And we have subtracted the transmission time between the host and the target sides. To measure the transmission time, we write a null function execution on DSP. And then the execution time on DSP is the differece between the total time and the transmission time. We can clearly see that the execution time on DSP is fast enough to achieve real-time operation. But the current setup requires the file read and write processing. They increase significantly the transmission time because we have a loop that processes the transmission between the host side and the target side for every frame.

	Time(s)
AAC decoder	1.7536e-4 s/frame

Table 5.1 Implementation result of AAC decoder on DSP

5.2 AAC Encoder Implementation on DSP

In this section, we will describe the MPEG-4 AAC encoder implementation on DSP. We also give some experimental results of implementation on DSP, including the compiler optimization.

5.2.1 Structure of AAC Encoder Implementation

Similar to the decoder, we also the adopt burst block transmission to implement the AAC encoder on DSP. The structure of the encoder is shown in Fig 5.2. The structure of the burst block transmission has been discussed in Section 5.1.

In this program structure, the host side does file read and write work, and the target side mainly does the encoding work. At first, the host side uses buffer to store one frame data and then transmit it to the target side. The target side will do board preprocessing job, including memory allocation and so on, until the buffer receives the frame data from the host side successfully. In addition, we put AAC encoder initialization on DSP, including opening the encoder library and configuring the options. The opening of the encoder library consists of default values initialization, default configuration, some coder functions initialization and so on. When the receiving buffer receives a frame data from the host side, the DSP board will encode input frame data. After encoding, the coded data will be stored in transmitting buffer to be sent into host side. And then, the host side will write the coded data from buffer to file. Every time the host side reads one frame data from input file, so there is a processing loop that finally completes the encoding task.

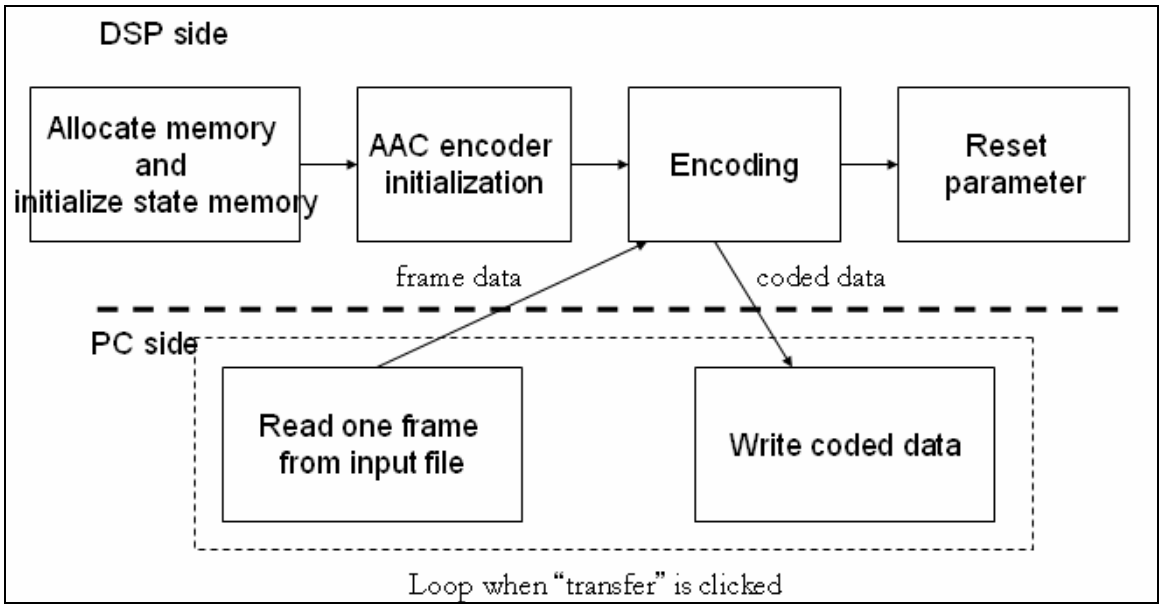


Fig. 5.2 Structure of AAC encoder implementation on DSP

5.2.2 Implementation Results of AAC Encoder

We have implemented the AAC encoder on DSP, and our test sequence is "guitar". The sampling rate is 44.1 kHz. Table 5.2 shows the implementation result. We measure the average computation time of encoding one frame. And we have subtracted the transmission time between the host and the target sides. From this table, we can see that the original computation time is 0.1742 second per frame. This value is not fast enough to achieve real-time operation. Therefore, we use some code acceleration techniques and algorithms modification have been described in chapter 4 to accelerate the AAC encoder system on this Quixote board. These measured values are shown in Table 5.2. And our final implementation time is 0.008 second per frame. This value is fast and acceptable. But the transmission time is not included.

Time (s)	Without open opt. level	Open opt. level (file level)	Code Acceleration	Code Acceleration with PAM	Code Acceleration with bit allocation
AAC encoder	0.1742 s/frame	0.13925 s/frame	0.08724 s/frame	0.0539 s/frame	0.0474 s/frame

Time (s)	Final implementation result
AAC encoder	0.008 s/frame

Table 5.2 Implementation result of AAC encoder on DSP

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The main goal of this work is to accelerate the MPEG-4 AAC encoder implemented on the TI C64x DSP processor. Our acceleration methods include the coding style modification to match the DSP hardware architecture and adopt several fast algorithms. Based on the profiling data, the psychoacoustic module and the bit-allocation module are the two heavy-load computational parts in the AAC encoder. For the psychoacoustic model, we reduce the calculation of spreading functions by using the fixed-coefficients and eliminate the original FFT calculation by using the MDCT-based spectrum. For quantization, we use the lookup table and linear interpolation method to accelerate it. And in the outer loop of the bit-allocation module, the noise estimation algorithm can reduce the iteration of outer loop to once and thus provides a significant reduction of execution time. The details and results can be found in chapter 4. The total performance has 77.89 percent improvement compared to the original program.

Furthermore, we have successfully implemented both the encoder and decoder of MPEG-4 AAC on the DSP platform. Our communication interface between the host and the target is the burst block transmission due to its simple control and easy implementation. With our acceleration, the execution speed of both encoder and decoder on the DSP platform is fast enough to achieve real-time operation. The implementation of the AAC encoder is about 21.78 times faster than the original version. The details and results can be found in chapter 5.

6.2 Future Work

If we can reduce the transmission time between the host and the target, our system will run faster. Hence, the transmission time reduction should be studied. At the moments, we transmit one frame data every time to the DSP side. We may transmit serial frame data once to reduce the number of transmission, but at the cost of delay and memory.

Also, the board provides us with the FPGA. We can integrate the FPGA implementation together with DSP to accelerate the overall system. But the transmission between DSP and FPGA is more complex, and we are unable to use it yet.

In addition, we do not implement some optional tools of the MPEG-4 AAC encoder on the DSP platform because we mainly focus on the speed of overall system. And the AAC encoder can be further accelerated by other optimization techniques.

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] T. Painter and A. Spanias, “Perceptual Coding of Digital Audio”, Pro. of the IEEE, Vol. 88, Issue 4, pp. 451-515, Apr. 2000

[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, “TMS320C64x Technical Overview”, SPRU395B, Jan. 2001

[9] T. H. Tsai, S. W. Huang and L. G. Chen, “Design of a low power psycho-acoustic model co-processor for MPEG-2/4 AAC LC stereo encoder”, IEEE Int. Symp. on Circuits and Systems, Vol. 2, pp. 552-555, 25-28 May 2003

[10] Draft revision of recommendation ITU-R BS.1387, “Method for objective measurements of perceived audio quality”, 1998

[11] J. D. Johnston, “Transform Coding of Audio Signals Using Perceptual Noise Criteria”, IEEE Journal on Selected Area on Communications, Vol. 6, No 2, Feb.

1988

[12] T. H. Tsai and C. C. Yen, “A high quality re-quantization/quantization method for MP3 and MPEG-4 AAC audio coding”, IEEE Int. Symp. on Circuits and Systems, Vol. 3, pp.851-854, 26-29 May 2002

[13] C. Y. Lee and et al., “A fast audio bit allocation technique based on a linear R-D model”, IEEE Trans. on Consumer Electronics, Vol. 48, pp. 662-670, Aug. 2002

[14] C. M. Liu and et al., “A fast bit allocation method for MPEG layer III”, in Proc. of ICCE, pp. 22-23, 1999

[15] H. Purnhagen, “An Overview of MPEG-4 Version 2 Audio”, AES 17th International Conference on High Quality Audio Coding, Sep. 1999

[16] B. Grill, “MPEG-4 audio: A preview into the technology of the future”, 108th Convention of AES, Feb. 2000

[17] T. Nomura, Y. Takamizawa, “Processor-Efficient Implementation of a High Quality MPEG-2 AAC Encoder”, presented at AES 110th Convention, May 2001

作者簡歷

王盈閔，民國七十年出生於嘉義市。民國九十二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系統組就讀，從事多媒體訊號處理系統設計與實現之相關研究。民國九十四年六月以論文題目為『MPEG-4 先進音訊編解碼器之增速及其在 DSP 平台上的實現』取得碩士學位。研究範圍與興趣包括：多媒體訊號處理、訊源編碼、軟硬體整合實現與最佳化。


E
S
A
1896


E
S
A
1896


E
S
A
1896


E
S
A
1896