Fast Implementation of Integer Transforms in 4X4 H.264/AVC Video Encoders

Smart Charles Lubobya

Supervisors : Professor M.E. Dlodlo.

: Professor G. de Jager

Co-supervisor : Dr K. L.Ferguson

This thesis is submitted in fulfillment of the academic requirements

for the degree of

Master of Science in Engineering

In the Faculty of Engineering and the Built Environment

University of Cape Town

2011

As the candidate's supervisors, we have approved this dissertation for submission

1. Name: Prof. Mqhele E.Dlodlo, UCT Associate Professor

Signed:_____

Date:_____

2. Name: Prof. Gerhard de Jager, UCT Emeritus Professor

Signed:_____

Date:_____

DECLARATION

I declare that this Research is my own work. Where collaborations with other people have taken place or materials generated by other researchers are included, the parties and/ or materials are explicitly stated with references as appropriate.

This work is being submitted for the Master of Science in Electrical Engineering at the University of Cape Town. It has not been submitted to any other university for any other degree or examination.

Smart Charles Lubobya	**01st August, 2011**
Name	**Date**

Dedication

To my wife Lydia; my three sons Timothy, Titus and Ted, I say thank you for your support and inspiration. Above all may Glory be to God, the giver of life.

Abstract

The Integer Discrete Cosine Transform (IDCT) and Hadamard transforms are adopted in the H.264/AVC standard encoder for the compression of residual video signals. Other video standards such as the H.261, H.262 and H.263 use Discrete Cosine Transform (DCT). The Integer DCT is used to transform the video signal from the spatial components to the equivalent spatial frequency components [AC decomposition]. Hadamard transforms (Luminance and Chrominance data) on the other hand are used to transform the DC component of the video signal. Video compression is important considering the limited storage environment and transmission bandwidth that characterise most networks. However, the video compression processes must be done within minimum time to meet the need of real-time applications and this is the hallmark of this research.

Experimental work was done on the Fast Implementation of 4X4 integer DCT and 4x4 Hadamard transforms for the H.264/AVC video encoder in software. We used the C++ programming language, Multi-Media eXtension (MMX) technology Single Input Multiple Data (SIMD) instructions and inline assembly programming tools. In the first set of experiments, the forward and inverse 4X4 Integer DCT transforms were implemented in software using 2-D Matrix Multiplication or 1-D butterfly methods. Results obtained for this transform type; show that 1-D implementation has a fast average computation time speed-up of 40% to 45% with reference to the 2-D matrix multiplication. In the second set of experiments, the 4x4 Hadamard transform was also implemented in software as 2-D matrix multiplication or as 1-D Hadamard butterfly algorithm on the horizontal followed by the vertical direction. Results on the Hadamard transform show a computation time speed-up of 41% to 47% in favour of the butterfly method.

At such computation time speed-up 1-D butterfly implementations are suitable for application in low bandwidth environment, real-time applications and /or internet broadcasting.

Acknowledgements

My sincere appreciation goes to my supervisors A/Prof. M. E. Dlodlo, Prof. G. De Jager and Dr K. L Ferguson for their guidance, support and encouragement during the course of my study at the University of Cape Town. My gratitude also goes to Ralf Globisch, for his warm help in my research work.

Sincere gratitude also goes to the Council for Scientific and Industrial Research (CSIR) for the financial support and encouragements. The prayers of my family and the church have been a source of comfort.

I also appreciate the critique and advice from the members of the Communication Research Group; Mthulisi Velempini, Guy-Alian Lusilao-Zodi, Bonginkosi Nhleko, Rohini Koduri and Josephine Nakato Kakande for reviewing my work.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ACCRONYMS

AMD : Advanced Micro Devices

ASO : Arbitrary Slice Order

AVC : Advanced Video Coding

Bpp : Bits per pixel

CABAC : Context Adaptive Binary Arithmetic Coding

CAVLC : Context-based Adaptive Variable Length Coding

CIF : Common Intermediate Format

CODEC : Encoder and Decoder

CPU : Central Processing Unit

DC : Direct Current

DCT : Discrete Cosine Transform

DVD : Digital Versatile Disk

FMO : Flexible Macroblock Order

HDTV : High Definition Television.

H.264 : A video coding standard

IEC : International Electrotechnical Commission

INTEL : **Int**egrated **El**ectronics Corporation

ITU-T : International telecommunication union-telecom

ISO : International Standard Organisation

JPEG : Joint Picture Expert Group

JVT : Joint Video Team

MPEG : Moving Picture Expert Group

MMX : Multi -Media Technology

OOP : Object Oriented Programming

PSNR : Peak Signal to Noise Ratio

QCIF : Quad Common Intermediate Format

SD : Standard Definition

SIMD : Single Instruction Multiple Data

SSE : Streaming SIMD extension

VCEG : Video Coding Experts Group

VLC : Variable Length Codes

TV : Television

LIST OF PUBLICATIONS

The under listed conference papers have resulted from this thesis and have been accepted for publications.

1. C.S.Lubobya, M.E.Dlodlo, G.de Jagar, "Fast Implementation of Integer Transforms in H.264/AVC Video Encoders" in *proceedings of the 13th southern African telecommunication network and applications conference (SATNAC),* Sept. 2010.

2. C.S.Lubobya, M.E.Dlodlo, G.de Jagar, "Optimisation of 4x4 integer DCT in H.264/AVC video encoders" in *proceedings of the 14th southern African telecommunication network and applications conference (SATNAC),* Sept. 2011.

3. C.S.Lubobya, M.E.Dlodlo, G.de Jagar, "SIMD implementation of integer DCT and Hadamard transforms in H.264/AVC encoders" in *proceedings of the IEEE African Conference (AFRICON),* Sept. 2011.

Glossary

Bit - smallest piece of computer information 0 or 1

Bit depth - the amount of data that can be accessed, for instance MMX can be accessed using 64 bit.

Byte – an array of eight bits

Data type- Specifically defined Software identification category

Double-word - an array of thirty-two bits

Pixel - a pixel is a smallest viewable area on a computer monitor

Quad-word – an array of sixty-four bits

Register - memory spaces inside the microprocessor used as storage

Saturation arithmetic – similar to unsigned arithmetic, however any carry or overflow-bit is ignored and the maximum value is accepted in such cases

Word - an array of sixteen bits

Chapter One

1.0 Introduction

The newest video compression standard H.264/AVC has been used in a wide range of applications from low-bit rate, low-delay mobile transmission through high definition consumer television (TV) to professional television production [1] [2] [3]. The H.264/AVC standard adopts four profiles: the Main, Baseline, Extended and High profiles.

In each of these profiles Integer Discrete Cosine Transforms (DCT) instead of the conventional 8x8 DCT, are used to transform the signal from the spatial domain to the spatial frequency domain [3]. The image or video signal is ideally changed from the image or video characteristic to the frequency or AC characteristic (AC decomposition). Studies of the human visual system indicate that the human eye is less sensitive to high frequency signals than the low frequency ones; the former can be removed via the quantisation process without significantly affecting the quality of the video [1] [2] [4]. Furthermore, like any transform, the decomposed or transformed signal consists of the DC components which in this case represents the luminance and chrominance components. These components are also decomposed using Hadamard transforms. We intend to show that both the integer DCT and Hadamard transforms can be implemented or optimised with minimum time for real-time applications.

Principally, the two transforms reduce redundancies in the video or image, which is critical and essential for video storage and transmission in a given bandwidth. Figure 1.1 illustrates the video compression, storage and transmission system under various video compression standards. [1] [2] argues that a Digital Versatile Disk (DVD), for example, is only capable of storing a couple of seconds of uncompressed video at television frame rate and quality resolution. Practically that means that DVD storage requires video and audio compression [1]. Video compression allows efficient use of the transmission bandwidth or resource. Additionally, where a high bandwidth or bitrate exists, it is more economical to send one compressed or several compressed videos than uncompressed and low resolution video signals [1] [2] [4].

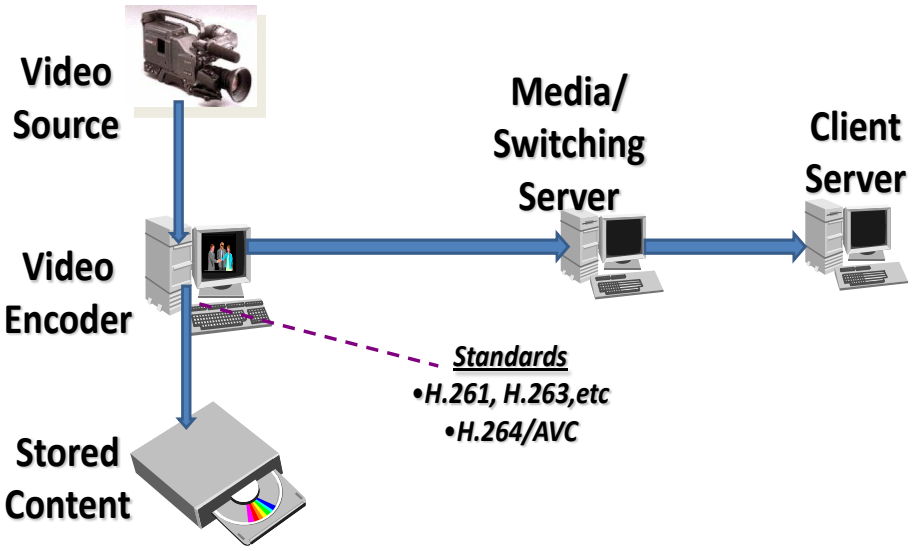


Figure 1.1: Video Compression, Storage and Transmission System under Various Video Compression Standards.

There are two categories of video compression schemes, Lossless and Lossy Compression. In Lossless Compression schemes, the compressed video and/or image after reconstruction is the exact copy of the original. It is essentially, digitally or statistically, identical to the original video. However lossless compression can only achieve a modest amount of compression [1] [4] [5] and is seldom used in most practical encoders. Lossless compression is, however, preferred in medical imaging and similar applications where quality is emphasised. On the other hand, Lossy Compression scheme, does not give the exact copy of the reconstructed image or video after the compression process. The video or image quality is reduced depending on compression settings. This is because certain correlated portions of the image or video that are not very noticeable to the human visual system are removed. In applications where storage space and bandwidth concerns are limited or video fidelity is less emphasised, Lossy Compression scheme is the choice solution. We will adopt the lossy compression scheme in the implementation of integer DCT and Hadamard

transform considering the storage and bandwidth limitations in addition to the speed-up concerns.

1.1 Background on the Integer Transforms

The earlier standards (see detailed description and application in Appendix A) such as H.261, H.262, H.263, MPEG-1, MPEG-4 and JPEG have been using the conventional DCT as transform coding for converting the video signal from the space domain into the frequency domain [1]-[17]. The JPEG 2000 however, uses the Discrete Wavelet Transform (DWT) [1] [2]. The DCT is a block-based transform with the advantage that it has low memory requirements and fits in well in block based motion estimation [1]-[10]. It operates on an 8x8 block size[2] in which the input space video is captured into 8x8 pixels on a 16x16 macroblock. The signal is later decomposed into AC cosine component terms and DC components representing the luninance and chrominace terms [3]. The cosine terms when evaluated reduce to floating points making the DCT non orthogonal [1] [2]. Thus the DCT requires a complicated multiplication process to convert the video signal into the frequency domain. Consequently, this leads to mismatch problems during the inverse process [1] [2] [18]. It also tends to suffer from artefacts at the block edges [1] [7] [8].

The integer DCT is essentially an approximation to the conventional DCT [2]. The cosine terms are approximated into integer values to avoid floating points and this makes integer DCT orthogonal [2] [18]. Therefore, the complicated matrix multiplication can be avoided by using simple adds and shifts [1] [2] [5] [18]. Unlike the conventional DCT, integer DCT uses various block sizes such as 4×4 or 8×8[5]. However, use of the integer DCT transforms comes at the expense of quality degradation in comparison with the conventional DCT. Another important integer transform type is the Hadamard transform. It is named after the French mathematician Jacques Hadamard and is also called Walsh-Hadamard or Walsh-Rademacher-Hadamard or Walsh-Fourier transform [19]. It is one example of the general classes of Fourier transforms [19]. This transform is applied on the DC component after integer DCT. It is performed on 2^n real or complex numbers, though the transform is itself purely real, as symmetrical, orthogonal and linear operation [19].

1.2 Problem Definition

There have been a lot of studies to reduce complexities of the integer DCT. Most of it has been hardware oriented to suit different block sizes defined in the four H.264/AVC profiles. This, however, does not provide flexibility in the integer DCT algorithm. One solution, explored in this work, is to provide hand codes for the integer DCT algorithm using loop optimization with high level languages like C++ or low level assembly language coding. This solution has the weakness that only the memory access part will be optimized while the algorithm part on which additions, subtractions and shifts occurs may still not be optimized. Notwithstanding, we anticipate some performance and speed-up gains.

Furthermore, the Intel IA-32, Advance Micro Devices (AMD)-64 and AltiVec technology power Personal Computers (PC) processors, have Single Input Multiple Data (SIMD) extensions. The SIMD extensions [MMX, Streaming SIMD Extension (SSE), SSE2 etc] along with the hand code optimization can be used to reduce memory access and parallel manipulation of pixels. With SIMD, processing and manipulation of pixels can be done at the same time thereby speeding up the video encoder, ensuring its use in real-time applications and /or low bandwidth environments.

1.3 Related Work on Forward and Inverse Integer Transforms

Research on implementation of the integer DCT and Hadamard transforms, from both the software and hardware perspectives, has been done but never exhausted. Most of this research is aimed at minimising the complex multiplications and optimising additions and shift operations as a solution. *Richardson* states that the original proposal for the 4x4 H.264/AVC forward and inverse transform processes describe alternate implementation methods using either a series of add and shift, a flow graph algorithm or matrix multiplication [1]. Due to the complexities of the matrix multiplication methods, the focus has been to explore other algorithms while achieving the same compression quality.

Malvar et al., suggests that we can treat two-dimension (2-D) 4X4 integer DCT transforms as row-column application of one-dimension (1-D) methods, that is, separate implementation of the 1-D four dot integer in each of the rows and columns [8]. This method only uses additions, subtractions and shifts operations, and is therefore less complex both for hardware and software implementation.

Fan introduced an algorithm based on sparse factorisations of the 4x4 matrix and reduced, the computational complexity by having 64 additions and 16 shifts [15]. *Ji Xiuhua et al.,* performed study on fast algorithms for inverse transform, and managed to reduce the additions to 12.7838 and shifts to 1.6954. In their experiment, they removed all the all-zero blocks and then did the inverse integer transform on the nonzero 4x4 blocks [20]. *Mohammed et al.,* compared the performance of 4x4 Integer DCT with the conventional 8x8 DCT on the basis of computational time and Peak Signal To Noise Ratio (PSNR). The 4X4 integer DCT was 0.30ms times faster than the conventional 8x8 DCT [21].

An optimized stream algorithm for integer transform in H.264 named Interleaved Streaming Transform (IST) was suggested by *Haiyan Li et al.,* in which a 4x4 integer transform was used to transform a CIF image [22]. *Wang et al.,* extended and adapted the concept of detection of all-zero quantization coefficients before transformation and quantization in H.264/AVC [23].

Integer DCT and Hadamard transforms can also be implemented using Single Input Multiple Data (SIMD) instruction sets to achieve matrix multiplications, additions, subtractions, shifts, etc. Special libraries of instruction in what is known as Streaming SIMD Extensions (SSE), SSE 2 and MMX technology are available to optimise the integer transforms. *Chen et al.,* designed a SIMD matrix multiplication scheme for both integer and Hadamard transforms. They also analysed different multi-threading schemes that have different quality/performance and proposed a scheme with good scalability and quality [24]. *Jianhong et al.,* suggested a fast and efficient parallel implementation of H.264 Integer transforms using SIMD instructions, based on the 8x8 block size [25]. *Xueming et al.,* outlined a method of using SIMD instructions for the 4x4 block size and applied the method to the ***forward*** integer DCT and Hadamard transforms. They achieved 60% and 127% computational speed up for the integer DCT and Hadamard transforms respectively [26]. This work

implements the forward and inverse integer DCT and Hadamard transforms using C++, Assembly language and SIMD instructions. Special focus is on speed-up optimisation while keeping the PSNR constant.

1.4 Aims and Objectives

In this research we look at fast implementation of 4X4 integer DCT transforms and Hadamard transforms using the 2-D Matrix Multiplication and 1-D Butterfly algorithm methods in software. Our set objectives were as follows:

1) To implement the forward and inverse integer DCT using 2-D Matrix multiplication and 1-D Butterfly methods in software.
2) To implement the forward and inverse Hadamard transforms using 2-D Matrix multiplication and 1-D Butterfly methods in software.
3) To measure and compute the average encode/decode time (computation time) in each of the methods in 1) and 2) and using the 2-D matrix multiplication method as a reference, ascertain speed up gains for adopting the various 1-D butterfly algorithms.
4) Based on the results obtained in each method, recommend appropriate applications.
5) Measure and ascertain the quality of the compressed video signal.

1.5 Research Hypothesis

The 2-D 4x4 integer DCT and Hadamard transforms implemented in a separable way as 1-D add, subtract and shift operations either in the horizontal followed by the vertical direction or vice versa is computationally faster than the direct 2-D matrix multiplication.

This hypothesis is stated due to the several advantages that 1-D implementation has:

- efficient
- less complex
- easy implementation in both software and hardware

1.6 Research Questions

When discussing related work in section 1.3, we alluded to the fact that the original H.264/AVC standard suggests three alternate implementation algorithms for integer transforms; shift and add, flow graph and matrix multiplication. This, related to our research objectives, leads to the following cardinal questions:

1. Why is one method better than another?
2. Which method is the fastest and is the most accurate?
3. Do any of the methods sacrifice accuracy for speed?
4. How much faster can the methods be done using SIMD instructions?

1.7 Scope of the Project

This research covers video compression and transforms coding techniques in video encoders as outlined in the H.264 and / or MPEG-4 visual part 10. It is limited to:

- Integer DCT and Hadamard transforms
- 4X4 block size.
- Baseline profile
- C++, SIMD MMX technology, inline assembly programming environments.
- Word length of 8 bits for a pixel value

The experiments were based on a windows platform with visual studio 2008 programming environment.

1.8 Thesis Outline

Chapter One has introduced the subject area of this research stating the background of video compression standards and integer DCT. Related work on integer DCT and Hadamard transforms has been outlined. Research objectives, hypothesis and scope have also been given.

Chapter Two discusses an overview of the H.264/AVC video compression standards in which integer DCT are specified. A description of H.264/AVC features, structure, profiles and encoder showing clearly the position of integer transforms is given

Chapter Three derives the mathematical models behind the integer transform and quantisation processes used in the H.264/AVC encoder. These models were then used to implement the software model during experiments suggested in chapter five (5). The models derived are for the three (3) transforms used in the baseline profile of the H.264/AVC.

Chapter Four describes the various SIMD technologies used to exploit data level parallelism. These technologies are discussed to ascertain their application in optimising integer transforms during experiments.

Chapter Five outlines the methods adopted in our experiments. The chapter also gives the experimental tools and measurement parameters. This chapter shows how the research questions will be answered.

Chapter Six gives the results and analysis obtained during experiments.

Chapter Seven summarises the thesis with conclusion and recommendations for future works.

1.9 Chapter summary

This chapter has given the introduction to software implementation of integer DCT and Hadamard transforms. The various standards and transforms discussed in the background section have shown how discrete cosine transforms have been used in contrast with the integer DCT used in the current H.264/AVC standard. In our problem definition of this research we have shown how memory access and optimisation can be speeded up using various software programmes and languages. The research objectives, scope and hypothesis statement has also been given.

Chapter Two

2.0 An Overview of the H.264/AVC Standard

2.1 Chapter Introduction

In this Chapter we give the principles adopted in video compression and a description of the H.264/AVC video encoder. The video encoder functional modules and the position of integer transforms are describes and shown respectively. The structure of the H.264/AVC showing the macroblocks, slice, video colour format, profiles and levels is also given.

2.2 Principles Behind video Compression in H.264/AVC

2.2.1 Redundancy Reductions

Video and/or image compression is accomplished by redundancy reduction. In redundancy reduction the basic idea is to remove any duplication in the input video or image. Figure 2.1 shows the various models and modules used to reduce redundancy in both lossy and lossless compression systems. In general, redundancy can be categorised as statistical (for lossless compression) or subjective (for lossy compression). It can also be classified based on the redundancy model or module. Thus we have spatial redundancy (from the spatial model), temporal redundancy (from the temporal model) and statistical redundancy from the entropy encoder module.

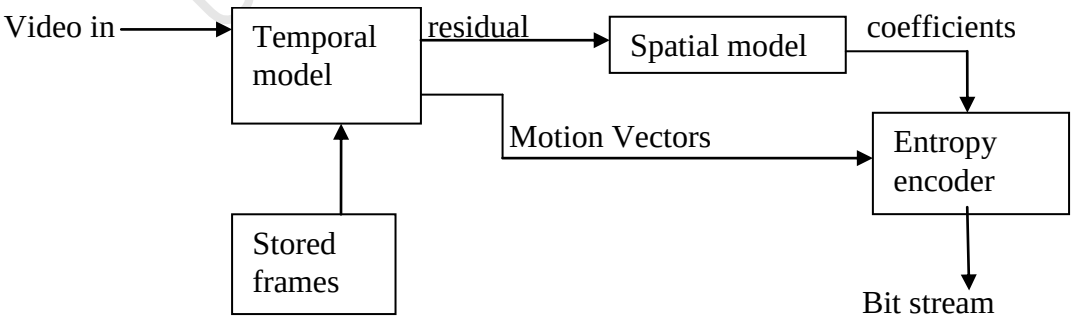


Figure 2.1: Block Diagram Illustrating Principles of Video Compression [1]

- Temporal redundancy – this is achieved by reducing correlation or similarities between adjacent video frames in time. This is achieved by constructing a prediction of a future frame and subtracting it from the current video frame [1] [18].
- Spatial redundancy- a spatial model is used to achieve spatial redundancy. The model takes the input residual frame from the temporal model and discards any correlation or similarities between neighboring pixels not detected by the human visual system. The output is a set of coefficients, consisting of high and low frequency video signal. Frequencies with small amplitudes are removed via the quantization process.
- Statistical redundancy - this is achieved by using an entropy encoder module. The module compresses the motion vectors and coefficients parameters of the temporal and spatial models respectively, thereby reducing statistical redundancy [1] [18]. A compressed bit stream can then be transmitted or stored.
- Spectral redundancy – this is correlation reduction obtained among different color planes.

2.2.2 Irrelevancy Reduction

Certain parts of the video or image signal tend not to be noticed by the signal observer. These can be considered as irrelevant and can therefore be omitted.

2.3 H.264/AVC Video Encoder

The H.264/AVC video encoder shares the same functional modules as those used in previous encoders in the H.261, H.263, MPEG-4 part 2, MPEG-2 and MPEG-1, standards [1]. The module details were redesigned and changed in order to offer better performance in coding algorithms and to include features such as [1] [11] [18]:

- Intra-picture prediction
- A new 4x4 integer transform
- Multiple reference pictures
- Variable block sizes and a quarter pixel precision for motion compensation
- A de-blocking filter and
- Improved entropy coding

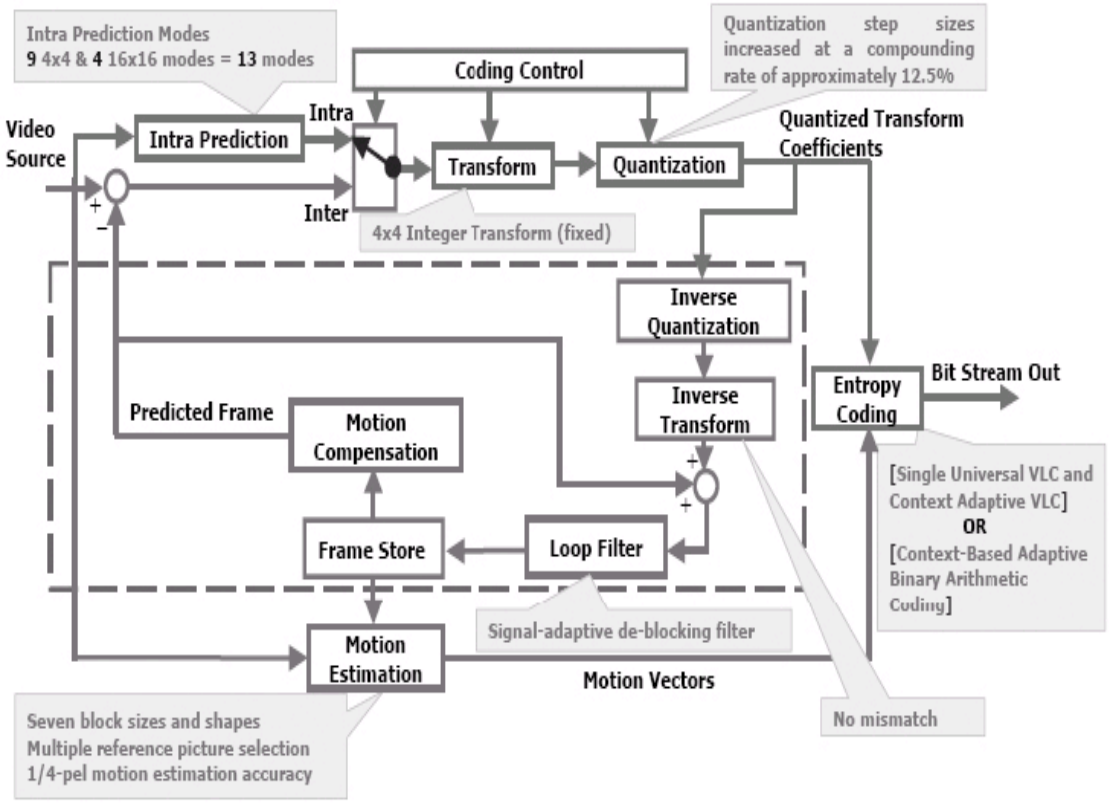


Figure 2.2: The H.264/AVC Video Encoder [27] [28]

Figure 2.2 shows the block diagram of a video encoder. It consists of various modules: Transform, Quantisation, Entropy Encoder, Motion Estimation and Compensation, Prediction, De-Blocking Filters, etc, which are used collectively to compress the video signal and therefore reduces redundancies. The main modules are discussed in the following sections with a special focus on the integer and quantisation modules - the basis of this thesis. Further, details are outlined in chapters two, three and six of [1].

2.3.1 Motion Estimation

The Motion Estimation (ME) module essentially identifies or detects and then removes any redundancies present between frames (ie temporal redundancies). In this module motion vectors are calculated between the current and future frames. This is achived by several algorithms and consequently these make it the most computationally intensive module during video compression [29]. Examples of ME algorithms include:

- Full search
- Hierarchical search
- Diamond search

2.3.2 Motion Compensation

Motion Compensation (MC) is used in decoding the image that is encoded by ME [1]. This reconstruction of the image is done from received motion vectors and the reference frame. Similar to the previous video coding standards, H.264 is also a block-based motion-compensated hybrid video coder. However, it possesses many specifics that considerably affect the coding efficiency and complexity of the ME process. In order to achieve higher coding efficiency, H.264 adopts multiple block sizes for ME. As specified in H.264, a Macro-block (MB) can be partitioned into 7 different block sizes or modes for inter-frame prediction. The block sizes can be of 16×16, 16×8, 8×16, and 8×8 pixels. The 8×8 mode can further lead to another 4 small block sizes, namely, 8×8, 8×4, 4×8, and 4×4, as illustrated in Figure 2.3 [3]. Moreover, H.264 supports multiple reference frames for prediction [24]. In other words, more than one previously decoded frame can be employed as the reference frames to code the current frame.

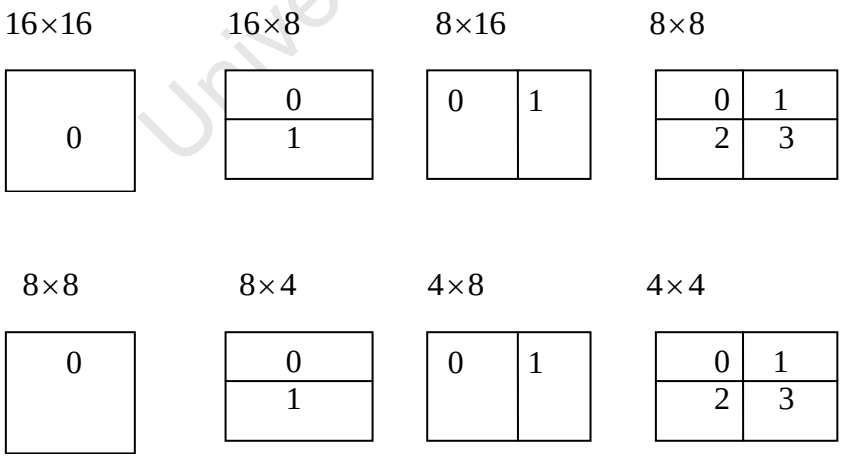


Fig. 2.3: The Various H.264 Block Sizes [1].

2.3.3 Inter And Intra Prediction

Prediction of the current macroblock can be achieved by using the current frame or field on one hand and previously coded frames or fields on the other. When the current frame or field is used for prediction it is called intra prediction and previously coded frames are used it is called inter prediction. The predicted macroblock is then subtracted from the current frame to create the residual macroblock. In both intra and inter prediction 4x4 and 16x16 block sizes are used.

2.3.4 Integer DCT

The main 4×4 or 8×8 integer DCT transform is used to convert the spatial domain signal into its equivalent frequency domain. The residual block is decomposed or transformed into a matrix of coefficients in which the near zero coefficients are of high frequency while the others are of low frequency [1] [2]. Each coefficient represents a weighting value for standard bases pertain and when combined the patterns recreate the residual block [1] [2] [4]. The detailed mathematical modelling is described in Chapter Three.

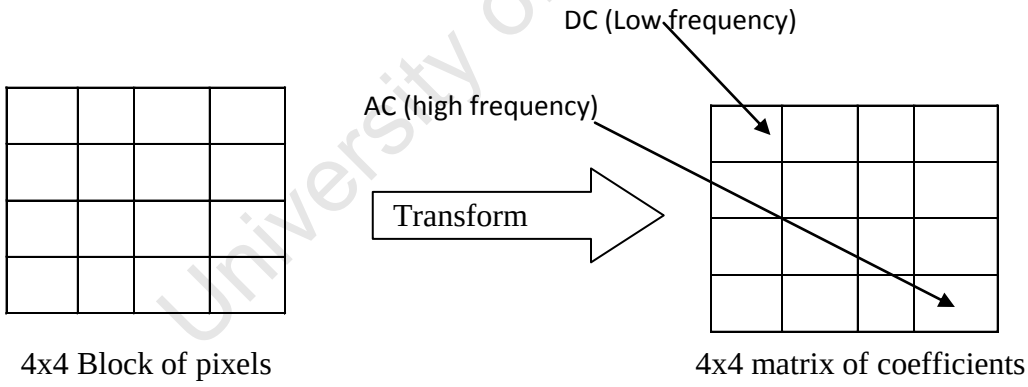


Figure 2.4: The Transform Process [27]

2.3.5 Quantisation

The integer transform coefficients are quantised to remove unimportant values, so that only significant coefficients are left for representing the residual frame as shown in figure 2.5. The process involves dividing each coefficient with an integer value depending on the chosen quantisation parameter (QP). If QP is high more

coefficients are set as redundant which means high compression and therefore reduced video quality and vice versa. A trade off has to be made such that QP is neither too high nor too low for efficient use of the bandwidth or storage media.

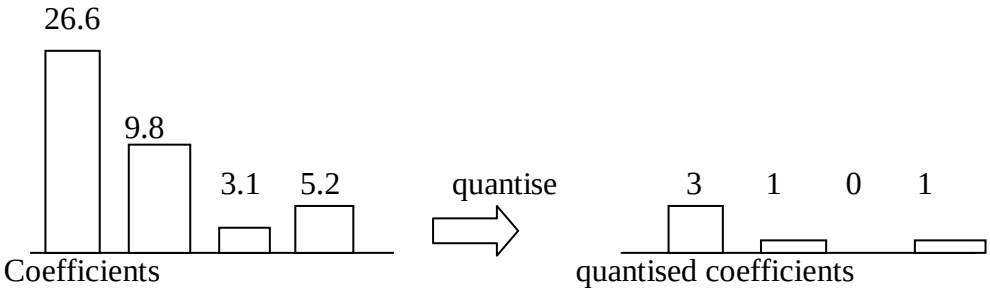


Figure 2.5: The Quantisation Process [18]

In practice two types of quantisation schemes exists and these are vector and scalar quantiser. The vector quantiser deals with a group of input signals while the scalar quantiser deals with a single input signal [1] [4]. These input signals and/or signal are mapped into the equivalent quantised values and/or value. H.264/AVC adopts the scalar quantiser [1] and its mathematical model is also decribed in Chapter Three.

2.3.6 Entropy Coding

An entropy coder converts the quantised series of symbols representing elements of a video sequences into compressed bit streams suitable for transmission and / or storage. There are two popular entropy encoding schemes: the Huffman and Arithmetic. Arithmetic coding is used in lossy compression algorithms. In Previous standards Variable Length Codes (VLCs) arithmetic coding was used [7]. The H.264/AVC standard adopts the Context Adaptive Variable Length Coding (CAVLC) and Context Adaptive Binary Arithmetic Coding (CABAC) which are advanced arithmetic coding schemes [15]. Huffman coding is used in lossless compression algorithm and since the lossy compression is considered here, Huffman coding will not be discussed in detail, suffice it to mention that in this scheme exact sets of code words are used for video sequences. The entropy encoder also compresses motion vector and header information. This removes statistical redundancy in the data in the compressed bit stream [1]

2.3.7 De-Blocking Filter

Irregularity in both the encoded and decoded video can be reduced by the de-blocking filter. These irregularities are normally caused by blocking artifacts especially on the block edges. This therefore improves the visual quality or appearance of the video frame. In the encoder the filter is applied after the inverse transform process while in the decoder it is applied before the reconstruction and display of the macroblock [1]. The filtered image is used for motion compensated prediction of future frames and this can improve compression performance because the filtered image is often a more faithful reproduction of the original frame than a block, unfiltered image.

2.4 H.264/AVC Structure

2.4.1 Macroblocks

A macro block is a representation of a coded picture into luminance (luma) and chrominance (chroma) samples. One microblock is 16x16 pixels. Each luma sample (Y) is 16x16 while each chroma sample (Cb and Cr) is 8x8 because the chrominance is subsampled. Macroblock can either be of I or P or B type and are arranged in slices.

2.4.2 Slice

A slice is a set of macroblocks in raster scan order [1]. There are several types of slices: I (Intra) slice-may contain only I (intra-coded) macroblocks. As mentioned a slice can have more than one macroblock, P (Predicted) slice may contain P and I macroblocks, B (Bi-predicted) slices-may contain B as well as I macroblocks, SP (Switching P) Slice-facilitate switching between the coded streams. Coded streams can contain P and/ or I macroblock, SI (Switching I) Slices- consists of special type of coded intra-coded macroblock.

2.4.3 H.264/AVC Profiles and Level

Profiles and levels specify well-defined sets of syntax constraints (for encoders) and decoder processing capabilities (for decoders), thereby enabling interoperability between encoder and decoder implementations within applications of the standard and between various applications that have similar functional requirements. A profile defines a set of syntax features for use in generating conforming bit streams [10].

The diverse coding implements and capabilities of the H.264/AVC video coding algorithm are arranged under various profiles (Fig. 2.5). Each of these profiles has been defined with specific target applications. The profiles and the specific tools offered in those profiles can be listed as [1] [3] [30]. H.264/AVC supports the following profiles: high, extended, main and baseline as seen in Figure 2.5[31].

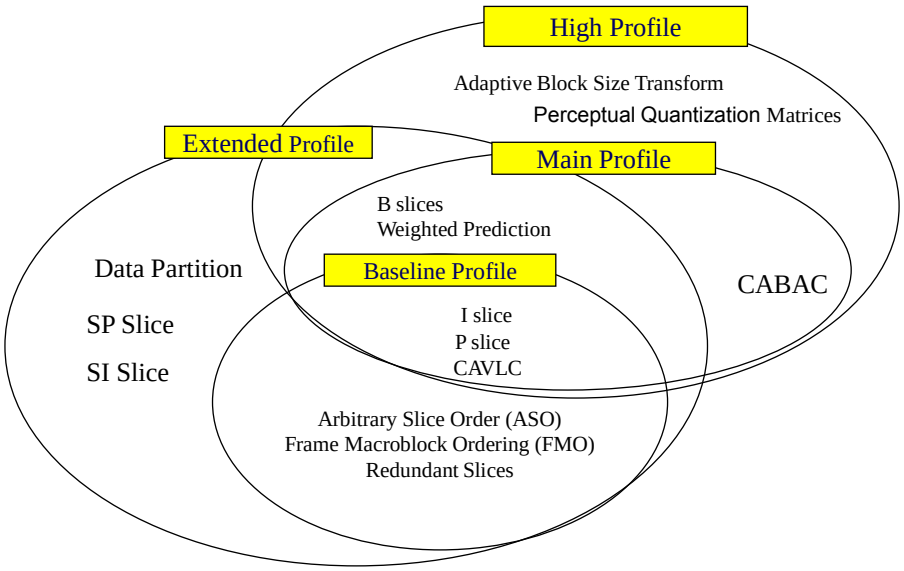


Figure 2.6:H.264 Baseline, Main and Extended profile [31]

2.4.3.1 Baseline Profile

This profile is mainly for video streaming applications. It is also deployed in applications such as mobile phones, mobile TV, video telephony, and portable storage media such as the Apple video iPod [10]. It has the following features [16]: that only I and P slice types may be present [3]. It uses CAVLC for entropy coding as opposed to

arithmetic and variable length coding used in previous standards. Flexible Macroblock Order (FMO**)** in which macroblocks may not necessarily be in the raster scan order. The map assigns macroblocks to a slice group. Arbitrary Slice Order (ASO**)** in which the ordering of the slices in the bitstream may not be in raster scan order [17].

The "baseline" profile of H.264/AVC uses three (3) transforms: a transform for the 4x4 array of luma DC coefficients (Hadamard transform) in intra macro-blocks (predicted in 16x16 mode), a transform for the 2x2 array of chroma DC coefficients (also Hadamard transform) in any macro-block and a transform for all other 4x4 blocks (main) in the residual data. The mathematical models, tables, etc on the integer and Hadamard transform are outlined in Chapter three.

2.4.3.2 Main Profiles

The Main profile is deployed in a limited number of storage media applications such as the Sony PlayStation Portable and Standard Definition Television(SD TV) [10].This profile has the following features: It has the following features: of I, P, B slices, weighted prediction, uses both CABAC and CAVLC for entropy coding.

2.4.3.3 High Profiles

A new profile called the high profile has been built on the main profile. The High profile is deployed broadly for Standard Definition (SD) and High Definition Television (HDTV) such as: BSkyB, Direct TV, Dish Network, Premiere,and optical storage media (Blu-ray Disc and HD DVD) [10].

There are various types of high profiles and these include [17]: high profile (HP) supporting 8-bit video with 4:2:0 sampling, addressing high end consumer use and other applications using high resolution video. High 10 profile (Hi10) supports 10-bit video with 4:2:0 sampling. High 4:2:2 profile (H422P) which supports up to 4:2:2 sampling and up to 10-bits per sample. High 4:4:4 profile (H444P) which supports up to 4:4:4 sampling, up to 12 bits per sample, lossless region coding and integer residual colour transform for coding RGB video [17]. These high profiles have the following features:

- All the features of main profile,
- Adaptive block size transform (introduction of 8x8 integer DCT),

- Perceptual quantisation matrices and
- 8x8 Intra prediction modes

2.4.3.4 Extended Profile

The extended profile has had no deployments reported yet [10]. This profile contains all the features of the baseline profile as shown in Figure 2.6. Further features include, SI slices, B slices, weighted prediction and it also supports data partitioning.

2.4.4 Video Colour Space

The YCbCr colour space and its variations are the popular way of representing colour images [1] in H.264. Only the luma (Y) and the blue and red (CbCr) are transmitted and/ or stored. The RGB colour space is avoided. YCbCr has the advantage that:

- Cb and Cr components may be represented with a lower resolution than Y because the HVS is less sensitive to colour (chrominance) than brightness (luminance).
- The amount of data required to represent the chrominance is reduced.

The sampling format adopted in H.264 is 4:2:0 which means that each of Cb and Cr has half the horizontal and vertical resolution of Y. Other sampling format include: 4:4:4 in which all the three components (Y, Cb and Cr) have the same resolution. In 4:2:2 sampling the chrominance components have the same vertical resolution as the luminance but half the horizontal resolution [1].

2.5 Chapter Summary

This chapter has given brief overview of MPEG-4(AVC) and H.264 standards on which integer transforms are based including principles of video compression. It also discussed the structure of H.264/AVC in terms of macroblocks, slices and colour format. Profiles have been outlined with particular interest in the baseline profile which consists of integer DCT transforms and Hadamard (Luma and Chroma DC transforms). The functional units of the H.264/AVC encoder have also been discussed.

Chapter Three

3. Video Coding Transforms In H.264/AVC Encoder

3.1 Chapter Introduction

This Chapter gives a detailed description and modelling of Integer DCT and Hadamard transform modules. The quantisation processes have also been outlined since, in most practical encoders, the transform process is linked with the quantisation process. These modules described here form the main focus area in this research.

3.2 Developing the Integer DCT from the Conventional DCT

When discussing the background on integer DCT in chapter one, we alluded to the fact that the H.261, H.263, JPEG, MPEG-1 and MPEG-2 use the discrete cosine transform as a transform coding in their encoder. Additionally, we stated that the JPEG 2000 and H.264/AVC use the discrete wavelet transform and Integer DCT respectively. We will now expand on our discussion of discrete cosine transforms and develop an understanding of integer DCT, the primary transform in this research.

Let us assume X is a block of $N \times N$ size and represents image samples or residual values after prediction and Y another block of $N \times N$ size representing transformed coefficients. If A is the DCT transform matrix of $N \times N$ size , then the forward DCT and inverse DCT are given by [1] [18]:

$$Y = AXA^T \tag{3.1}$$

The elements of A are [1]:

$$A_{ij} = C_i \cos\frac{(2j+1)i\pi}{2N} \text{ where } C_i = \sqrt{\frac{1}{N}}(i=0),\ C_i = \sqrt{\frac{2}{N}}(i>0) \tag{3.2}$$

Equation (3.1) may be written in summation form:

$$Y_{xy} = C_x C_y \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} X_{ij} \cos\frac{(2j+1)y\pi}{2N} \cos\frac{(2i+1)x\pi}{2N} \tag{3.3}$$

For a 4x4 matrix, N=4 and substituting in equation (3.2), the transform matrix A will be [1]:

$$A=\begin{bmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ \sqrt{\frac{1}{2}\cos\left(\frac{\pi}{8}\right)} & \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}\cos\left(\frac{3\pi}{8}\right)} & -\sqrt{\frac{1}{2}\cos\left(\frac{\pi}{8}\right)} \\ 1/2 & -1/2 & -1/2 & 1/2 \\ \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}\cos\left(\frac{\pi}{8}\right)} & \sqrt{\frac{1}{2}\cos\left(\frac{\pi}{8}\right)} & -\sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) \end{bmatrix} \quad (3.4)$$

This can be written as:

$$A=\begin{pmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{pmatrix} \quad (3.5)$$

Where: $a=\frac{1}{2}$, $b=\sqrt{\frac{1}{2}}\cos\frac{\pi}{8}$, $c=\sqrt{\frac{1}{2}}\cos\frac{3\pi}{8}$

In most practical encoders the DCT operates on the 8x8 block size. The concept of equations (3.1) to (3.5) can therefore be adapted to this bock size, by setting N=8.

3.3 Forward and Inverse Integer DCT Transforms

3.3.1 Forward Integer DCT Transforms

The main 4x4 integer DCT transform can be developed as shown by [1] [18] from the DCT. From the 4x4 DCT equations (3.5); the transpose of A is:

$$A^T=\begin{pmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{pmatrix} \quad (3.6)$$

If the input spatial domain matrix (i.e. 4x4 block sample from an image) is:

$$X=\begin{pmatrix} x00 & x01 & x02 & x03 \\ x10 & x11 & x12 & x13 \\ x20 & x21 & x22 & x23 \\ x30 & x31 & x32 & x33 \end{pmatrix} \quad (3.7)$$

The output frequency domain matrix (i.e. transform coefficients) will be as in equation (3.1):

Substituting equations (3.5), (3.6) and (3.7) into (3.1), we have:

$$Y = \begin{pmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{pmatrix} X \begin{pmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{pmatrix} \quad (3.8)$$

3.3.1.1 Matrix Multiplication Algorithm-Forward Integer Transform

The 4x4 DCT Equation (3.8) can be factorised as follows:

$$Y = \left(C_f X C_f^{\,T}\right) \otimes E_f \quad (3.9)$$

From which: C is the integer transform matrix, $C_f^{\,T}$ is the transpose of matrix C_f, X is the input spatial domain matrix, E is a matrix of scaling factors and $\otimes$ Indicates that each element of $C_f X C_f^{\,T}$ is multiplied by the scaling factor in the same position in matrix E_f (Scalar multiplication and not matrix multiplication) [1] [9].

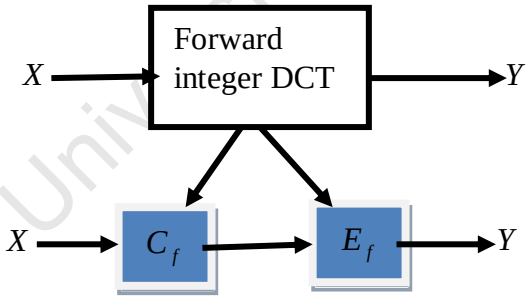


Figure 3.1: Block Diagram of Forward integer DCT

Hence:

$$Y = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{pmatrix} X \begin{pmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{pmatrix} \otimes \begin{vmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{vmatrix} \quad (3.10)$$

Where: $a = \frac{1}{2}$, $b = \sqrt{\frac{1}{2}}\cos\frac{\pi}{8}$, $d = \frac{\sqrt{\frac{1}{2}}\cos\frac{3\pi}{8}}{\sqrt{\frac{1}{2}}\cos\frac{\pi}{8}} \approx 0.414$

To simplify the implementation of the transform $d \approx 0.5$ and b is modified to $b = \sqrt{\frac{2}{5}}$ to make it orthogonal. The second and fourth rows of C_f, and second and fourth columns of C_f transpose, are scaled by a factor of two (2). Conversely, E_f is scaled down to avoid multiplication by half in $C_f X C_f^{\,T}$

Therefore:

$$Y = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{pmatrix} \begin{pmatrix} \quad \end{pmatrix} X \begin{pmatrix} \quad \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{pmatrix} \otimes \begin{vmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{vmatrix} \qquad (3.11)$$

The core matrix ($C_f X C_f^{\,T}$) of equation (3.11), can be implemented by multiplying the three matrices C, X and $C_f^{\,T}$ or using additions, subtraction and shift operations in software.

3.3.2 Inverse Integer DCT Transform

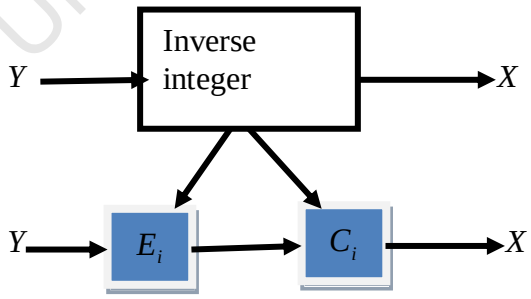


Figure 3.2: Block Diagram for the Inverse integer DCT

Equation (3.11) gave the video signal in the frequency domain (Y).We can now proceed to derive the equation to transform equation (3.11), back to the spatial domain (X). The inverse integer transform is given by [2] [18]:

$$X = A_i^T Y A_i \tag{3.12}$$

Where X is the spatial domain video sample, Y is the frequency domain video signal equivalent, A_i is the inverse transform matrix and A_i^T is the transpose of the inverse transform matrix.

Here:

$$A_i = \begin{pmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{pmatrix}, \; a = 1/2, \; b = \sqrt{1/2}\cos \pi/8 \text{ and } c = \sqrt{1/2}\cos 3\pi/8$$

Choosing a particular approximation by scaling each row of A_i and rounding to the nearest 0.5 gives C_i

Thus:

$$C_i = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0.5 & -0.5 & -1 \\ 1 & -1 & -1 & 1 \\ 0.5 & -1 & 1 & -0.5 \end{pmatrix} \text{ and } C_i^T = \begin{pmatrix} 1 & 1 & 1 & 0.5 \\ 1 & 0.5 & -1 & -1 \\ 1 & -0.5 & -1 & 1 \\ 1 & -1 & 1 & -0.5 \end{pmatrix}$$

The rows of C_i are orthogonal. They have non- unit norms. According to [1] [18], we have to multiply all the values of C_{ij} in row r by:

$$\frac{1}{\sqrt{\sum_j C_{rj}^2}} \tag{3.13}$$

this way we restore orthonormality[2] [18].

$$A_i = C_i . R_i \tag{3.14}$$

Where:

$$R_i = \begin{pmatrix} 0.5 & 0.5 & 0.5 & 0.5 \\ \sqrt{2/5} & \sqrt{2/5} & \sqrt{2/5} & \sqrt{2/5} \\ 0.5 & 0.5 & 0.5 & 0.5 \\ \sqrt{2/5} & \sqrt{2/5} & \sqrt{2/5} & \sqrt{2/5} \end{pmatrix}$$

$$A_i^T = C_i^T . R_i^T \tag{3.15}$$

3.3.2.1 Matrix Multiplication Algorithm-Inverse Integer Transforms

Substituting equation (3.14) and 3.15) into (3.12) we have:

$$X = [C_i^T \otimes R_i^T].[Y].[C_i \otimes R_i]$$

Rearranging:

$$X = C_i^T [Y \otimes R_i^T \otimes R_i] C_i$$

$$X = C_i^T [Y \otimes E_i] C_i \tag{3.16}$$

Where E_i, the rescaling factor, is given by [2] [18]:

$$E_i = R_i^T \otimes R_i$$

$$E_i = \begin{pmatrix} \frac{1}{4} & \frac{1}{\sqrt{10}} & \frac{1}{4} & \frac{1}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} & \frac{2}{5} & \frac{1}{\sqrt{10}} & \frac{2}{5} \\ \frac{1}{4} & \frac{1}{\sqrt{10}} & \frac{1}{4} & \frac{1}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} & \frac{2}{5} & \frac{1}{\sqrt{10}} & \frac{2}{5} \end{pmatrix}$$

Therefore:

Equation (3.16) is the inverse transform matrix derivation for changing frequency domain to spatial domain. The equations also show that X is obtained by multiplying the four matrices Y, E_i, C_i and the transpose C_i.

3.4 Quantization and Re-Scaling

3.4.1 Quantisation

The scalar multiplication, in the core forward and inverse transforms, can be absorbed into the scalar quantization step. The basic quantization operation (Z) for forward transform can be defined as follows [1]:

$$Z_{ij} = round\left(Y_{ij} / Q_{step}\right) \tag{3.17}$$

There are 52 values of Q_{step} supported by H.264/AVC. Each of these Q_{step} has a Quantization Parameter (QP) associated with it as shown in table 3.1.

Table 3.1: Quantisation Steps and Parameter [1]

QP	0	1	2	3	4	5		18			48		51
Q_{step}	0.625	0.702	0.787	0.884	0.992	1.114		5			160		224

The ratio between successive Q_{step} values is chosen to be $\sqrt[6]{2} = 1.2246$ such that Q_{step} doubles in size for every increment of 6 in QP. The wide range of quantizer step sizes makes it possible for an encoder to control the trade-offs accurately and flexibly between bit rate and quality. The values of Q_{step} can be different for luma and chroma. Both parameters are in the range 0–51[1] [18]

To integrate the scalar multiplication by matrix E_f or E_i, as shown above, and to avoid any division operations, the equation is changed to:

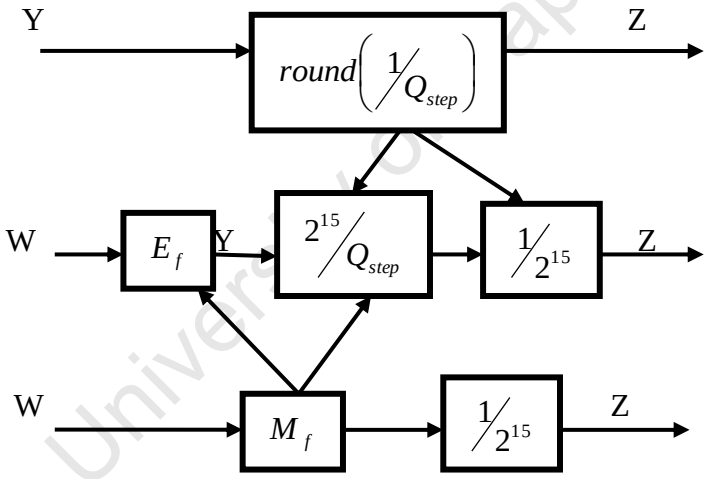


Figure 3.3: Block Diagram of Quantisation Process

$$Z_{ij} = round\left(W_{ij} * \left(PF / Q_{step}\right)\right) \tag{3.18}$$

Where:

$$PF / Q_{step} = M_f / 2qbits \tag{3.19}$$

$$qbits = 15 + floor\left((QP)/6\right) \tag{3.20}$$

And PF is the value from matrix E_f or E_i, depending on the location of Y_{ij}

Finally, (3.17) can be rewritten as follows:

$$\left|Z_{ij}\right| = \left(\left(\left|W_{ij}\right| * M_f + f\right)\right) >> qbits \tag{3.21}$$

$$\left(sign\left(Z_{ij}\right) = sign\left(W_{ij}\right)\right) \tag{3.22}$$

The offset parameter, *f*, is *2qbits/3* for Intra Blocks or *2qbits/6* for Inter Blocks

3.4.1.1 Development of Multiplication Factor, M_f

The multiplication factor M_f can be derived [2] [18]

$$M_f = \frac{E_f . 2^{15}}{Q_{step}} \tag{3.23}$$

And

$$Q_{step} = \frac{V_i}{E_i 2^6} \tag{3.24}$$

Equation (3.23) and (3.24) can be combined to yield [2] [18]:

$$M_f \approx \frac{E_f . E_i . 2^{21}}{V_i}$$

Where:

E_f, E_i are the forward and inverse scaling matrices and V_i is defined in the H.264 standard as illustrated in equation (3.29) and (3.18)

Thus [2] [18]:

$$M_f = round\left(\frac{E_f . E_i . 2^{21}}{V_i}\right) \tag{3.24}$$

$$E_f . E_i . 2^{21} = \begin{pmatrix} 131072 & 104857.6 & 131072 & 104857.6 \\ 104857.6 & 83886.1 & 104857.6 & 83886.1 \\ 131072 & 104857.6 & 131072 & 104857.6 \\ 104857.6 & 83886.1 & 104857.6 & 83886.1 \end{pmatrix}$$

The entries in matrix M_f may be calculated as shown in Table 3.2

Table 3.2: Multiplication Factor [18]

QP	M_f position (0,0)(0,2) (2.0)(2,2)	M_f position (1,1)(1,3) (3,1)(3,3)	remaining M_f position
0	13107	5243	8066
1	11916	4660	7490
2	10082	4194	6554
3	9362	3647	5825
4	8192	3355	5243
5	7282	2893	4559

Therefore, for QP values from 0 to 5, M_f may be obtained from the last three columns of table 3.2

The complete forward transform, scaling and quantisation process (for 4x4 blocks and for modes excluding 16x16 intra) is [2] [18]:

$$Z = round\left(\left[C_f X C_f^t \right] \otimes M_f . \frac{1}{2^{15}} \right).$$

$$Z = round\left(\left[C_f X C_f^t \right] \otimes [m(QP\%6, n) / 2^{floor(QP/6)}] . \frac{1}{2^{15}} \right). \quad (3.25)$$

3.4.2 Inverse Quantisation (Re-quantisation)

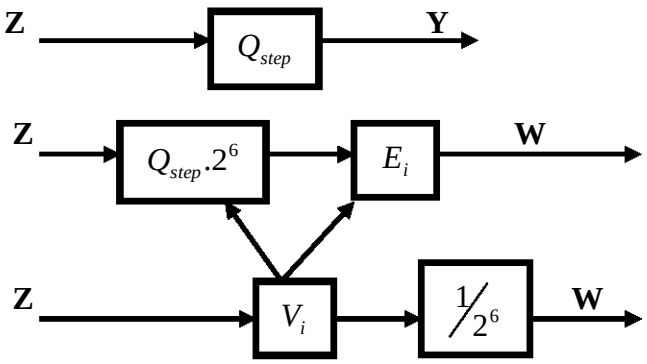


Figure 3.4: The Basic Inverse Quantizer

The basic inverse quantizer is given as [1] [2]:

$$Y_{ij} = Z_{ij} Q_{step} \tag{3.26}$$

When the pre-scaling matrix *Ei* is incorporated together with a constant scaling factor of 64 (to avoid rounding errors), equation 21 changes to:

$$W_{ij} = Z_{ij} Q_{step}.PF.64 \tag{3.27}$$

In the H.264/AVC, the parameter V_i (Q_{step} $.PF.64$) is defined. Thus:

$$W_{ij} = Z_{ij} V_{ij}.2^{floor(QP/6)} \tag{3.28}$$

3.4.2.1 Development of the Scaling Matrix, V_i

Though the H.264 standard supports a total of 52 values of Q_{step}, indexed by a QP, the precise step sizes are not defined in the standard, but rather the scaling matrix, V_i , is specified. Q_{step} values corresponding to the entries in V_i are computed and shown in tables 3.3 and 3.4 respectively. The values in the matrix V_i depend on Qstep and therefore QP and the scaling matrix E_i. V_i can be derived from equation (3.24) and is given by [2] [18]:

$$V_i = E_i.Q_{step}.2^6 \tag{3.29}$$

For QP =0, V_i can be computed as follows:

Table 3.3: Computation of Scaling Matrix

QP	$Q_{step}.2^6$	$V_i = round\left(E_i.Q_{step}.2^6\right)$
0	40	$\begin{pmatrix} 10 & 13 & 10 & 13 \\ 13 & 16 & 13 & 16 \\ 10 & 13 & 10 & 13 \\ 13 & 16 & 13 & 16 \end{pmatrix}$

Table 3.4: Matrix V defined in H.264 Standard [18]

QP	V(r,0) V_i position(0,0) (0,2)(2.0)(2,2)	V(r,1) V_i position(1,1) (1,3)(3,1)(3,3)	v(r,2) remaining V_i position
0	10	16	13
1	11	18	14
2	13	20	16
3	14	23	18
4	16	25	20
5	18	29	23

There are only three unique values in each matrix V_i. These are defined as a table of values in the H.264 standard, for QP 0 to 5:

The complete inverse transform, rescaling and re-quantisation process (for 4x4 block and for modes excluding 16x16 intra) is [2] [18]:

$$X = round\left(\left[C_i^T \right] . [Z \otimes V_i . \frac{1}{2^6} \right).$$

$$X = round\left(\left[C_i^T \right] . [Z \otimes [v(QP\%6, n) . 2^{floor(QP/6)}] . [C_i] \frac{1}{2^6} \right).$$

3.5 Hadamard Transforms

The Hadamard transforms is applied to the DC components of the residual video signal. It has two parts: Hadamard transform for the 4x4 array of luma DC coefficients in intra macroblock predicted in 16x16 modes and a Hadamard transform for the 2x2 array of chroma DC coefficient in any macroblock

3.5.1 Hadamard Luma DC Forward Transform and Quantisation

Luma DC transform is added on top of intra 16x16 transform. The input matrix is formed by picking out DC coefficients from the 16 transformed 4x4 blocks

W_D. W_D will be transformed using a 4x4 Hadamard transform to exploit correlations among DC coefficients.The grouped DC block of luma is transformed as follows [1][2]:

$$Y_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} & W_D & \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \Big/ 2 \tag{3.30}$$

Quantization is performed using the following equation [1]

$$Z_{D(i,j)} = \left(\left| Y_{D(i,j)} \right| .MF(0,0) + 2f \right) >> (qbits + 1) \tag{3.31}$$

$$sign\left(Z_{D(i,j)}\right) = sign\left(Y_{D(I,J)}\right) \tag{3.32}$$

3.5.1.1 Hadamard Luma DC Inverse Transform and Re-Quantisation

The inverse Hadamard transforms can be applied in the similar way as the forward Hadamard transforms. The only exception is that unlike in the integer DCT the order of matrix is not reversed. The overall product is also not dived by two as shown in equation (3.33).

$$W_{QD} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} & Z_D & \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \tag{3.33}$$

The rescaling process is performed as follows, for QP greater than or equal to 12[1]:

$$W_{D(i,j)} = W_{D(i,j)} . V_{(0,0)} . 2^{floor(QP/6)-2} \tag{3.34}$$

$$W_{D(i,j)} = W_{D(i,j)} . V_{(0,0)} . 2^{(QP/6)-2} \tag{3.35}$$

If QP is less than 12, rescaling is performed by:

$$W_{D(i,j)} = [W_{D(i,j)} . V_{(0,0)} . + 2^{1- floor(QP/6)}] >> \left(2 - floor\left(QP/6\right)\right) . \tag{3.36}$$

$$W_{D(i,j)} = [W_{D(i,j)} . V_{(0,0)} . + 2^{1-(QP/6)}] >> \left(2 - floor\left(QP/6\right)\right) \tag{3.37}$$

3.5.2 Hadamard Chroma DC Forward Transform And Quantization

Chroma DC transform is added on top of Chroma transform. The input matrix is formed by picking out DC coefficients from the 4 transformed 4x4 blocks.
The grouped DC blocks of chroma are further transformed as follows

$$W_{QD} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} & \\ W_D & \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.38}$$

$$\left|Z_{D(ij)}\right| = \left(\left|Y_{D(ij)}\right| . MF_{(0,0)} + 2f\right) >> \left(qbits + 1\right) \tag{3.39}$$

$$sign\left(Z_{D(ij)}\right) = sign\left(Y_{D(ij)}\right) \tag{3.40}$$

3.5.2.1 Hadamard Chroma DC Inverse Transform And Quantization

During decoding, the inverse transform is applied before rescaling
inverse transform is identical to the forward transform

$$W_{QD} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} & \\ W_D & \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.41}$$

If QP is greater than or equal to 6, rescaling is performed by:

$$\hat{W}_{D(i,j)} = W_{QD(i,j)} . V_{(0,0)} . 2^{floor(QP/6)-1} \tag{3.42}$$

If QP is less than 6, rescaling is performed by:

$$\hat{W}_{D(i,j)} = [W_{QD(i,j)} . V_{(0,0)} .] >> 1 \tag{3.43}$$

3.6 Chapter Summary

This chapter has given a detailed mathematical description of integer DCT and Hadamard transforms as used in H.264/AVC encoder. It has also discussed the quantisation and re-quantisation processes. These models will be used in experiments discussed in chapter five.

Chapter Four

4.0 An Overview of SIMD Technologies

4.1 Chapter Introduction

This chapter gives an overview of SIMD technologies used for exploiting data level parallelism. The technologies are discussed here to ascertain their application to integer transforms (4x4 integer DCT and 4x4 Hadamard transforms).

4.2 The SIMD Technologies

There are various categories of SIMD I technologies depending on the type of processor and manufacturer. Three common ones are those from Intel‘s Pentium three (III) and four (IV), Advanced Micro Devices (AMD)’s Athlon XP and K6-2 (and above) and AltiVec’s PowerPC®RISC SIMD processors. SIMD instructions in any SIMD machine essentially exploit data stream property known as data parallelism. Software performance can therefore be speeded-up since SIMD instructions allow parallel manipulation of multiple data in a single input operation [32] - [36]. Consequently, scientists, professionals and even software developers have been urged to use these technologies to exploit parallel benefits in modern processors [37] [38]. In this context, we get a number of pixels that requires the same instruction performed on it and process it in parallel at once as indicated in the figure below.

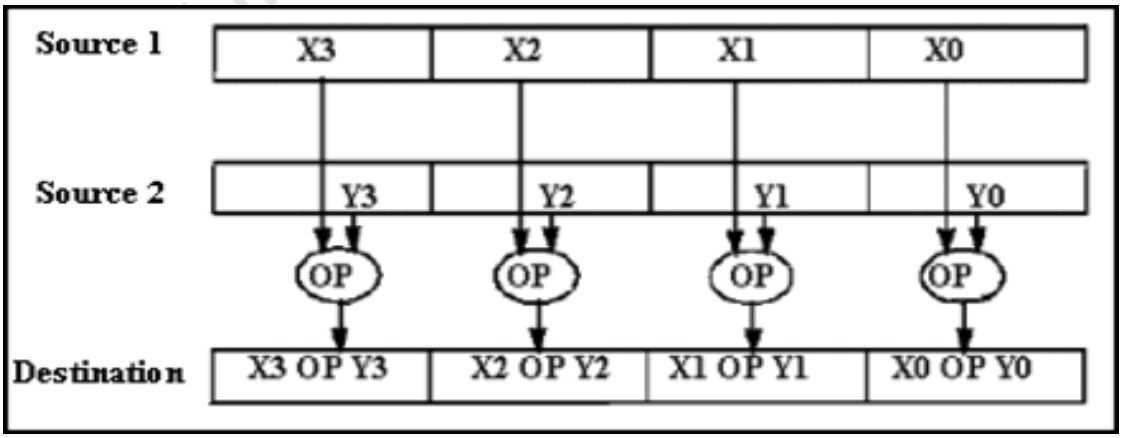


Figure 4.1: SIMD Execution Model [33]

4.3 The Intel's SIMD Instructions

Intel's Pentium three (3) and upward processors have SIMD capabilities. Supported in these processors are the MultiMedia eXtension (MMX), the Streaming SIMD Extension (SSE) and the Streaming SIMD Extension two (SSE2) instructions [31]. Others include SSSE3, SSE4 SIMD instructions

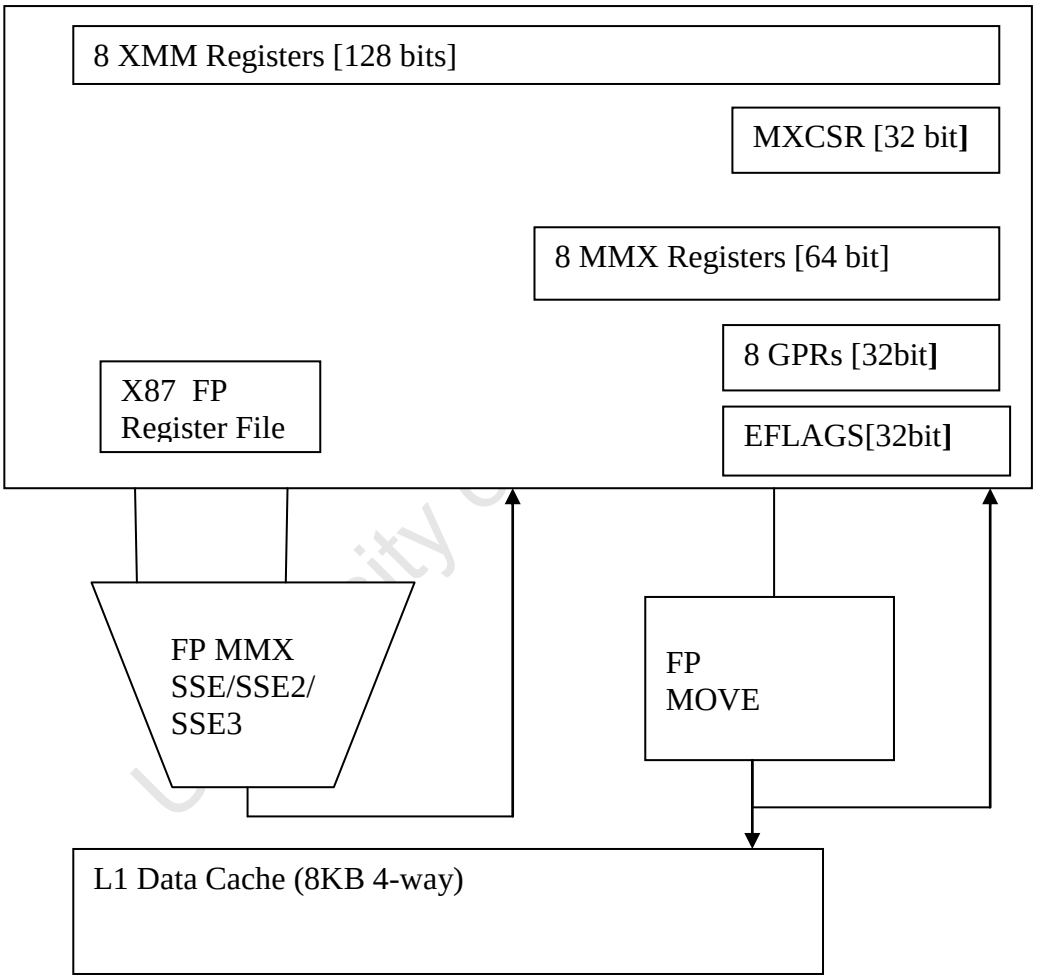


Figure 4.2: Pentium 4 Processor Intel SIMD Architecture [31]

4.3.1 The MMX Technology

The MMX technology SIMD instructions were added to the Intel's 1A-32 architecture to manipulate the packed 64-bit integer data type [4] [37]. In multimedia and as well as communications applications, [37] states that we can obtain between 1.5 to 2 speed-up with MMX technology. However, that is only true when compared with running the same experiment without MMX SIMD instructions [37]. Detailed examples on use of SIMD instructions in MMX technology have been given in [39] to illustrate its merits. Additional features to the Intel's 1A-32 architecture have been included, without loss of backward compatibility and operating systems mode. These features include [37]:

i. Four MMX data types
ii. Eight 64-bit data registers
iii. Fifty-Seven (57) MMX instructions.

4.3.1.1 The MMX Data Types

The MMX is a 64-bit data technology. It consists of packed and unpacked data in various types. Figure 4.3 is a diagram illustrating the various data types supported by MMX technology.

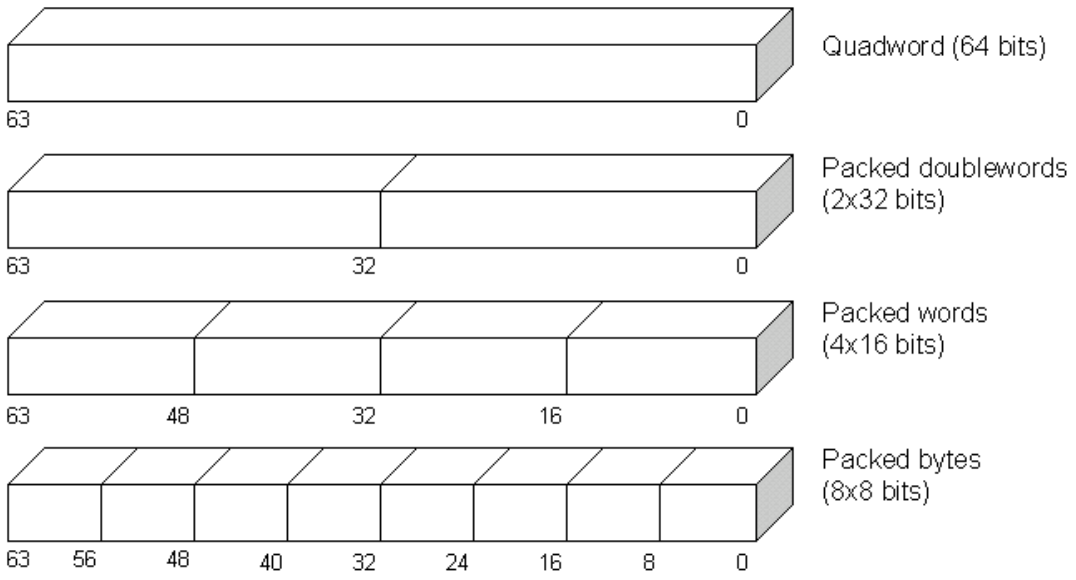


Figure 4.3: MMX Data Types [30]

The MMX technology has one unpacked quardword (64-bit) and therefore only one data symbol can be manipulated. The technology also consists of two packed double

(32-bit), four packed word (16-bit) and eight packed bytes (8-bit) data types. All numerical operations on symbols or numbers, which could represent pixels or images, can be interpreted as consisting quadword, doubleword , word or bytes [32]. In each of the packed data categories, each element is represented as a fixed-point integer [37].

4.3.1.2 The MMX Registers

There are eight (8) general purpose registers in the MMX architecture. Each of these registers can be addressed using a well established designation mm0-mm7 [37] [39]. These general purpose registers are considered as separate registers in the design of the Intel architecture. The pictorial view and designation of the eight general purpose registers is shown in Figure 4.4.

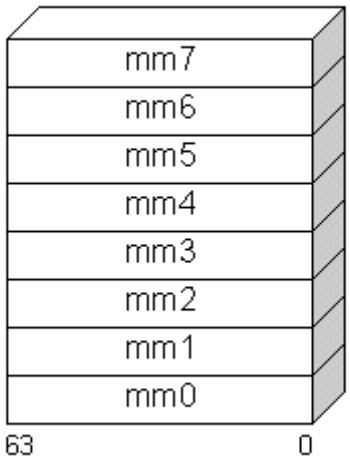


Figure 4.4: MMX registers [39].

Other registers include eight (8) 32 bit EAX, EBX, ECX, EDX, ESI, ESP and EBP which can be used in conjunction with the MMX registers to load and store data.

4.3.1.3 The MMX Instructions

A total of Fifty-seven (57) SIMD instructions have been added to the MMX technology. There are various categories of the MMX SIMD instruction. Intel's software development manual outlines these instructions as follows:

- **General support instructions**-these include instructions to convert from 32 bit to 64-bit integer data types and vice versa, loading instructions, pack and unpack (for interleaving data) instructions. Where a need to convert from integer to float arises the empty MM state instruction is used. The SIMD intrinsic instructions for the general support and their equivalent assembly instructions are given in Appendix B, table 1.

- **Arithmetic instructions** – these include instructions to perform addition and subtraction of eight, sixteen and thirty-two bit data types. It also consists of instructions to multiply and add or multiply only the upper and lower sixteen bit set of data. Appendix B, Table 1, gives the SIMD instrinsics and the equivalent Assembly mnemonics.

Other instructions set categories for MMX Technology intrinsics and assembly mnemonics codes given in Appendix B, tables 3 to 6 include:

- **Shift instruction** – these are of two types: the arithmetic and logical shifts. The logical shifts can operate on 16, 32 and 64 data type to either shift right or left. The arithmetic shift instructions operate only on 16 and 32 bit data.

- **Logical instructions** – These instructions can be used to perform the bitwise AND, AND NOT (NAND), OR and Exclusive OR operations.

- **Comparison instructions**- the comparison instructions include intrinsic and assembly instruction to perform *equal to* and *greater than* operations.

- **Set instruction –**two categories of set instructions include instructions to set to zero and set integer values. The set integer instructions are used only for MMX intrinsics and up-to eight 8-bit, two 32-bit, four 16-bit can be used to set data either in ascending or descending order.

4.3.2 The Streaming SIMD Extensions

The SSE is a 128-bit packed data type, in contrast with the MMX technology which is a 64-bit data type. It was also introduced in the IA-32 architecture to cater for various data types and multimedia applications needs. Thus, we have the SSE for float and SSE2 for integer data types. Other extensions and improvements to the SSE2

include SSSE3, SSE4 etc. Their primary operation is in the 128-bit register as well as memory [32].

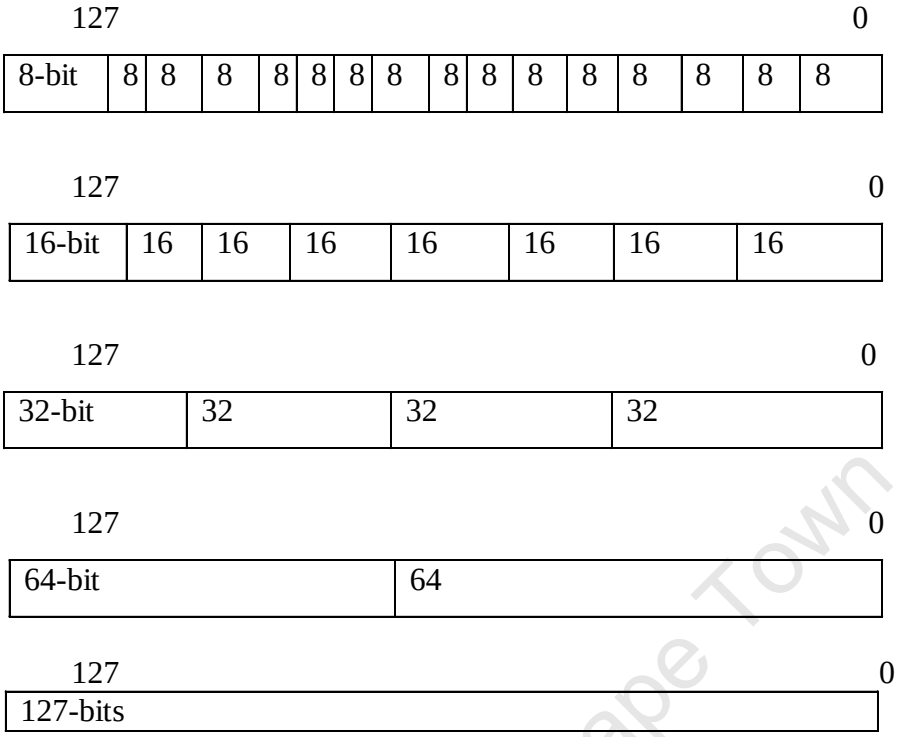


Figure 4.5: Data Types Introduced with the SSE Extension

Similar to MMX technology, SSE contains either packed data types or scalar single/double precision floating-point data type [33]. The packed float data type include sixteen byte (8-bit), eight word (16-bit), four doubleword (32-bit) and two quadword (64-bit) as illustrated by figure 4.5 .It can also contain a one 128-bit.

For SSE2, a similar arrangement of data to SSE exists. It however, operates on packed integer data types. The packed integers are arranged as [32].

• 128-bit packed byte integers — sixteen packed bytes

• 128-bit packed word integers — eight packed words

• 128-bit packed doubleword integers — four packed doublewords

• 128-bit packed quadword integers — two packed quadwords

4.4 AMD SIMD Instructions

The AMD technology consists of the 3DNow for the AMD K6-2, and above, processor and the 3DNow Professional for the AMD Athlon XP processor. Each of the 3DNow! Set of SIMD instructions supports conversion to and from integers, multiplication, division, addition and subtraction. It can also use d for data pre-fetching, absolute, reciprocal square root and negation. Furthermore, these instructions can also be used for computing division [37] [39].

As stated in *section 5.2*, our experiments were done using the Pentium processor. The AMD processors are discussed here to appreciate that SIMD instruction extends beyond the Intel Pentium processors [40].

4.4.1 3DNow SIMD

In 1998 AMD Athlon Released the AMD K6-2 microprocessor. The processor operates on a 64-bit floating data type. The 64-bit SIMD AMD K6.2 is equivalent to the MMX technology in terms of data bit length. The difference is that while the MMX operates with integer data types, the AMD K6.2 deals with floating-point data types of instructions. The AMD K6.2 was an improvement on the AMD K6 processor. Three more execution units were added to the K6-2 giving a total of 10 execution units compared to the AMD K6 processor. Two of these extra execution units were designed to support 3DNow using SIMD instructions [37] [40].

4.4.2 3DNow Professional SIMD

AMD's Athlon XP processors are similar to the Intel's Pentium Three and Four processors. They support 128-bit floating and integer data type similar to SSE and SSE2 instructions [39] of the Intel's Pentium.

4.5 AltiVec SIMD Instructions

AltiVec technology power PC processor architectures have also SIMD processing capabilities [33]. Like SSE and SSE2 these processors work on a dedicated file register of 128-bit wide through its vector execution module or unit. While SSE, SSE2 and SSE3 requires eight, 128-bit width, registers (xmm0 to xmm7) in its implementation, the Altivec instructions requires four (4) times that number (32 registers) for the same register width of 128-bit. In the AltiVec register, each value represents a vector consisting of elements. There are altogether 162 integer and floating-point SIMD instructions. Various vector lengths are supported depending on size of data and these include Four (4), eight (8) and sixteen (16) long. Signed and unsigned integers can be supported by these instructions as [37]:

i. 32-bit integers and floating-point numbers (IEEE), 4-way parallelism.

ii. 16-bit integers, 8-way parallelism[29]

iii. 8-bit integers, 16-way parallelism

4.6 Chapter Summary

An overview of SIMD technology has been given in this Chapter. Three main categories of processors from the Intel, AMD and Altivec are used to manipulate data in parallel. Various data types and sizes are supported in each of these SIMD technologies. Each of these technologies can be adopted in multimedia processing, including video compression, depending on type of processor.

Chapter Five

5 Experimental Methods

5.1 Chapter Introduction

This chapter describes the experimental tools and methods that were used to achieve the results of Chapter Six. The mathematical equations, derived in chapter three for the integer DCT and Hadamard transforms were implemented in software using C++ programming, inline assembly and/or MMX technology instructions. Two methods the matrix multiplication and the butterfly algorithm, were adopted in experiments. Each method was measured as a function of time and the faster method noted, taking into account each of the research questions outlined in Chapter One.

5.2 Experimental Tools

In the experiments, one Personal Computer (PC) equipped with Intel Pentium(R) Dual core 2.5 GHz CPU and 1.98GB memory was used. Microsoft visual studio 2008 was used as the programming environment. With the Microsoft visual studio 2008 programming environment, various software tools can be used to compress the video. The programming software supports both high (such as C++, C#, etc) and low (assembly) level programming languages. The various software languages used in this study are now briefly outlined. This section will also describe the three test videos which was used to generate the input matrix during the video compression process.

5.2.1 The C++ Programming Language

In the early 1980's Bjarne Stroustrup developed C++ high level programming language as an extension to C. Thus, C++ was added to the list of other high level programming languages such as FORTRAN, JAVA, C# and the like. We adopted this language as our main programming tool due to the following advantages that it has:

- It is easier to program and debug the codes
- The C++ compiler is readily available and compatible with many personal desk top computers

- Compared to assembly codes, C++ offers short code size
- Its flexibility property allows it to be used with Intel, Altivec and AMD SIMD intrinsics
- The C++ codes can also be inlined to exploit assembly language programming, a property that can be extended to SIMD instructions as well

Notwithstanding the merits outlines above, the C++ Language has the weakness that it uses a lot of control and branch statement which tends to slow programming codes and therefore minimises its use in some real-time applications.

5.2.2 Assembly Programming Language

Inline assembly programming allows direct accessing of the computer hardware. It is a technique of integrating the assembly language snippets in the C or C++ codes. With this technique, the programmer can use the function parameters and local variables defined in C or C++ classes and structures. In general, the inline declaration instructs the compiler to insert the assembly codes at the specified point. This part of the code then becomes an inline function [37]. There are several benefits derived in inlining and they include but are not limited to the following:

- It reduces function-call overhead [41].
- It can be used to produce codes or programs that execute faster than the equivalent compiled codes[41] [42]
- It enables the output to be made visible on C or C++ variables [41].

However, assembly language should not be used indiscriminately as they are architecture dependant and therefore tend to lack code portability [42] [43]. For example, programs using x86 Instructions cannot be compiled on PowerPC computers [42] .Additionally, like all low level assembly codes or languages, they have higher maintenance costs and are more tedious and error prone than compiled codes [43].

5.2.3 SIMD Programming

In our experiment the input matrix of pixels to be manipulated consists of four 16-bit *short integer* data type pixels. From the analysis of the various SIMD technologies discussed in chapter four, SSE instructions could not be used as it is a 128- bit quardword length, supporting manipulation of four 32-bit *floating point* data pixels. SSE2 too could not be used as it deals with four 32- bit *integer data* type pixels in parallel. We could also not use AltiVec SIMD instructions for the same reasons as SSE and in addition AltiVec operates on PowerPC®RISC processor architecture. Similarly, AMD instructions are also based on a different processor from our Pentium IV processor.

Our only appropriate technology for implementing 4x4 integer transforms is the MMX Technology instructions. With these instructions we can add, subtract or shift right or left four 16-bit integer data types in parallel.

5.2.4 Standard Test Video Sequences

Three standard test video sequences *Foreman, Soccer* and *Mobile &Calendar* were used as input video signals in our experiments. Each of these videos were captured or converted to the Common Intermediate Format (CIF), a popular format for videoconferencing applications [1], and at a horizontal by vertical luminance resolution of 352x 288. The video sequence has a total of 300 frames each. The video sequences are sampled at the rate of 30 frames per second.

The integer DCT and Hadamard transform codes used in the H.264/AVC are compared to our implementation in terms of computation time while keeping the peak signal to noise ratio constant.

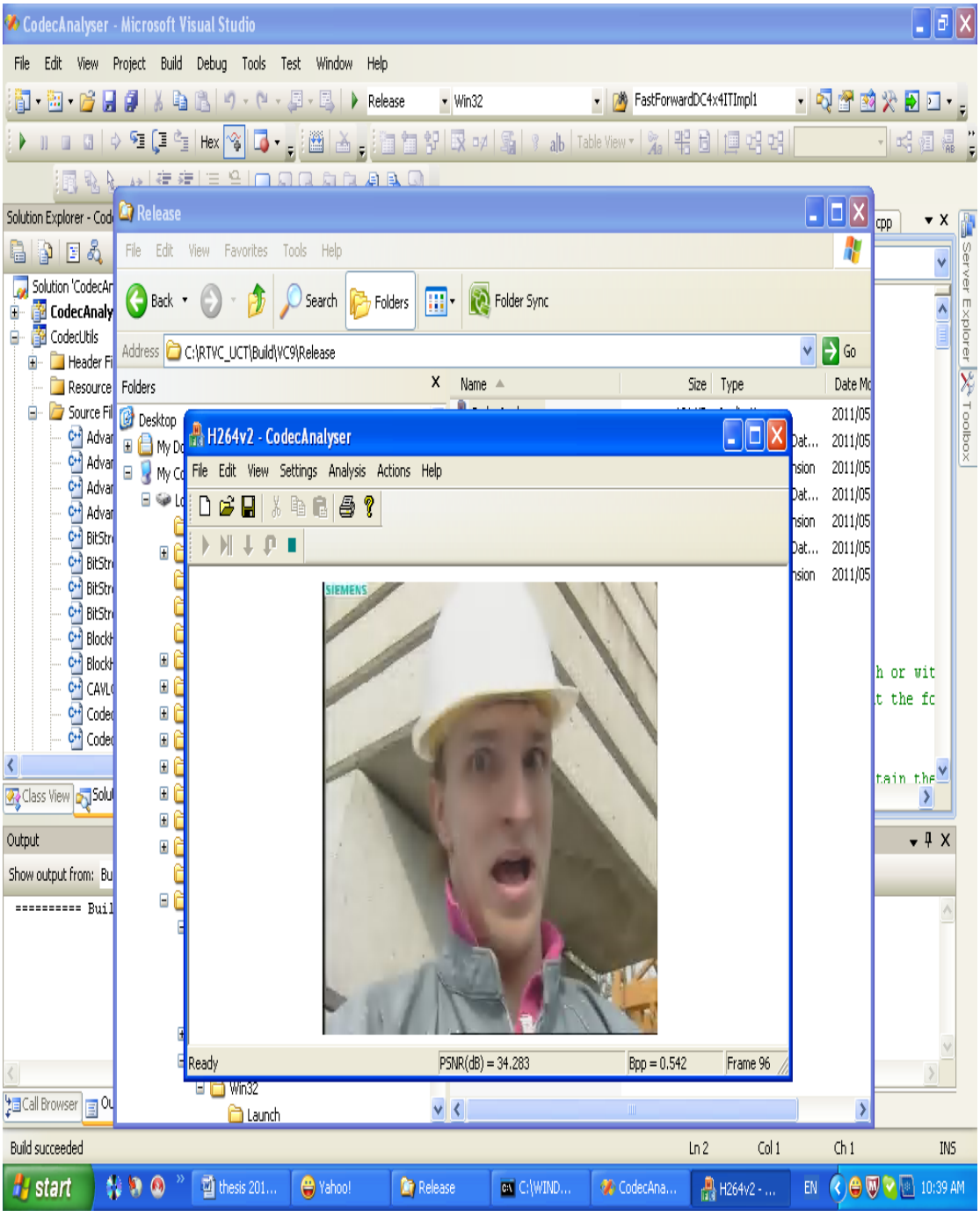


Figure 5.1: One of the Standard Test Signals used in the Experiments

5.3 Experimental Set Up

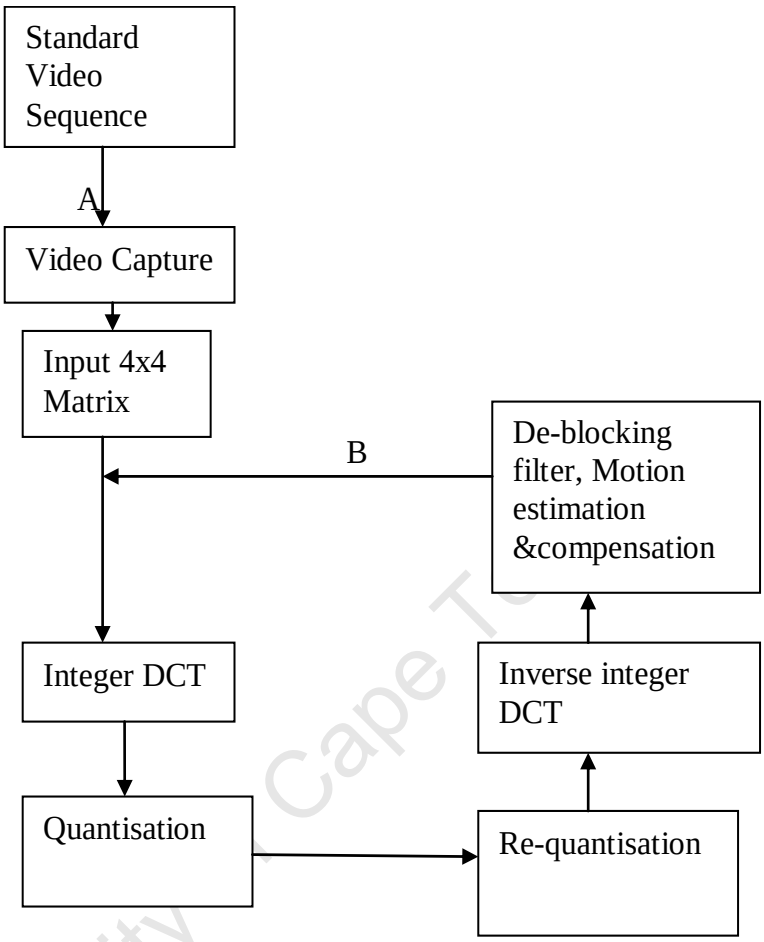


Figure 5.2: Experimental Approach

To achieve the objectives outlined in Chapter One, we set up the experiments as shown in Figure 5.2. We implemented the integer transforms (DCT and Hadamard) using two methods, 2-D matrix multiplication and 1-D butterfly in software as described in *section 5.2*. A couple of header files and source codes were written to realise the transform and quantisation processes, the main focus area in our experiments. The source codes and header files combines the integer DCT and quantisation processes. For the inverse processes another set of Header and source files were written for both the integer DCT and Hadamard transforms. The output of the inverse quantisation and inverse integer DCT is set to the motion estimation and compensation module via the de-blocking filter.

To measure the encode/decode time, the standard video was initially sampled spatially and temporally in 2-D 4x4 matrix format. Each pixel or picture element

(special-temporal sample) represents one or more numbers that describes the brightness (luminance) and colour (chrominance) [2]. The 4x4 integer transforms are then applied on the pixel arrays. The computational time includes the time taken to capture and convert the standard video sequence into a 4x4 matrix, the transform and quantisation, re-quantisation and inverse integer DCT, Filtering of the video and the motion estimation and compensation processes. From Figure 5.2, it is the period from point A to point B.

5.4 Two Dimensional (2-D)Matrix Multiplication Method

This method involved, in general, multiplying the integer transform matrix by the input matrix of a given video sequence. The resultant matrix was again multiplied by the transpose of the transform matrix.

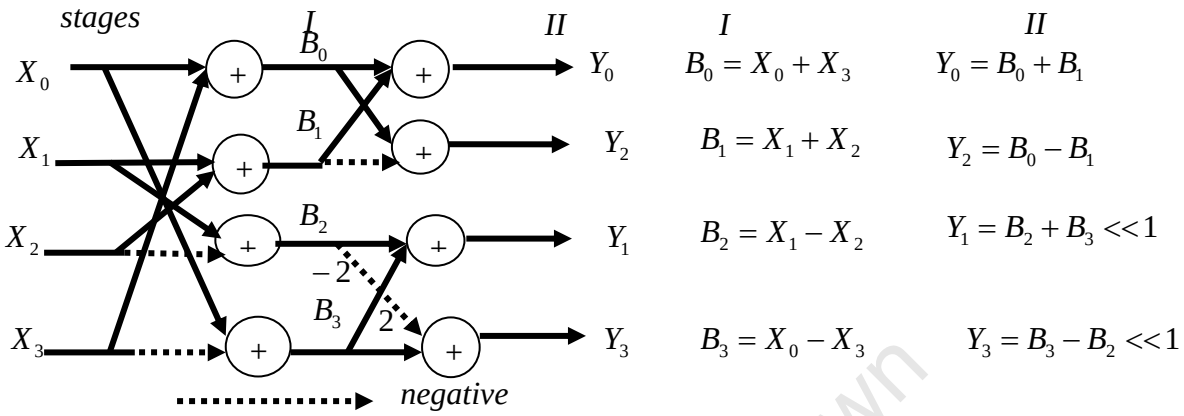
For the forward and inverse integer DCT transforms, equations (3.5) and (3.11) of Chapter Three were used. The integer DCT transform identity matrix C and the input matrix X are of the *short integer (16-bit)* data types. However, for the inverse integer DCT transform, the multiplication method meant multiplying *short data* types with *double data* type since the inverse identity matrix has 0.5s which are *floats* or *double data* type. They were multiplied according to C++ programming language as outlined in [44] [45]. Appendix C-3 shows the code snippet.

For the forward and inverse Hadamard transform, equations (3.12) and (3.13) were used for direct matrix multiplication. The Hadamard transforms have pure integer values and no floats or decimals. The input block is a 16-bit *short data* type, therefore, the identity Hadamard transform matrix was also declared as a 16-bit *short data* type. The Hadamard transforms use the same identity matrix for the inverse process as shown in equation (3.13). The two equations were also multiplied according to [44] [45].

5.5 One Dimension (1-D) Butterfly Algorithm Method

In this method the forward 2-D 4x4 integer DCT was computed in a separate way as a 1-D horizontal transform followed by a 1-D vertical transform. Figures 5.3(a) and (b) illustrate how the forward integer DCT was implemented in software. The butterfly algorithm has two stages; the first stage uses additions and subtraction while the

second stage combines additions, subtraction and left shift operations. This way we avoid the complex multiplication derived in equation 3.11 of chapter three.



(a) (b)

Figure 5.3: (a) Forward Butterfly Algorithm for the Integer DCT (b) Pseudo Codes for the Forward Integer DCT Implemented in Software

The separable property of the transforms has two advantages:

- The number of operations is reduced.
- The 1-D transform can be readily manipulated to streamline the operations further or to otherwise simplify calculations [1] [3] [9] [10] .

In addition, the above approach makes the implementation less complex.

The forward 4x4 Hadamard transforms can also be implemented in the similar manner as Figures 5.3(a) and (b) with the exception that the second stage will not constitute shift operations and the first stage remains the same as in the integer DCT.

The Butterfly technique can also be applied to the inverse integer DCT as illustrated in Figures 5.4 (c) and (d). It is essentially, the reverse of integer DCT in which case right shift operations are used instead of the left shift operations.

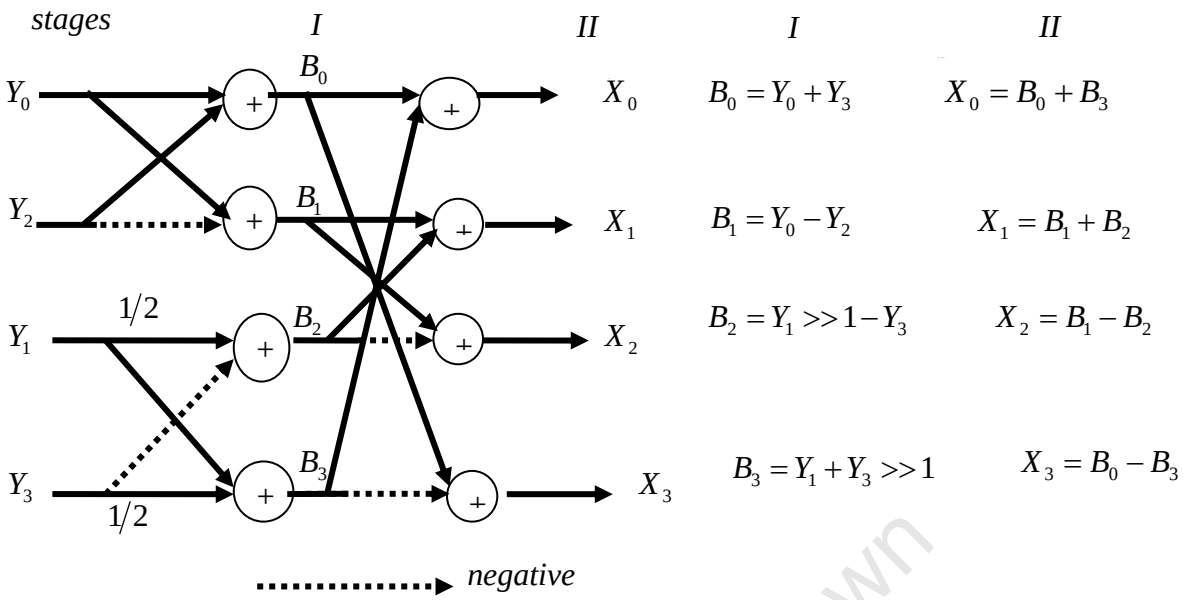


Figure 5.4(c) Inverse Butterfly Algorithm for the Integer DCT (d) **Pseudo Codes for the inverse Integer DCT Implemented in Software**

In like manner, the inverse 4x4 Hadamard transforms can be implemented as illustrated in Figure 5.4 (c) and (d) but with right shift operations removed.

In our experiments we implemented both the integer DCT and Hadamard transforms butterfly algorithms using:

- C++ only
- inline assembly and
- MMX technology instructions (SIMD)

5.5.1 Butterfly Algorithm with C++ Only

The algorithm and pseudo code of Figures 5.4 and 5.5 can be implemented in C++ with the iterative ***for statements*** consisting of four (4) loops. Loops have the advantage of reduced code size. However, small loops can be unrolled to increase performance.

```
short* block = (short *)ptr;
 for(j = 0; j < 16; j += 4)
        {

         C++ butterfly Codes here
        }//end for j...
```

In this particular approach and method, the number of loops is fewer compared to the direct 2-D Matrix multiplication. Only additions, subtraction and shift operations are used. The snippet codes have been given in Appendix D-2.

5.5.2 Butterfly Algorithm with SIMD Intrinsics

Intel's MMX technology intrinsics are high level instructions that can be used alongside the C++ compiled codes. They are classified as set, compare, shift, logical, arithmetic and general support as shown in Appendix B. SIMD Intrinsic can be used in a similar way as C++ except that all local variables must be declared as _m64 (For MMX instructions). Equally, the data to be manipulated using SIMD intrinsic must be converted into the appropriate SIMD data type by way of ***type casting.*** For example, a matrix declared as *short block [4] [4]* needs to be converted to *__m64*block1 = (__m64*) block* before application of intrinsic instructions. The code format for our SIMD intrinsics is shown below:

```
__declspec(align(16))short* block = (short *)ptr;
__declspec(align(16)) __m64*block1 = (__m64*)block;
__m64 s0,s1,s2,s3,f0,f1,f2,f3;
(
        MMX transpose
        MMX butterfly
        MMX transpose
        MMX butterfly
        )
```

Since the coefficients are kept in one packed word in one register, the 4x4 matrix is first transposed before applying the butterfly algorithm [25] [26]. The complete forward integer DCT transform process involves several steps as illustrated in Figure 5.5 [25] [26]:

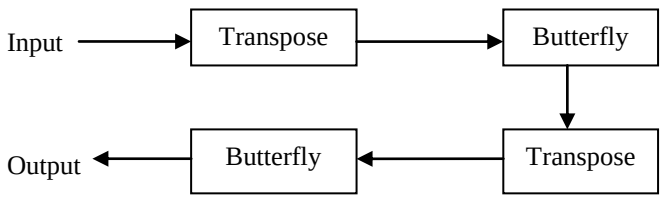


Figure 5.5: Forward Integer Transform Implementation with SIMD Instructions

a) Transposing the 4x4 matrix of the input residual video signal. In this process the rows of the 4x4 input matrixes are changed into columns. This is achieved by making use of four MMX conversion instructions: *_mm_unpacklo_pi16(row1,row2]),_mm_unpacklo_pi32(row0,row1),_mm_unpackhi_pi16(row1,row2])* and *_mm_unpackhi_pi32(row0,row1)* [37] [46]. For example, assuming a 4x4 input of:

$$X = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \text{ whose transpose } X^T = \begin{pmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{pmatrix}$$

The snippet codes for the transpose process are also shown in Appendix C.

b) Butterfly algorithm to the transposed input video signal in a). The butterfly algorithm has two stages as indicated in figure 5.3 (a) and (b). The first stage involves two additions and two subtraction of the transposed matrix using MMX intrinsic. Thus, *_mm_add_pi16 (row0, row3)* and*_mm_sub_pi16 (row0, row1)* were applied for addition and subtraction respectively [32] [47]. In the second stage, two additions, two subtraction as well as two shifts (such as _mm_slli_pi16 (row3, 1)) were used to arrive at the final value (see snippet codes in Appendix D-3).

c) Transpose of the resulting matrix in b) as in a)

d) Butterfly algorithm to the matrix in c) as in b)

The reverse of the above steps can be implemented for the inverse integer transform.

For the Hadamard transform, a similar implementation of SIMD is followed as for the integer DCT transform, except that no right or left shift operations are used

5.5.3 Butterfly Algorithm SIMD Inline Assembly

The procedure for implementing integer DCT and Hadamard transforms is the same as one outlined in *section 5.5.2* that is; transpose, butterfly, transpose and butterfly. The only difference lies in using assembly code equivalent of the SIMD intrinsic. Appendix C and Appendix D-3 also shows the equivalent SIMD assembly for each of the intrinsics discussed in *section 5.5.2.* However, unlike the intrinsic instructions which use a compiler, the SIMD assembly instructions use an assembler.

It is a low level programming set of mnemonics; close to machine codes. They can be used along with C++ codes by introducing an ASM as

__asm
{
SIMD MMX assembly codes
}

Four rows of four pixel values each, are loaded into the general register using the assembly MOVQ instruction. The MOVQ is also used for storing pixel values in registers. To transpose the 4x4 input block matrix, the PUNPCKHBW, PUNPCKLBW, PUNPCKHDQ and PUNPCKLDQ MMX instructions are used. The butterfly instructions are implemented using the PADDWA and PSUBW MMX arithmetic instructions along with the PSLLW and PSRAW MMX shift instructions.

To speed up our SIMD assembly Implementation further, we can substitute the load instruction *movq mm0, mm1* with a *pshufw mm4, mm4,0xE4.The movq* takes 6 cycles to load each element in the registers while the *pshufw* takes only 2 cycles [26]. This means that the *pshufw* is three times faster than the *movq* instruction when loading pixels in MMX registers.

5.5.4 Butterfly Algorithm with Inline Assembly in C++

The **assembly** statements in C/C++ programs or codes can be mixed using the ASM keyword to exploit the benefits outlined above. There are basically two categories or styles used in Assembly language: AT&T style and Intel Style. The major difference between the two styles lies in register naming, ordering of operands, operand size, indexing, etc. As our personal computer has an Intel processor we adopted the Intel style in our experiments. The Intel code syntax and style is shown below.

_asm
{
Assembly statements
}

5.6 Measurement Parameters

Our measurement parameters included the computation time, PSNR and Bits per pixel.

5.6.1 Computation Time

This is the time taken in computing the forward and inverse transform process in both the integer DCT and Hadamard transforms [24].Alternatively it is the time taken in seconds to encode and decode the video signal or sequence.

The computation time depends on a number of factors [1]:

- Number of frames- computation time increases as the number of frames increase. Thus a 300 frame video takes more computation time than a 100 frame video.
- Motion sequence-for high video motion sequence, computation time is more than in low video motion sequence at constant bit rate.
- Bitrate – the higher the bitrate the longer is the encode/decode time
- sequence details –a video sequence with more details takes more time to encode and decode
- complex movements- also tends to increase the computation time [3]

The percentage speed-up ($\%S_{up}$) is given by [21 [48]:

$$\%S_{up} = \frac{t_{referencce} - t_{av}}{t_{reference}} \times 100 \quad (5.1)$$

Where:

$t_{reference}$ is the average computation time of the original or reference codes in milliseconds

t_{av} is the average computation time for the new or optimised codes measured in milliseconds.

And the actual speed- up (S_{up}) was can be calculated from [21]:

$$S_{up} = \frac{t_{reference}}{t_{av}} \quad (5.2)$$

The variance (σ) can be calculated from:

$$\sigma = \left[\frac{1}{N} \Sigma (t_i - t_{av})^2 \right]^{1/2} \quad (5.3)$$

Where N is number of experimental trials and i=1,2,3,4.....

The Microsoft windows Operating System (OS), like other general purpose OSs, is multi-tasking and will switch to other tasks regularly while running the codes. Consequently, any timing measurements taken for both the reference and new codes will vary slightly from one experimental to another. The resolution of the accessible timer from which the timing tests are derived is also often limited and it often occurs that these precision errors accumulate. A number of experimental trials should be in order to collect a large time measurement that can be averaged to minimise these variations. In this research we did up to ten (10) trials per video sequence and then the average computation time per video sequence was computed.

5.6.2 Video Quality Measurements

Visual quality measuring is admittedly a difficult art. It can either be subjective or objective depending on methods adopted in measurements. The choice of the measurement methods depends on available tools, measurement bias, etc.

5.6.2.1 Subjective Quality Measurements

This criterion of visual quality measurement is never a complete measure of quality. It is influenced by several factors such as [1]:

- How the Human Visual System (HVS) interact with the eye and brain.
- How visible the video or image scene is. Is there any clear distortion? It is basically a question of spatial fidelity.
- How the motion appears such as whether they are natural and smooth ie temporal fidelity.

Additionally factors such as [1] a passive watch of a DVD movie scene, live videoconference talk, sign language communication or identifying a person using video scenes from a surveillance camera can affect the opinion of the viewer on visual quality.

5.6.2.2 Objective Quality Measurement

This is the quantitive visual quality measuring criteria. It has the advantage that the results can be repeated and therefore offers reliable and believable results [1]. The Objective quality measurements do not, however, give subjective human satisfactions whose viewing inclinations and bias may differ [6]. A commonly used

example of objective video quality measurements is the Peak Signal to Noise Ratio (PSNR). PSNR measurements have these additional weaknesses [1]:

- It requires a perfect copy of the original video or image to compare which may not be available. Verification of the original video or image is also not easy.
- It lacks correlation with other subjective video quality measures such as the ITU-R 500 standard measurements. The ITU-R500 uses a common test procedure called Double Stimulus Continuous Quality Scale (DSCQS) in which two video sequences, the original and compressed are randomly tested and graded from excellent to bad.
- While a high PSNR indicates high quality, it does not always give true subjective quality measurement.

We adopted this (PSNR) quality measurement in our experiments

5.7 Assumptions and Constants

The focus in our experiments was determine the computation times for each of the methods outlined above and compute the speed-up improvement, taking the matrix multiplication method as a reference. We essentially, sought to know which method was faster and how fast it is. The PSNR, bits per pixel measurements were taken to be constant and were therefore kept exactly the same in each method.

5.8 Chapter Summary

This chapter has described the two methods that were used in experiments-the 2-D matrix multiplication and 1-D butterfly algorithm. The butterfly algorithm was implemented using C++, SIMD instruction (intrinsic and assembly) and inline assembly while the matrix multiplication implemented using C++ only. We have also discussed the measurement parameters considered in experiments. The results obtained in experiments and their analyses are given in the next chapter.

Chapter Six

6 Results and Analysis

6.1 Chapter Introduction

This chapter presents the results on the computation time or the encode/decode time and the PSNR as measured in the experiments. The measured results for 2-D matrix multiplication and 1-D butterfly methods have been recorded in tables, analysed and discussed. Graphs of the PSNR, Bits per pixel (Bpp) versus number of frames, for the three video sequences, have also been plotted and analysed.

6.2 Computation Time Measurements

Tables 6.1 through to 6.7 shows the measured results for the computation time in milliseconds for the three standard video signals using 2-D Matrix Multiplication and 1-D Butterfly methods. The percentage speed-up in the computation time, on each of video sequence and optimised methods, has been calculated based on equation (5.1) in *section 5.6.1*. The Matrix Multiplication method has been taken as a reference.

6.2.1 Computational Time For the 2-D Matrix Multiplication Method

Table 6.1, is the results obtained for the forward and inverse integer DCT transforms using Matrix Multiplication. The results are based on C++ multiplication as described in *section 5.4* in Chapter Five. The *foreman* and *soccer* video sequence had an average computational time of 36965 and 36217 respectively while *mobile & calendar* had 42887ms. The difference in computation times between each video sequence is attributed to differences in sequence details, bitrate (Bits per pixel as stated in Chapter Five, *section 5.5*. Notwithstanding the above factors, which are considered to be constant in every method, we also noted that the computational times are generally high in this method. This means that video compression was slow for all the three sequences. We attributed such high computational time (slow compression) to numerous loops or iterative statements requirement in this method. As shown in Appendix C-1, a number of ***for statements (loops)*** are used to multiply equation

(3.12) for the forward integer DCT and equation (3.18) for the inverse integer DCT transform. These loops require a considerable amount of time to be executed in a computer programme.

Table 6.1 Computational Time for the Forward and Inverse 4x4 Integer DCT Transform using 2-D Matrix Multiplication Method

Experimental trials	Encode/decode time in milliseconds(ms)		
	Soccer	**Foreman**	**Mobile & Calendar**
T0	36175	36972	42875
T1	36212	36962	42894
T2	36225	36964	42874
T3	36218	36976	42900
T4	36224	36948	42872
T5	36221	36959	42885
T6	36222	36965	42899
T7	36232	36971	42900
T8	36218	36969	42877
T9	36223	36967	42896
AVERAGES	**36217**	**36965**	**42887**
VARIANCE	**14.8526**	**7.4899**	**11.1982**

The 2-D Matrix Multiplication method can be applied for the forward and inverse Hadamard transforms in similar way as in the forward and inverse integer DCT. The difference, largely, lies in the type of the identity Hadamard transform matrix used in each transform. Equations (3.30) and (3.33) were used for the forward and inverse 2-D Matrix Multiplication respectively. The results are indicated in Table

6.2. As observed in the inter DCT, matrix multiplication in Hadamard transform equally uses numerous loops which also makes this method slow.

Table 6.2: Computational Time for the Forward and Inverse 4x4 Hadamard Transforms using 2-D Matrix Multiplication Method

Experimental trials	Encode/decode time in milliseconds(ms)		
	Soccer	**Foreman**	**Mobile & Calendar**
T0	36853	37068	43055
T1	36850	37058	43054
T2	36853	37060	43049
T3	36846	37072	43060
T4	36852	37044	43052
T5	36849	37061	43048
T6	36850	37061	43059
T7	36860	37067	43060
T8	36846	37065	43057
T9	36851	37063	43056
AVERAGES	**36851**	**37062**	**43055**
VARIANCE	**3.8209**	**7.1903**	**4.0743**

The computational time measurements for the Hadamard transform are similar to those of the integer DCT. The matrix multiplication process and ***for loops*** involved accounts for low speed-up obtained in this method.

6.2.2 Computational Time For The 1-D Butterfly Method

Table 6.3: Computation Time for the Forward and Inverse 4x4 Integer DCT Transform using 1-D Butterfly Algorithm Method-C++ Only

Experimental trials	Encode/decode time in milliseconds(ms)		
	Soccer	**Foreman**	**Mobile & Calendar**
T0	21341	22121	27033
T1	21301	22114	26941
T2	21356	22116	26994
T3	21368	22194	26990
T4	21387	22161	26975
T5	21378	22164	27017
T6	21357	22164	26972
T7	21387	22134	26984
T8	21377	22144	26993
T9	21367	22124	26986
AVERAGES	**21362**	**22144**	**26989**
VARIANCE	**24.5254**	**25.1038**	**23.6324**

Table 6.3, above, shows the results for the integer DCT transforms using 1-D Butterfly method with C++ as programming language. In this set of experiments, the butterfly method gives an average of 14000 ms time gain when encoding and decoding each of the video sequences representing a 40% speed-up in comparison to the 2-D Matrix Multiplication method. The speed-up gains can be attributed to a number of reasons such as reduction in the use of iterative control statements (loops), avoidance of the complicated additions, subtractions and shifts operations, compared

to the matrix method. Additionally, codes written in this method are also easy to debug, small in size and portable across different platforms.

Table 6.4: Computation Time for the Forward and Inverse 4x4 Integer DCT Transform using 1-D Butterfly Algorithm Method-Intrinsic Only

Experimental trials	Encode/decode time in milliseconds(ms)		
	Soccer	**Foreman**	**Mobile & Calendar**
T0	21245	21962	26870
T1	21282	21961	26864
T2	21261	21975	26863
T3	21264	21978	26851
T4	21257	21965	26864
T5	21267	21969	26861
T6	21261	21950	26872
T7	21253	21974	26865
T8	21258	21971	26867
T9	21267	21968	26868
AVERAGES	**21262**	**21967**	**26865**
VARIANCE	**9.3113**	**7.8804**	**3.8079**

Table 6.4 shows the results obtained when using SIMD MMX intrinsic. The SIMD intrinsic codes have a superior speed-up advantage in comparison to the codes written using the Matrix Multiplication method. When compared with those obtained in table 6.3 (C++ only for butterfly), the speed-up advantage is significantly lower and does not immediately show the merits of using intrinsics. While the intrisics have eliminated the use of loop statements and have enabled parallel manipulation of

pixels, a close look at the compiled codes and the generated assembly codes reveals the following weaknesses:

- ***Use of type casting*** -the type casting of input matrix block from short to __m64 Data type consumes quite some significant time and this tend to slow the codes.

- ***Limited use of MMX registers***-the assembly generated by the intrinsic shows that only two registers (mm0, mm1) are used during transposing and butterfly manipulation of pixels. The other registers such as mm2, mm3, mm4, mm5, mm6, mm7, etc are underutilised.

Irrespective of the above weakness, this method can be used in all scenarios where the C++ only Butterfly method is used as the speed-up measurements of the two are almost equal. Further, as the intrinsics are compiled codes, they are a high level programming language and like C++, they are portable and easy to maintain.

To mitigate the weakness observed in the SIMD intrinsics we converted the intrinsic code to inline assembly. These were hand written codes to realise the transpose and butterfly algorithm. Table 6.5 shows the results obtained. From the results it can be observed that the SIMD assembly outperforms both the C++ only and the SIMD intrinsics by an average percentage speed-up of 10% and is 45% faster than the matrix multiplication method. This is attributed to a number of factors such as:

- Efficient use of MMX registers

- Parallel manipulation of pixels

- Non-use of casting operations

- Non-use of loop statements such as for statement

- Inherent advantage of Low level programming which is close to machine code

Table 6.5: Computation Time for the Forward and Inverse 4x4 Integer DCT Transform using 1-D Butterfly Algorithm Method-SIMD Assembly only

Experimental trials	Encode/decode time in milliseconds(ms)		
	Soccer	**Foreman**	**Mobile & Calendar**
T0	19243	20057	24764
T1	19237	20003	24788
T2	19231	20034	24746
T3	19238	20061	24781
T4	19249	20081	24783
T5	19239	20078	24774
T6	19216	20070	24804
T7	19224	20063	24774
T8	19236	20054	24801
T9	19238	20034	24787
AVERAGES	**19235**	**20054**	**24780**
VARIANCE	**8.9275**	**22.5853**	**15.1029**

Table 6.6 shows the computation time measurements using the 1-D Butterfly algorithm for the integer DCT using inline Assembly of the C++ codes. The results reveal a speed-up improvement of 44% and 8% in comparison to the Matrix Multiplication and the Butterfly method (C++ only and Intrinsic) respectively. This was attributed to:

- Low level programming which is inherently faster than high level compiled codes

However, this method could not outperform the SIMD inline assembly owing to among others, serial or sequential manipulation of pixels and long assembly codes which tend to use up most of the memory.

Table 6.6: Computation Time for the Forward and Inverse 4x4 Integer DCT Transform using 1-D Butterfly Algorithm Method-Inline Assembly

Experimental Trials	Encode/decode time in milliseconds(ms)		
	Soccer	**Foreman**	**Mobile & Calendar**
T0	20043	21730	25756
T1	20069	21739	25721
T2	20026	21749	25760
T3	20032	21721	25775
T4	20039	21756	25745
T5	20045	21732	25741
T6	20056	21745	25735
T7	20054	21746	25730
T8	20047	21749	25746
T9	20036	21751	25754
AVERAGES	**20045**	**21742**	**25746**
VARIANCE	**11.9708**	**10.4594**	**14.9298**

The Hadamard transforms results, for the three 1-D butterfly methods, are shown in Table 6.7. Like in the integer DCT, the 1-D butterfly method outperforms the matrix multiplication method in terms of the percentage computational time speed-up. We obtained computational speed-ups of 41%, 42, 45% and 47% using C++, SIMD intrinsic, Inline assembly and SIMD assembly instructions respectively,

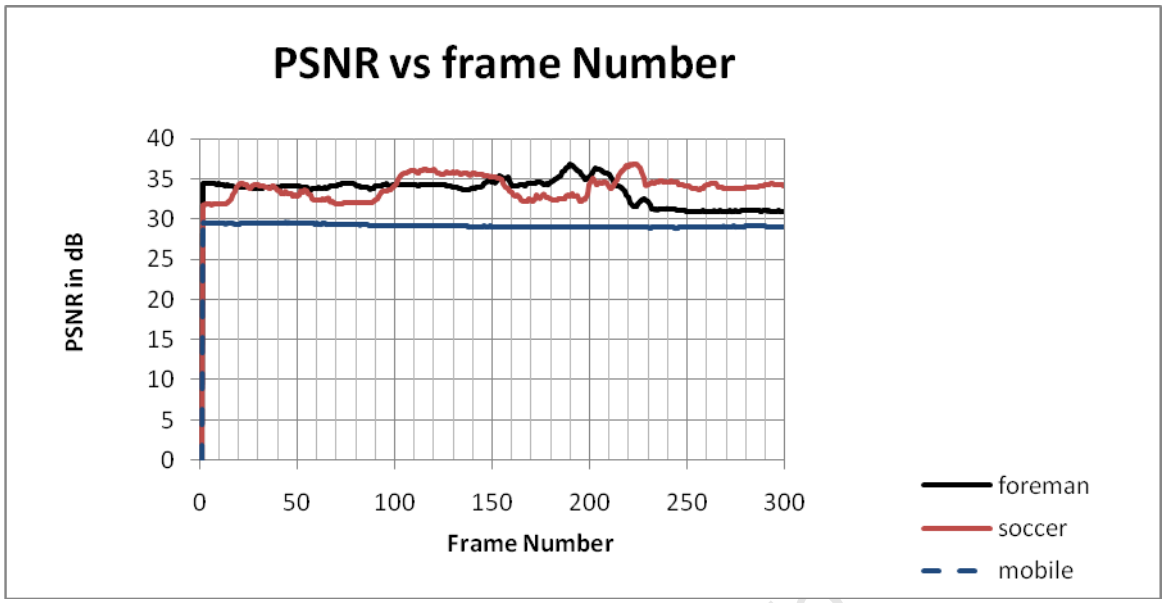
the matrix multiplication method being used as a reference. The Hadamard transform was about 2% faster than the integer DCT when the two transforms were compared. This is due to non-use of shifts and less complicated quantisation process for the DC components.

Table 6.7: Computation Time for the Forward and Inverse 4x4 Hadamard Transform using 1-D Butterfly Algorithm Methods

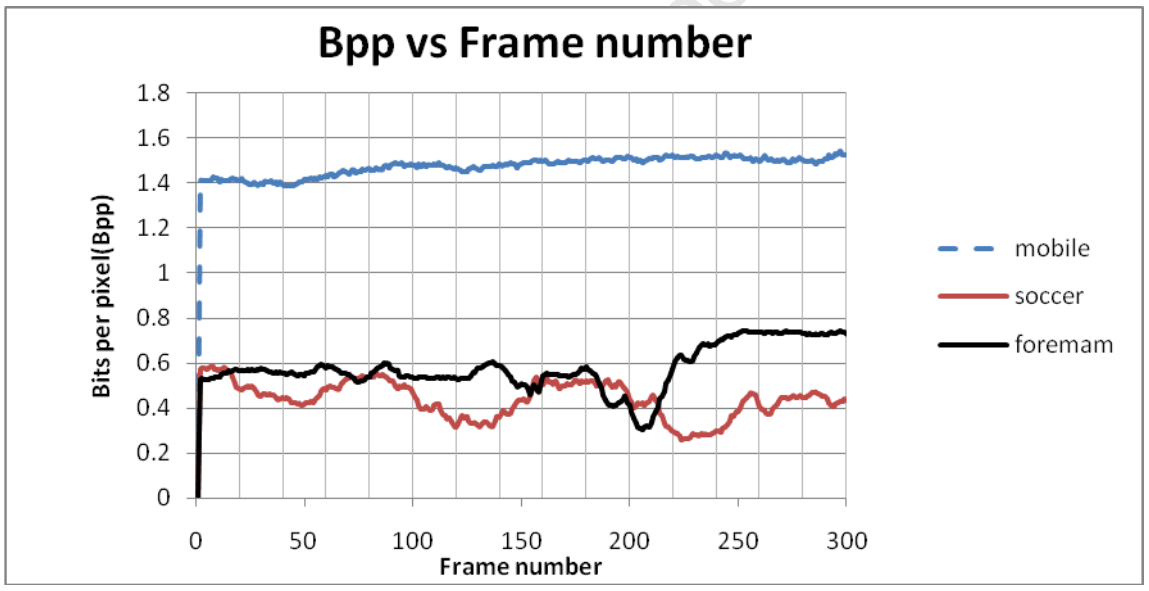
Standard Video	Programming environment	Encode/decode time in milliseconds										
		T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	Average time
Foreman	C++	22064	22081	22066	22024	22037	22060	22064	22047	22058	22070	22057
	Assembly	20803	20834	20841	20813	20807	20805	20851	20843	20825	20832	20825
	Intrinsics	21850	21845	21851	21856	21887	21890	21845	21857	21867	21849	21860
	SIMD	19891	19888	19919	19899	19904	19916	19918	19909	19911	19907	19906
Soccer	C++	21169	21128	21158	21134	21141	21151	21137	21156	21154	21143	21147
	Assembly	19988	19980	19956	19887	19956	19964	19939	19967	19965	19951	19955
	Intrinsics	21029	21079	21065	21057	21101	21100	21026	21033	21072	21088	21065
	SIMD	19007	19099	19088	19092	19105	19101	19106	19103	19102	19088	19089
Mobile	C++	26794	26791	27740	26735	26751	26723	26768	26779	26767	26771	26762
	Assembly	25707	25721	25719	25731	25723	25725	25738	25745	25745	25756	25731
	Intrinsics	26634	26641	26652	26645	26637	26654	26639	26678	26657	26678	26652
	SIMD	24684	24650	24635	24627	24668	24629	24634	24645	24636	24631	24644

The computational time speed-ups obtained for butterfly method, in both the integer DCT and Hadamard transforms, are due to reduced complexities introduced in the video encoder owing to the use of additions, subtractions and shifts.

6.3 Peak Signal to Noise Ratio Measurements



(a)



(b)

Figure 6.1: (a) plot of PSNR versus Number of Frames for the three test video sequence (b) plot of Bpp versus Number of Frames measured simultaneously with the PSNR

Figure 6.1 (a) and (b) are graphs of the PSNR and Bpp versus number of frames during encode/decode period for each of the video sequences. For Figure 6.1 (a), *soccer* has the higher average PSNR than *Foreman* and *Mobile & Calendar*. The first seventy-five (75) frames of *Foreman* have a higher PSNR than *Soccer* whose last

first seventy-five (75) frames are higher than *Foreman* and *Mobile* & *Calendar*. *Mobile & Calendar* has the least PSNR of the three. In Figure 6.1(b) the bitrate or Bpp values are inversely proportional to the PSNR of Figure 6.1(a).The *Mobile & Calendar* video sequence has a higher bitrate than *Soccer* and *Foreman.*

There is also a close relationship between Bpp, and PSNR, and computation time. Computational time decrease with increase in PSNR or decrease in Bpp. For example in Figure 6.1(b), the *Mobile* & *Calendar* video sequence has the highest bits per pixel and when this is compared with tables 6.2, 6.3, 6.4 and 6.5, *Mobile* & *Calendar* video sequence has the highest computational time. The same trend was observed for *Foreman* and *Soccer* video sequences with the later having the smallest bitrate and sequence details and therefore the fastest.

6.4 Chapter Summary

This Chapter has given and discussed the results for the Matrix Multiplication and Butterfly methods as it relates to computation time, PSNR and bit per pixel measurements. From the results the integer DCT Butterfly method is on average 40% faster than the Matrix Multiplication method. Similarly, the Hadamard transforms with butterfly method is 45%, on average, faster than the reference Matrix Multiplication method. Four Butterfly methods, C++ only, SIMD Intrinsic, SIMD inline assembly and inline assembly, were implemented. Of the four, SIMD inline assembly gave superior speed-up.

Chapter Seven

7 Conclusion And Recommendations

7.1 Chapter Introduction

This Chapter gives the conclusion of the thesis arising from the experimental approach, results and general work done on integer DCT and Hadamard transforms.

7.2 Conclusions

The need for video compression is inevitable considering the limited storage and transmission bandwidth. A number of video coding standards have been developed to compress videos and or images for various applications. Some of these standards are H.261, H.262, MPEG-1, MPEG-2, MPEG-4, JPEG, JPEG 2000 and the recent H.264/AVC. Each of these standards adopts a transform for removing special redundancy in the video or image such as the DCT and DWT. The H.264 /AVC standards adopt the integer DCT and Hadamard transform coding methods for removing redundancy in the image and /or video signals.

The principle objective of this thesis was to implement the 4x4 integer DCT and 4x4 Hadamard transforms using 2-D matrix multiplication and 1-D butterfly algorithm methods in software. In each case we ascertained the speed-up, in the computation time at constant PSNR and Bpp for each of the three video sequences. The programming environment was C++ alongside with inline assembly and Intel's SIMD instructions.

With the matrix multiplication method as our reference, the 1-D butterfly algorithm for the integer DCT using C++ only and SIMD intrinsic instruction was 40% and 40.5% respectively faster than the matrix multiplication algorithm. A further 44% speed-up was obtained when using inline assembly mnemonic codes. A 45% speed-up was obtained when using 1-D butterfly algorithm with SIMD assembly instructions over the matrix multiplication method. Similarly, results obtained for the Hadamard transform using butterfly method showed a 41%, 42%, 45%and 47% speed-up with C++, intrinsic instructions, inline assembly and SIMD assembly respectively. The 1-D butterfly algorithm therefore gave better optimised results than

the matrix methods, all testing parameters (PSNR and Bpp) being equal. These results have confirmed our hypothesis which was "The 2-D 4x4 integer and Hadamard transforms implemented in a separable way as 1-D add, subtract and shift operations either in the horizontal followed by the vertical direction or vice versa is computationally faster than matrix multiplication"

Another, important, conclusion to make is that the SIMD intrinsic instructions, like C++ codes, have few code sizes but do not make efficient use of the MMX registers. This makes them less fast than the inline SIMD assembly codes.

1-D butterfly algorithm results were faster than 2-D matrix multiplication algorithm for both the integer DCT and Hadamard transforms.

7.3 Recommendations

While the 4x4 block size integer and Hadamard transforms makes good use of SIMD instructions of the MMX Technology type, its use on SSE2 instructions is discouraged. This block size does not use the 128-bit SSE2 registers efficiently as it causes truncation errors and PSNR turns out to be poor. However, the concepts and principle used on the 4x4 block size can be extended to:

- The 8x8 block size used in the high profiles in which case either SSE 2 or SSE, depending on whether the input matrix is a **float** or **integer** type**,** instead of MMX instructions can be used.

- Other modules such as motion estimation can also benefit from SIMD implementation. For instance in interpolation (such as half pixel, quarter pixel [49] and linear interpolation of B frames [31]).

The quantisation processes can also be implemented using SSE2 or SSE as it involves large value multiplication of integers which could not be performed by MMX technology instructions. However, we noticed that it is not possible to mix SSE2 instruction with MMX technology in one function.

The speed-ups obtained in the 1-D butterfly algorithm is best suited for use in low bandwidth and/or internet broadcasting applications including other real-time Digital Signal Processing (DSP) such as video conferencing, radar and sonar (target detection) and electrocardiogram (biomedical) [50]. The 2-D matrix multiplication

method, on the other hand, can be used in non-real time DSP applications such as image retrieval from data bases, music play back and manipulation (mixing and special effects) [50] and for education purposes.

Chapter Summary

The conclusions on Fast Implementation of 4x4 Integer Transforms (integer DCT and Hadamard) have been presented and the research questions answered. We also managed to verify our research objectives and suggested industrial applications for each method. Considering the benefits derived in using SIMD instructions, our results can be adopted for the stated applications. The given recommendations on future work, when adopted too, are expected to give similar or better speed-up in the stated modules.

References

[1] I. E. G Richardson, H.264 *and MPEG-4 Video Compression,* John Wiley and Sons, West Sussex, UK, 2003.

[2] I. E. G Richardson, *The H.264 Advanced Video Compression Standard*, John Wiley and Sons, 2nd edition, West Sussex, UK, 2010

[3] H.264: International Telecommunication Union, *Recommendation ITU-T H.264: Advanced Video Coding for Generic Audiovisual Services*, ITU-T, 2003.

[4] I. E. G Richardson, *Video Codec Design, Developing Image and Video Compression Systems,* Published by John Wiley and sons, West Sussex, UK, 2002.

[5] M. Ghanbari, *Standard Codecs: Image Compression to Advanced Video Coding*, Hertz, UK, 2003.

[6] H. Kalva, J. B. Lee, *The VC-1 and H.264 Video Compression Standards for Broadband Video Services*, Springer, New York, USA, 2008.

[7] T. Wiegand, G.J. Sullivan, G. Bjontegaard and A.Luthar, "Overview Of The H.264/AVC Video Coding Standard," *IEEE Transactions on circuits and systems for video technology*, vol. 13, No 7, July, 2003, pp. 598-603.

[8] H. S. Malvar, A. Hallapuro, M. Karczewicz and L. Kerofsky, "Low Complexity Transform and Quantisation in H.264/AVC*," IEEE Transactions on circuits and systems for video technology*, vol. 13, No 7, July, 2003, pp. 560-576.

[9] M. M. Ghandi and G. Mohammed, "The H.264/AVC Video Coding Standard for the Next Generation Multimedia Communication," *Journal of Iranian Association of Electrical and Electronics Engineers*, vol. 1, No 2, 2004, pp. 1-10.

[10] T. Wiegand and G. J. Sullivan, "The H.264/AVC Video Coding Standard," *IEEE Signal Processing Magazine*, March 2007, pp. 148-153.

[11] H.262: International Telecommunication Union, *Recommendation ITU-T H.262: Information technology-generic coding of moving pictures and associated audio information: video,* ITU-T, 1995.

[12] H.263: International Telecommunication Union, *Recommendation ITU-T H.263: Video Coding for Low Bit Rate Communication*, ITU-T, 1998.

[13] MPEG-2: ISO/IEC JTC1/SC29/WG11 and ITU-T, *ISO/IEC 13818-2: Information Technology-Generic Coding of Moving Pictures and Associated Audio Information: Video*, ISO/IEC and ITU-T, 1994

[14] MPEG-4: ISO/IEC JTCI/SC29/WG11, *ISO/IEC 14 496:2000-2: Information on Technology-Coding of Audio-Visual Objects-Part 2: Visual*, ISO/IEC, 2000.

[15] C. P. Fan, "Fast 2-Dimentional 4x4 Forward Integer Transform Implementation for H.264/AVC," *IEEE Transactions on circuits and systems II: Express briefs*, vol. 53, Issue 3, March, 2006, pp. 174-177.

[16] H. D. Tiwari, G. Gankhuyag, G. H Kim, C. M. Kim and Y. B. Cho, "Multiplier Less Fast Loss Less Integer DCT for H.264," IEEE *International SoC Design conference*, 2008, pp. 342-345.

[17] S. Kwon, A. Tamhankar and K. R. Rao, "Overview of H.264/MPEG- 4 part 10," *Journal of Visual Communication and Image Representation* vol. 17, Issue 2, April 2006, pp. 186-216

[18] I. E.G Richardson, (2009, April). 4x4 *Transform and Quantization in H.264/AVC*. [On-Line]. Available: http://www.vcodex.com/h264transform4x4.html [August 20, 2009].

[19] Wikipedia, (2011, February). *Hadamard transform*. [On-line]. Available: http://en.wikipedia.org/wiki/Hadamard_transform [May 12, 2010].

[20] J. Xiuhua, Z. Caiming and W. Yanling , *Fast Algorithm of the 2-D 4x4 Inverse Integer Transform for H.264/AVC*, IEEE Innovative Computing, Information and Control International Conference(ICICIC), 2007, pp. 144-148.

[21] Y. Muhammed, K. E. Khan and M. S. Beg, "Performance Evaluation Of 4x4 DCT Algorithms for Low Power Wireless Applications," *First International Conference on Emerging Trends in Engineering and Technology*, 2008

[22] L. Haiyan, C. Zhang, J. Ren, L. Li and D. Liu, "Stream Algorithm for 4*4 Integer Transform in H.264," *IEEE International Conference on Multimedia and Ubiquitous Engineering*, 2007, pp. 1106-1111.

[23] Y. Wang, Y. Zhou and H. Yang, "Early Detection Method of All-Zero Integer Transform Coefficients," *IEEE Transactions on Consumer Electronics*, vol. 50, No. 3, August, 2004, pp. 923-928.

[24] Y. Chen, E. Q. Li, X. Zhou 1 and Steven Ge, "Implementation of H.264 Encoder and Decoder **On** Personal Computers*," Journal of Visual Communication and Image Representation,* 2006, pp. 509-532.

[25] Y. Jianhong and L. Jilin, "Fast Parallel Implementation of H.264 /AVC Transform Exploiting SIMD Instructions," *Proceedings of 2007 International Symposium on Intelligent Signal Processing and Communication Systems*, Xiamen, China; November 28-December 1, 2007. pp. 870-873.

[26] L. Xueming and F. WEI, "An Improved Practical Efficient Implementation of ICT Used in H.264," *IEEE international conference on multimedia*, 2004, pp. 1163-1166.

[27] UB Video Inc, (2002). *Emerging H.264 Standard: Overview and TMS320C64x Digital Media Platform Implementation (White paper).* [On-line]. Available: http://focus.ti.com/pdfs/vf/vidimg/ubvideo_h26l_whitepaper_v24.pdf [May12, 2011].

[28] J. Golston and A. Rao, (2006). *Video codecs tutorial: Trade-offs with H.264, VC-1 and other advanced codecs*. Texas Instruments. [On-line]. Available: http://i.cmpnet.com/videsignline/2006/03/ti-5.jpg [July13, 2010].

[29] S.Valia and S. Jamkar. *Performance Enhancement of Video Compression Algorithms with SIMD,* [On-line], Available: http://homepages.cae.wisc.edu/~ece734/project/s04/valia.doc [October 23, 2010].

[30] J. Osterman, J. Bormans, P. List, D. Marpe, M. Narroschke and F. Pereira, "Video Coding With H.264/AVC; Tools, Performance and Complexities," *IEEE Circuits and Systems,* August, 2004, pp. 7-28

[31] T. Bhatia, (2010, December) *Optimization of H.264 High Profile Decoder for Pentium 4 Processor.* [On-line], vol.(issue), available: http://www-ee.uta.edu/Dip/Courses/EE5359/.../bhatiaslides.ppt' [May 12, 2011].

[32] Intel®, (1999). *Architecture Optimization reference Manual.* [On-line], (245127-001). Available: http://download.intel.com/design/PentiumII/manuals/24512701.pdf [October 20, 2010].

[33] Intel®, (2007). *C++ Intrinsics Reference*. [On-line], (312482-003US). Available: http://www.info.univ-angers.fr/pub/richer/ens/l3info/ao/intel_intrinsics.pdf http://www.daimi.au.dk/~zxr/papers/sse_intrinsic_reference.pdf [March 20, 2010].

[34] Intel®, (2007). *C++ Intrinsics Reference*. [On-line]. Available: http://cachewww.intel.com/cd/00/00/34/76/347603_347603.pdf [March 21, 2010].

[35] S. Krishnaprasad, "SIMD Programming Illustrated Using Intel's MMX Instruction Set," *Journal of Computing Sciences in Colleges (JCSC*), vol. 19, issue 3, January, 2004, pp. 268-277

[36] F. Franchetti, S. Kral, J. Lorenz and C. Ueberhuber, "Efficient Utilization of SIMD Extensions," *Proceedings of the IEEE*, vol. 93, No. 2, February, 2005, pp. 409-425,

[37] M. Hassaballah, S. Omran and Y. B. Mahdy, "A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications," *The Computer Journal*, vol.51, issue 6, 2008, pp. 630-649.

[38] A. Ortiz, "Teaching the SIMD execution model: assembling a few parallel programming skills," *Proceeding of 34 Technical Symposium on Computer Science Education*; February 19–23, 2003, pp. 74-78,

[39] A. Ortiz, (1999, May). *An overview of Intel's MMX technology*. Linux Journal, [On-line], (61), Available: http://www.linuxjournal.com/article/3244 [November21, 2010].

[40] S. Oberman, G. Favor and F. Weber, "AMD 3dnow! Technology Architecture and Implementations," *IEEE Micro*, Match, 1999, pp. 37-48

[41] P. K Jain, (2006, October). *Using Inline Assembly in C/C++*. [On-line], Available: http://www.codeproject.com/KB/cpp/edujini_inline_asm.aspx. [October 20, 2010].

[42] S. W Copen, (2005, January). *Optimizing C/C++ with Inline Assembly Programming*. [On-line]. Available: http://drdobbs.com/architecture-and_design/184401967?pgno=1. [November 25, 2010].

[43] K. Nguyen, "Optimisation of the Intel Pentium 4 Processor using Assembly Languages," *Intel Corporation*, 20th October, 2008

[44] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs And Designs*, 3rd Edition, Dorling Kindersley, New Delhi, India.

[45] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Special Edition, New Jersey, USA.

[46] F. R. Alex, (2003, July). *Introduction to SSE Programming*. [On-line]. Available: http://www.codeproject.com/KB/recipes/sseintro.aspx **[**November 24, 2009**].**

[47] Intel's, (1997). *Intel ArchitectureOptimization Manual*. [On-line], (242816-003). Available: http://download.intel.com/design/pentiumii/manuals/24281603.pdf [October 19, 2010].

[48] H. Wang, S. Kwong and C. W. Kok, "Efficient Prediction Algorithm Of Integer DCT Coefficients For H.264/AVC Optimisation," *IEEE transaction on circuits and systems for video technology*, vol. 16, NO.4, April 2006, pp. 547-552.

[49] S. Pigeon and S. Coulombe, "Speeding Up Motion Estimation in Modern Video Encoders Using Approximate Metrics and SIMD Processors" *IEEE Symposium on Industrial Electronics and Applications* (ISIEA), October 2009, pp.233-238.

[50] E. Punskaya,(2008). *Introduction to Digital Signal Processing (DSP*). [On-line]. Available: http://www-sigproc.eng.cam.ac.uk/~op205/3F3_1_Introduction_to_DSP.pdf [November 15, 2010]

APPENDICES

Appendix A: Evolution of Video compression standards

The evolution of video coding standard from 1984 to 2004 may be divided into the work of the MPEG of International Standard Organisation/ International Electrotechnical Commission (normally stated as ISO/IEC) and the International Telecommunication Union - Telecommunication sector (ITU-T)'s Video Coding Experts Group (VCEG) or a joint ITU-T/MPEG. The chart in Figure 1.3 illustrates how video coding standards have evolved over the years.

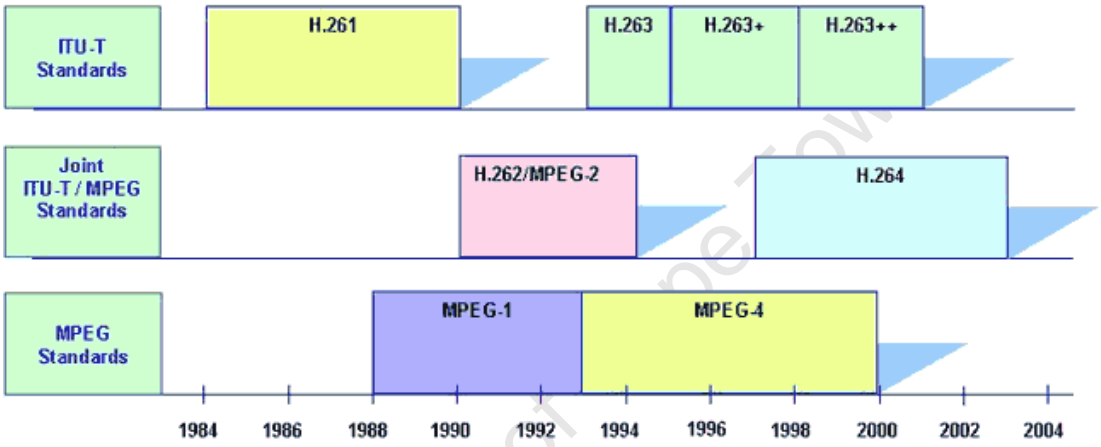


Figure 1.3: Evolution of video coding standards[5][9][27]

The ITU-T Video Compression Standards

The H.261 video coding standard was developed by the ITU-T in the period between 1984 to about 1990. The standard was developed for videoconferencing application [1] [2] [5]. It has a typical operating bit rate of between 64 and 384 kbps [2] [4]. The H.261 encoder uses discrete cosine transforms as a transform coding method. At the time of its development hardware and software processing was limited, which means that it's less complex and this is its primary advantage. Notwithstanding, it lacks flexibility and has poor compression performance especially for bit rate below 100kbps [2] [4].

To improve on the weaknesses of the H.261 standard, the H.263 video coding standard was developed in 1995. It has the advantage of good and high compression efficiency, and consequently improved processing performance [2] [12] [4]. It also provides greater flexibility than the H.261 standard: for example, a wider range of frame size is supported such as 16CIF (1408X1152), 4CIF (704X576), CIF (352X288), QCIF (176X144) and subQCIF (128x96) [2] [4]. It also uses discrete cosine transforms as transform coding. H.263 has had its primary application in video communication (two-way) of low bit rate and low delay. It can support bit rate as low as 20kbps and is therefore a popular standard in video telephony and video conference applications [2] [12] [4].Four optional modes were introduced and each of those modes is described in the annex to this standard on the first version. More options were added to the later version, version two (also called H.263+ standard), of 1998.

ISO/IEC video compression standards

While the ITU-T was developing standards H.261 through to H.263, the ISO/IEC too has been developing image and video coding standards; almost in tandem with the ITU-T. Their standards are primarily aimed at supporting storage and distributed applications [13]. The two ISO groups; the MPEG and the JPEG and developed standards MPEG-1, MPEG-4, JPEG and JPEG 2000.

In 1993 the MPEG developed the MPEG-1 standard. It was intended and designed for compression of video and audio for storage and real time play back CD-ROM, at a bit rate of 1.4 Mbps with the quality that is comparable to the VHS video tapes [2] [4]. In 1998 ISO/IEC improved their MPEG-1 to MPEG-4 for video and audio compression and transport for multimedia terminals. It supports a wide range of bit rates from 20-30 kbps to high bit rates to cover both video and audio compression [2] [4].

The JPEG standard was developed in 1992 for compression of still images for storage purposes [1] [2] [4]. The standard is currently also used to code moving images. It uses the 8x8 DCT followed by quantization. The video signal is then re-ordered before the application of the variable length coding in entropy encoder [1]. Though JPEG has been outperformed by JPEG 2000; applications such personal computer (PC), Digital cameras and website pages still use JPEG [2] [4]. JPEG 2000 was developed in 2000 as an improvement to the original JPEG of 1992 for compression of still images as well. However, unlike the original JPEG, JPEG 2000

adopts Discrete Wavelet Transform (DWT) for transform coding [1] [2] [4]. Therefore JPEG 2000 avoids the weaknesses of DCT and offers better compression when compared with JPEG.

Joint ISO/IEC and ITU-T Video Compression Standards

MPEG-2 and H.262 standards were jointly developed by ISO/IEC and ITU-T in 1995, for Compression and transmission of video and audio signals. The standards were to be used in storage and broadcast applications such as digital television (TV) and High Definition Television (HDTV) applications at typical bit rates of 3-5 mbps and above [9] [12] [4].

In 2001 a Joint Video Team (JVT), comprising of the VCEP from the ITU-T and the MPEG from the ISO/IEC was formed. Their mandate was to develop a new ITU-T recommendation which was later named H.264. This standard and/or recommendation is similar to MPEG-4 Part 10 standard also known as AVC and was developed in 2003. The H.264/AVC standard, or recommendation, was to cover a broad range of applications for video content including, but not limited, to the following [3]; multimedia mailing, multimedia services over packet networks, cable TV on optical networks, copper etc, digital subscriber line video services, digital terrestrial television broadcasting, remote video surveillance, real-time conversational services such as video conferencing [13], videophone and the like.

APPENDIX B: MMX Technology SIMD instruction

Table 1**:** MMX(TM) Technology General Support intrinsic [48] [49]

Intrinsic name	Corresponding MMX instruction	Operation
_mm_emty	EMM	Empty MM state
_mm_cvtsi32_si64	MOVD	Convert to int
_mm_cvtsi64_si32	MOVD	Convert to int
_mm_cvtsi64_m64	MOVQ	Convert from __int64
_mm_cvtsi64_si64	MOVQ	Convert to __int64
_mm_ packs_pi16	PACKSSWB	Pack
_mm_ packs_pi32	PACKSSDW	Pack
_mm_ packs_pu16	PACKUSWB	Pack

_mm_unpackhi_pi8	PUNPCKHBW	Interleave
_mm_unpackhi_pi16	PUNPCKHWD	Interleave
_mm_unpackhi_pi32	PUNPCKHDQ	Interleave
_mm_unpacklo_pi8	PUNPCKLBW	Interleave
_mm_unpacklo_pi16	PUNPCKLWD	Interleave
_mm_unpacklo_pi32	PUNPCKLDQ	Interleave

Table 2: MMX(TM) Technology Packed Arithmetic Intrinsics [48] [49]

Intrinsic name	Corresponding MMX instruction	Operation
_mm_add_pi8	PADDB	Addition
_mm_add_pi16	PADDW	Addition
_mm_add_pi32	PADDD	Addition
_mm_adds_pi8	PADDSB	Addition
_mm_adds_pi16	PADDSW	Addition
_mm_adds_pu8	PADDUSB	Addition
_mm_adds_pu16	PADDUSW	Addition
_mm_sub_pi8	PSUBB	Subtraction
_mm_sub_pi16	PSUBW	Subtraction
_mm_sub_pi32	PSUBD	Subtraction
_mm_subs_pi8	PSUBSB	Subtraction
_mm_subs_pi16	PSUBSW	Subtraction
_mm_subs_pu8	PSUBUSB	Subtraction
_mm_subs_pu16	PSUBUSW	Subtraction
_mm_madd_pi16	PMADDWD	Multiply and add
_mm_mulhi_pi16	PMULHW	multiplication
_mm_mullo_pi16	PMULLW	multiplication

Table 3: MMX(TM) Technology Shift Intrinsics [48] [49]

Intrinsic name	Corresponding MMX instruction	Operation
_mm_sll_pi16	PSLLW	Logical shift left
_mm_slli_pi16	PSLLWI	Logical shift left
_mm_sll_pi32	PSLLD	Logical shift left
_mm_slli_pi32	PSLLDI	Logical shift left
_mm_sll_pi64	PSLLQ	Logical shift left
_mm_slli_pi64	PSLLQI	Logical shift left
_mm_sra_pi16	PSRAW	Arithmetic shift right
_mm_srai_pi16	PSRAWI	Arithmetic shift right
_mm_sra_pi32	PSRAD	Arithmetic shift right
_mm_srai_pi132	PSRADI	Arithmetic shift right
_mm_srl_pi16	PSRLW	Logical shift right
_mm_srli_pi16	PSRLWI	Logical shift right
_mm_srl_pi32	PSRLD	Logical shift right
_mm_srli_pi32	PSRLDI	Logical shift right
_mm_srl_pi64	PSRLQ	Logical shift right
_mm_srli_pi64	PSRLQI	Logical shift right

Table 4: MMX(TM) Technology Logical Intrinsics [48] [49]

Intrinsic name	Corresponding MMX instruction	Operation
_mm_and_si64	PAND	Bitwise AND
_mm_andnot_si64	PANDN	Bitwise ANDNOT
_mm_or_si64	POR	Bitwise OR
_mm_xor_si64	PXOR	Bitwise Exclusive OR

Table 5: MMX(TM) Technology Compare Intrinsics [48] [49]

Intrinsic name	Corresponding MMX instruction	Operation
_mm_cmpeq_pi8	PCMPEQB	Equal
_mm_cmpeq_pi16	PCMPEQW	Equal
_mm_cmpeq_pi32	PCMPEQD	Equal
_mm_cmpgt_pi8	PCMPGTB	Greater than
_mm_cmpgt_pi16	PCMPGTW	Greater than
_mm_cmpgt_pi32	PCMPGTD	Greater than

Table 6: MMX(TM) Technology Set Intrinsics [48] [49]

Intrinsic name	Corresponding MMX instruction	Operation
_mm_setzero_pi8	PXOR	Set to zero
_mm_set_pi32	composite	Set integer values
_mm_set_pi16	composite	Set integer values
_mm_set_pi8	composite	Set integer values
_mm_set_pi32		Set integer values
_mm_set_pi16	composite	Set integer values
_mm_set_pi8	composite	Set integer values
_mm_set_pi32	composite	Set integer values
_mm_set_pi16	composite	Set integer values
_mm_set_pi8	composite	Set integer values

Appendix C: SIMD Transpose snippet Codes

```
__asm
{
mov eax,block
pshufw mm0,[eax],0xE4
pshufw mm1,[eax+8],0xE4
pshufw mm2,[eax+16],0xE4
pshufw mm3,[eax+24],0xE4
punpcklwd mm0,mm1
punpcklwd mm2,mm3
pshufw mm4,mm0,0xE4
punpckldq mm0,mm2
punpckhdq mm4,mm2
pshufw mm7,mm0,0xE4//movq
pshufw mm6,mm4,0xE4
pshufw mm0,[eax],0xE4
pshufw mm1,[eax+8],0xE4
pshufw mm2,[eax+16],0xE4
pshufw mm3,[eax+24],0xE4
punpckhwd mm0,mm1
punpckhwd mm2,mm3
pshufw mm4,mm0,0xE4
punpckldq mm0,mm2
punpckhdq mm4,mm2
pshufw mm5,mm0,0xE4
pshufw mm4,mm4,0xE4

}
```

Appendix D: Algorithm Snippet Codes

Appendix D-1 Code snippet for the matrix multiplication method

```
    short r[4][4];
    short c[4][4] = { { 1, 1, 1 ,1},
                      { 2, 1, -1, -2},
                      { 1, -1, -1, 1 },
                      { 1, -2, 2, -1 } };

// transpose input block1
    for ( i=0;i<4;i++)
    for ( j=0;j<4;j++)
    r[i][j]=block1[j][i];
// multiply c with r

    for (int i=0;i<4;i++)
      for (int j=0;j<4;j++) block1[i][j]=0;
    for (int i=0;i<4;i++)
      for (int j=0;j<4;j++)
        for (int k=0;k<4;k++)
    block1[i][j] +=c[i][k]*r[k][j];
```

Appendix D-2 Code snippet for the integer DCT butterfly method (C++ only)

```
/// 1-D forward horiz direction.
for(j = 0; j < 16; j += 4)
{
      /// 1st stage transform.
      int s0 = (int)(block[j]       + block[j+3]);
      int s3 = (int)(block[j]       - block[j+3]);
      int s1 = (int)(block[j+1] + block[j+2]);
      int s2 = (int)(block[j+1] - block[j+2]);

      /// 2nd stage transform.
      block[j]          = (short)(s0 + s1);
      block[j+2]  = (short)(s0 - s1);
      block[j+1]  = (short)(s2 + (s3 << 1));
      block[j+3]  = (short)(s3 - (s2 << 1));
}//end for j...
```

Appendix D-3 Code snippet for the integer DCT butterfly method (C++ with SIMD)

```
__asm
{

.

.

//butterfly
//stage one
pshufw mm0,mm7,0xE4
pshufw mm1,mm6,0xE4
pshufw mm2,mm5,0xE4
pshufw mm3,mm4,0xE4
paddw mm0,mm3
paddw mm1,mm2
movq [eax],mm0
movq [eax+8],mm1
pshufw mm0,mm7,0xE4
pshufw mm1,mm6,0xE4
pshufw mm2,mm5,0xE4
pshufw mm3,mm4,0xE4
psubw mm1,mm2
psubw mm0,mm3
movq [eax+16],mm1
movq [eax+24],mm0

//stage two
mov  edi, block
pshufw mm0,[eax],0xE4
pshufw mm1,[eax+8],0xE4
pshufw mm2,[eax+16],0xE4
pshufw mm3,[eax+24],0xE4
psubw mm0,mm1
pshufw mm6,mm0,0xE4
pshufw mm0,[eax],0xE4
pshufw mm1,[eax+8],0xE4
pshufw mm2,[eax+16],0xE4
pshufw mm3,[eax+24],0xE4
```

```
        paddw mm0,mm1
        psllw mm3,1
        paddw mm2,mm3
        movq [edi],mm0
        movq [edi+8],mm2
        pshufw mm2,[eax+16],0xE4
        pshufw mm3,[eax+24],0xE4
        psllw mm2,1
        psubw mm3,mm2
        movq [edi+16],mm6
        movq [edi+24],mm3

}
```