HARDWARE IMPLEMENTATION OF INVERSE TRANSFORM & QUANTIZATION AND DEBLOCKING FILTER FOR LOW POWER H.264 DECODER

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÖNDER ÖNSAY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2009

Approval of the thesis:

HARDWARE IMPLEMENTATION OF INVERSE TRANSFORM & QUANTIZATION AND DEBLOCKING FILTER FOR LOW POWER H.264 DECODER

submitted by **ÖNDER ÖNSAY** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. İsmet Erkmen _____
Head of Department, **Electrical and Electronics Engineering**

Prof. Dr. Gözde Bozdağı Akar
Supervisor, **Electrical and Electronics Engineering Dept., METU** _____

Examining Committee Members:

Prof. Dr. Murat Aşkar _____
Electrical and Electronics Engineering Dept., METU

Prof. Dr. Gözde Bozdağı Akar _____
Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Dr. A. Aydın Alatan _____
Electrical and Electronics Engineering Dept., METU

Assist. Prof. Dr. Cüneyt Bazlamaçcı _____
Electrical and Electronics Engineering Dept., METU

Fatih Say, M. Sc. _____
SST ABTM, ASELSAN

Date: **09.09.2009**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Önder ÖNSAY

Signature :

ABSTRACT

HARDWARE IMPLEMENTATION OF
INVERSE TRANSFORM & QUANTIZATION AND
DEBLOCKING FILTER FOR
LOW POWER H.264 DECODER

Önsay, Önder
M.Sc., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Gözde Bozdağı Akar

September 2009, 151 pages

Mobile devices became indispensable part of business and entertainment world. Applications running on these devices tend to increase day by day causing more battery power consumption. Video decoding is one of the critical applications running on a mobile device. H.264/AVC is an emerging video compression standard that is likely to be used widely in multimedia environments. As a mobile application, H.264 decoder has a complex structure which results high power consumption. In order to reduce this power demand, power consuming parts of the algorithms are required to be optimized in terms of power consumption, like deblocking filter and inverse transform & quantization. Data reuse and reduced processing time for moderate quality video are some of the methods to reduce power consumption. In this thesis, a deblocking filter architecture and inverse transform/quantization architecture with efficient data reuse and reduced memory access for low power 264/AVC decoder is proposed and implemented on Spartan-3 series FPGA. Proposed architectures obtained moderate processing speed with minimum external memory access.

Keywords: H.264, Power-efficient Codec, Deblocking, Transform & Quantization, Decoder Hardware

ÖZ

DÜŞÜK GÜÇ TÜKETİMLİ H.264 ÇÖZÜCÜ İÇİN TERS DÖNÜŞÜM & NİCELEME VE BLOKLAMA SÜZGECİNİN DONANIMSAL GERÇEKLENMESİ

Önsay, Önder

Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Gözde Bozdağı Akar

Eylül 2009, 151 sayfa

Taşınabilir cihazlar iş ve eğlence dünyasının vazgeçilmez bir parçası durumundadır. Bu cihazların üzerinde birçok uygulama çalışmakta, her geçen gün bunlara yenileri eklenmekte ve daha fazla batarya gücü tüketimine sebep olmaktadır. Video çözücü bu uygulamalardan biridir. H.264 birçok çoklu ortam uygulamasında kullanılacak yeni bir video sıkıştırma standardıdır. H.264 video çözücü taşınabilir cihazlar için karmaşık bir yapıya sahiptir ve bu nedenle bunu gerçekleyen donanım daha fazla güce ihtiyac duyar. Bu ihtiyacı düşürmek için algoritmada fazla güç tüketimine neden olan bloklama süzgeci ve ters dönüşüm & niceleme gibi parçaları düşük güç tüketimli uygulamalar için uygun hale getirmek gerekmektedir. Veriyi yeniden kullanım, orta kaliteli video için düşük işleme zamanı düşük güç tüketimi için kullanılan yöntemlerden birkaçıdır. Bu tezde, veriyi verimli şekilde yeniden kullanabilen ve düşük hafıza erişimi olan bir bloklama süzgeç ile ters dönüşüm & niceleme mimarileri ortaya konmakta ve Spartan-3 serisi FPGA üzerinde gerçeklenmektedir. Ortaya konan mimariler makul işlem zamanı ve minimum harici bellek erişimi ile çalışabilmektedir.

Anahtar Kelimeler: H.264, Güç Verimli Kodlayıcı-çözücü, Bloklama, Dönüşüm & Niceleme, Çözücü Donanımı

To My Family and Gülşah…

ACKNOWLEDGEMENTS

I would like to express my appreciation to my supervisor Prof. Dr. Gözde Bozdağı Akar for her patience, guidance, tolerance and friendship throughout the work.

I would like to thank to my colleagues for their assistance and technical support. I would like to thank my company ASELSAN for the use of technical resources and facilities that I used thorughout thesis work .

I would also like to express my thanks to my dear friends Ümit İrgin and Gökhan Güvensen for their invaluable academic motivation, encouragement and help throughout my academic life in METU.

Finally, I would like express my deepest thanks to my sister Özlem, for teaching me how to read and write and extraordinary support in every situation and to Gülşah Kafadar for being by my side.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

AVC	Advanced Video Coding
bS	Boundary strength
CABAC	Context-based Adaptive Binary Arithmetic Coding
CAVLC	Context-based Adaptive Variable Length Coding
CIF	Common Intermediate Format
DCT	Discrete Cosine Transform
DSP	Digital signal processor
DVD	Digital Video Disc
DWT	Discrete Wavelet Transform
FIR	Finite impulse response
FPGA	Field programmable gate array
HD	High Definition
HDL	Hardware description language
IC	Integrated circuit
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
ITU	International Telecommunication Union
JPEG	Joint Photographic Experts Group
JTAG	Joint Test Action Group
JVT	Joint Video Team
KLT	Karhunen-Loeve Transform
LUT	Look-up table
MF	Multiplication factor
MPEG	Motion Picture Experts Group
NAL	Network Abstraction Layer
PLD	Programmable logic device

PSNR	Peak signal-to-noise ratio
QCIF	Quarter Common Intermediate Format
QP	Quantization parameter
RAM	Random access memory
ROM	Read-only memory
SAD	Sum of absolute differences
SD	Start Definition
UART	Universal asynchronous transmitter receiver
VCEG	Video Coding Experts Group
VCL	Video Coding Layer
VHS	Video Home System
VLSI	Very large scale integration

CHAPTER 1

INTRODUCTION

1.1 General

Available bandwidth and limited storage resources made video compression indispensable part of video technologies. Researchers focused on this subject and developed various standards for compression of raw video content. In order to realize these applications, various hardware and software environments have been released. Mobile devices such as cellular phones and PDA's, and media players running on PC are some examples.

Since 1991, certain video compression standards have been developed by the two institutes ISO/IEC and ITU-T. MPEG-1, MPEG-2 and MPEG-4 are some examples which are developed by ISO whereas ITU-T developed H.261, H.263, H.263+ standards. In each new standard lower bitrate and higher subjective quality, in other words better rate-distortion performance, was aimed. Some of these standards are currently used in many multimedia environments. After development of these standards, H.264/AVC was released by JVT, which is a group formed by video coding experts from ISO and ITU-T. H.264/AVC is aimed to be an optimum compression standard that has extremely low bitrate, higher quality, and high adaptation to different network environments. H.264/AVC is further open to new extensions such as multi-view and scalable codec extensions.

Implementation of H.264 is well done in software environments. Open source reference software [1] is available and updated frequently. Algorithms related to

rate-distortion optimization are well improved. Another variable to be optimized is complexity which started to become important after the codec structure became highly complex for a real-time or mobile implementation. In order to solve the problem, efficient hardware architectures with effective parallel processing and pipelining capabilities started to be developed. For hardware implementation, application specific integrated circuits (ASIC) dedicated for H.264 encoder/decoder implementation are designed, in which dedicated hard processing blocks for each function (motion estimation, transform coding, quantization, deblocking filtering etc.) are used for processing.

Power effective solutions are also proposed to support mobile environments for H.264 decoder. Design for a low power H.264 decoder system requires special attendance to certain design aspects. In general low power design can be implemented by physical, architectural and algorithmic methods. [24]

Physical characteristic of silicon is always determinant on the performance of designed hardware. Silicon technology of device affects power and processing speed of the design. Also it determines the static and dynamic power consumption due to transistor losses and switching respectively. Dynamic power consumption is proportional with switching frequency, voltage and gate capacitance whereas static power consumption increases with smaller transistor size due to process technology.

Obviously hardware with higher operating frequency consumes more power than the same hardware with low operating frequency. A low-power system should be operated at a moderate frequency to fulfill performance and power requirements. Voltage is also effective on power consumption. New generation ASIC, FPGA and general purpose processors are designed to operate at lower core voltages whereas the old ones operate at much higher voltages. Dynamic frequency and voltage scaling, multiple voltages, multiple thresholds are some techniques to reduce power

consumption. Used gate count for the design may also be effective since every switching gate is a capacitance that consumes power during charge and discharge.

Processing platform is significant to get benefit of these physical methods. For application specific integrated circuits, total gate count is equal to required gate count for the design. This eliminates power consumption due to redundant gates which is a problem for FPGA, DSP and general purpose processor platforms. ASIC is designed for a specific application and the design process is physical which helps the designer to use the right operating voltage, frequency and resource to fulfill requirements. For other platforms operating voltage is fixed and operating frequency may not exceed a value which is not supported by the logic blocks contained in the design.

The architecture designed for a specific application determines the performance in terms of processing speed and power consumption. Hardware blocks with different characteristics are effective on this determination. In terms of performance, for instance, parallel processing blocks process concurrently more amount of data in a specific time or concurrently do more processing on the same amount of data than sequential processing blocks. Pipelined processing elements are also effective since they process flow of data and outputs to next one for next process which allows efficient data processing in every clock cycle time. In terms of power, for instance, memory access consumes more than register access. On the other hand, external memory access consumes more than internal memory access. Therefore data reuse is significant to reduce external and internal memory access and use fewer registers.

There are various low power implementations of H.264 decoder in the literature that used the techniques mentioned above. Physical methods such as process technology, clock gating and operating voltage reduction are used by [25], [11], [13] and [14] to reduce power consumption. Architectural methods such as lower memory use, pipelining and hardware reduction are used by [14], [13], [31], [16]

and [19]. In algorithm level, [25], [23] and [14] tried to modify the hardware algorithm so that it can be implemented by architecture with lower power consumption.

The methods mentioned above are the inputs to a design stage. The architecture is designed to use the right processing blocks and connect them in a right way to obtain the desired performance metric. Hardware algorithm is determinant on architecture design. If architecture is designed according to the right algorithm and implemented on the right processing platform, high performance, low power consumption or efficient resource use is achieved.

1.2 Scope of the Thesis

The main objective of the study is to develop main building blocks of a low-power H.264 decoder. As a starting point, inverse transform and quantization part is selected since it is well defined by the standard and can be designed and tested individually without the need for other building blocks. Low-power architecture with reduced memory access is aimed for video with a limit for quantization parameter.

On the other hand, deblocking filter is also selected to be analyzed and implemented for a low-power design, which is the most power consuming part of H.264 decoder (%34 of total power) [28]. Deblocking filter is again an independent part of the decoder which can be designed and tested individually. Architecture with reduced memory access, effective data reuse and sufficient processing speed is aimed for moderate quality low-power applications.

1.3 Outline of the Dissertation

The order of the chapters in this thesis is similar to work undertaken. Totally it consists of 4 further chapters and an appendix part.

In chapter 2, the history and overview of H.264 compression standard is summarized. Main processing units in H.264 codec structure are briefly explained.

Chapter 3 focuses on forward and inverse transform quantization concept. Theoretical background related to this chapter, conducted literature survey and proposed architecture is explained in detail. Implementation details and results are also given.

Fourth chapter focuses on deblocking filtering concept. Its explanation by the standard, literature survey about proposed processing orders and architectures are explained in detail. Proposed low-power implementation and obtained results are given in detail. In third and fourth chapters theoretical information and implementation are combined in the form of a detailed chapter to ease referencing of implemented architecture to its theoretical requirement.

Finally, in fifth chapter, the thesis work is summarized and conclusions drawn from the work are stated. Further studies that can be carried out on this subject are also emphasized.

CHAPTER 2

OVERVIEW OF H.264/AVC RECOMMENDATION

2.1 Introduction

Innovations in digital video applications influenced researchers to focus on video compression techniques which are essential for transmission and storage of raw video. On the other hand, transmission and storage techniques are also improved as well, nevertheless in order to use high quality and high resolution instead of low quality uncompressed video, video compression is still the key technology. Since early 1990s, mainly two research groups worked on video compression and developed several standards for specific applications. These are the ITU-T Video Coding Experts Group (VCEG) and ISO/IEC Motion Picture Experts Group (MPEG).

Firstly in 1990, H.261 [4] was developed by ITU-T. The purpose was compression for video conferencing and video telephony applications. H.261 supported frame sizes up to CIF (352x288). Based on gained experience, MPEG-1 [2] was introduced by ISO/IEC as a solution for home multimedia applications. MPEG-1 allowed compression of VHS-quality raw digital video and Video CD's (VCD).

After 3 years, MPEG-2 [3] was released by ISO/IEC as an improved version of MPEG-1. In MPEG-2, prediction modes and entropy coding methods were improved and support for interlaced video was provided. These improvements made MPEG-2 a generic and worldwide video standard for various applications that have different bitrate and quality requirements. Flexibility and interlaced video support allowed MPEG-2 to be used for broadcast cable-TV, compression in digital video discs (DVD), standard (SD) and high definition (HD) TV.

ITU-T released H.263 [5] in 1995, as an improved version of H.261 in terms of compression performance. Motion estimation and entropy coding techniques firstly introduced with H.263 have played pivotal roles for the development of next generation compression standards. In order to support a wide range of video content, with real-life and synthetic material, MPEG-4 Visual was standardized in 1999. MPEG-4 is an improved version of MPEG-1 and MPEG-2 in terms of compression performance, error resilience and especially flexibility for supported video content, such as natural video, 2D and 3D graphical objects.

After the commercial success of MPEG-1 and MPEG-2, and the compression performance and flexibility achievements with H.263 and MPEG-4 respectively, ISO and ITU-T decided to join in a group to develop a robust network-friendly standard with better compression performance and open to further improvements. Joint Video Team (JVT) formed by video coding experts of two leading groups, released H.264/AVC (Advanced Video Coding) [6] in 2003, using the background experience gained with the development of previous standards. H.264/AVC outperforms MPEG-2 and MPEG-4 in terms of rate-distortion performance. [8]

Most of the processing units of H.264 coding structure are actually improved versions of the ones used for previous standards. Motion compensated prediction to reduce temporal redundancy, transform coding to reduce spatial correlation, quantization for bitrate control and entropy coding are some examples. In order to solve the problems encountered before, also new methods are introduced with

H.264, to enhance the performance of building blocks. Generally, a layered structure is designed for H.264, to provide robustness and adaptability to different network environments. It consists of a video coding layer (VCL) for compression and a network abstraction layer (NAL) for network adaptation.

2.2 H.264 Encoder & Decoder Structure

Before explaining each of these blocks in detail, basic structures of H.264 encoder and decoder are illustrated in Figure 2-1 and 2-2.

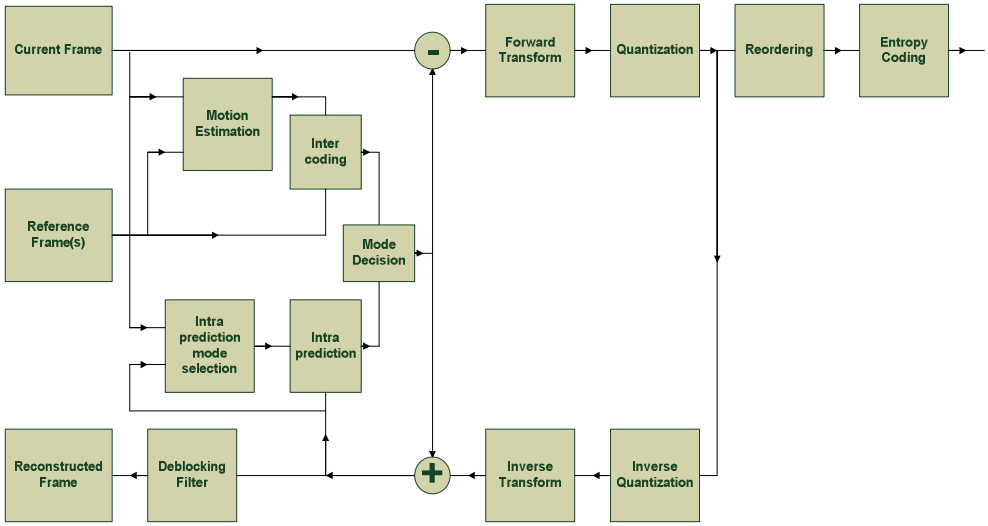

Figure 2-1. H.264 encoder structure

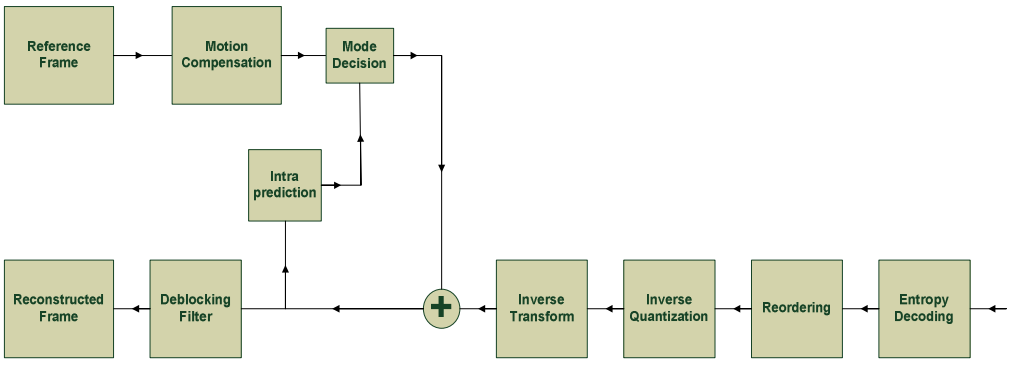

Figure 2-2. H.264 decoder structure

2.3 Supported Video Format

Interlaced (one of each two lines is scanned) or progressive (each line is scanned) video with 4:2:0 sampling format and 8-bit pixel representation is supported by H.264, as a default configuration. As illustrated in Figure 2-3, 4:2:0 sampled video contains 2 chroma samples (Cb and Cr) for every 4 luminance samples. Therefore horizontal and vertical resolutions of chrominance samples are half of the luma sample resolution. Some profiles of H.264 also support 4:2:2, 4:4:4 sampling formats (where chroma resolution is higher with respect to 4:2:0) and up to 12-bit pixel representations.

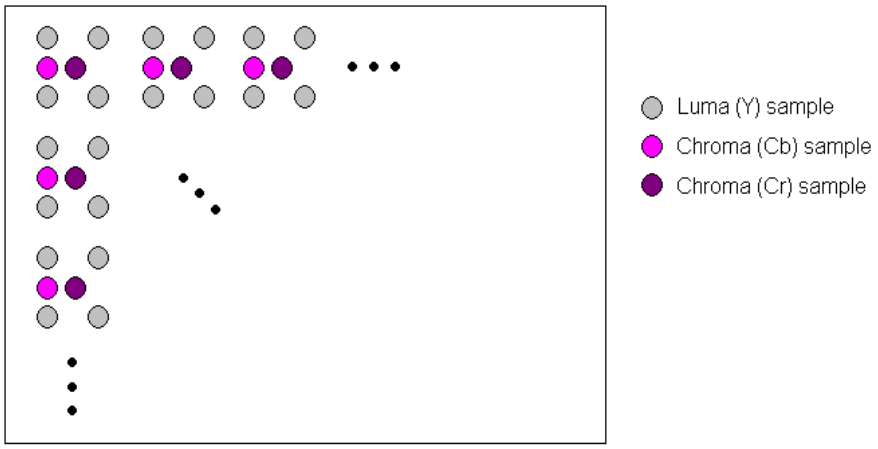

Figure 2-3. 4:2:0 sampled video frame

2.4 Macroblock and Frame Structure

In H.264, frames are splitted into 16x16 blocks, called macroblock, which are the main processed units for operations such as motion estimation & compensation, transform & quantization and entropy coding. As illustrated in Figure 2-4, for 4:2:0 sampling format, a macroblock consists of 16x16 luma (Y) samples and 8x8 chroma (Cb and Cr) samples.

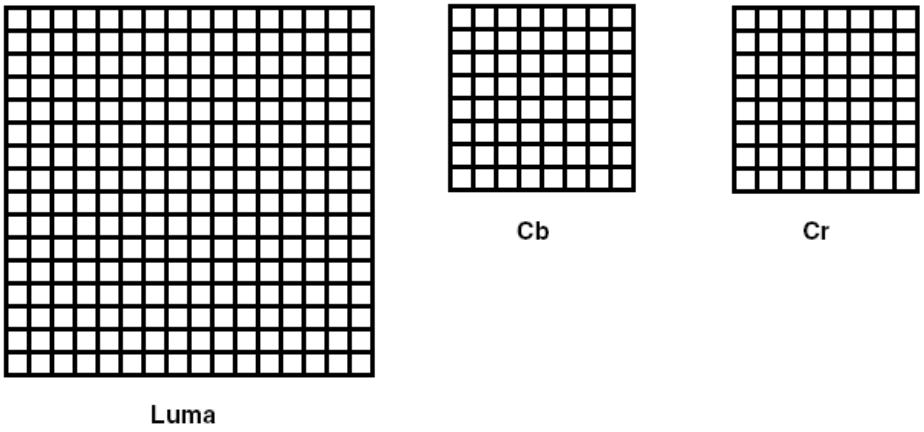

Figure 2-4. Macroblock structure for 4:2:0 sampling format

In H.264, frames are divided into macroblock groups, called slices. A slice may be an entire frame whereas a slice may contain a single macroblock. Slices are independently processed which may have different prediction and quantization options. There are 5 different slice modes depending on prediction types: I (Intra), P (Predicted), B (Bi-predictive), SI (switching I) and SP (Switching P).

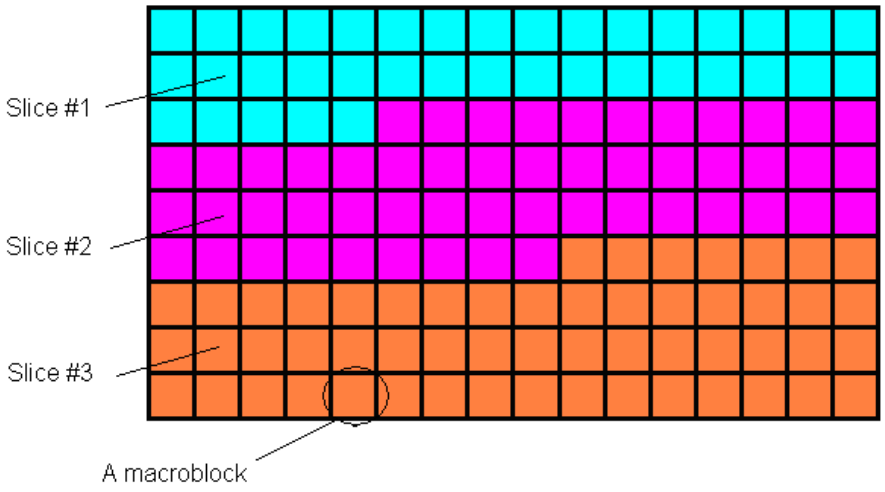

Figure 2-5. A slice configuration for a QCIF (176x144) frame

2.5 Intra Prediction & Coding

Intra coding is applied to make benefit of spatial correlation between samples. It is simply performed by prediction of a block using previously encoded upper and left neighboring blocks. Size of the prediction blocks depends on the detail of the frame. Parts with significant detail are usually coded using intra4x4 mode, whereas smooth areas are coded using intra16x16 mode. This adaptation has significant effect on bitrate reduction.

Prediction modes are defined for intra coding and one of them is selected for the best representation of currently predicted block. Intra16x16 prediction modes are illustrated in Figure 2-6. In vertical and horizontal modes prediction block is formed by copying up and left neighbor samples to current macroblock respectively. In DC mode, mean of upper and left samples are calculated and used for prediction whereas a plane function of upper and left samples is used for plane mode. As illustrated in Figure 2-7, similarly for intra4x4 there are 9 prediction modes and neighboring samples are used to predict the block samples. The sum of absolute difference (SAD) between a block and its prediction for each mode is calculated and the one with minimum SAD is selected.

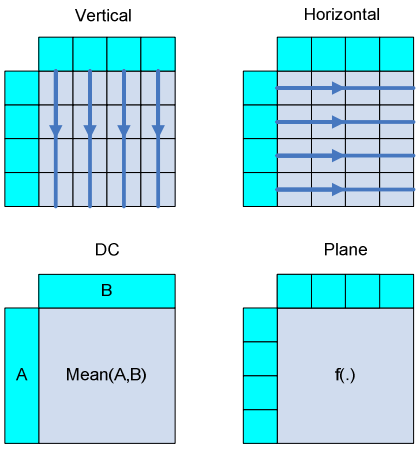

Figure 2-6. Intra16x16 prediction modes

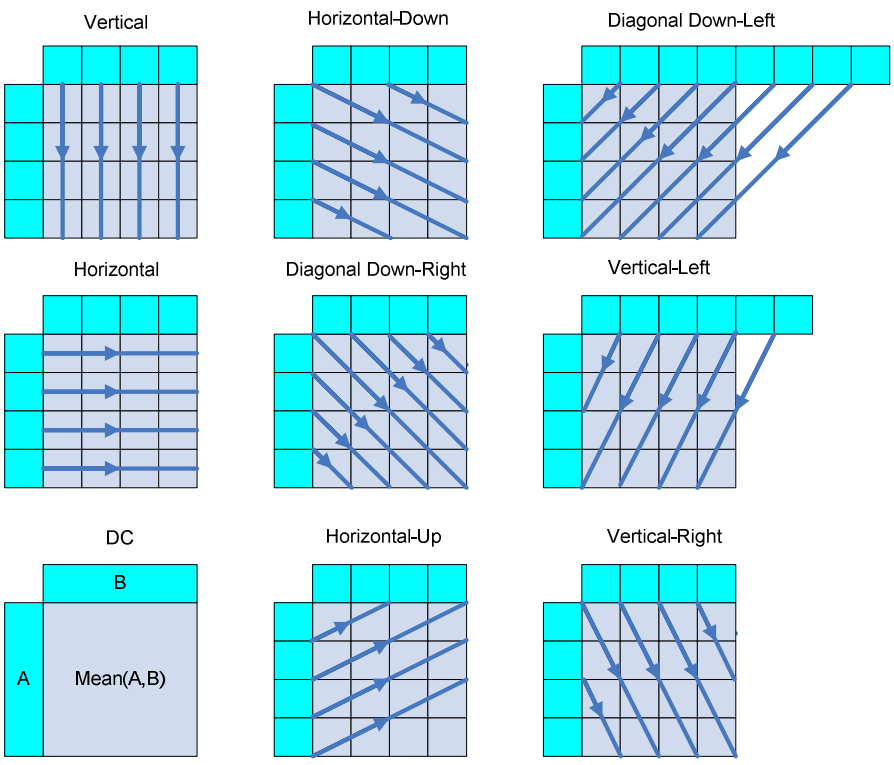

Figure 2-7. Intra4x4 prediction modes

2.6 Inter Prediction & Coding

Temporal correlation is the main source of redundancy for video compression. Since the similarity between adjacent frames is extremely high, reference frames are used to predict the current frame by obtaining motion vectors. This process is known as motion estimation. After motion vectors are determined, current frame can be reconstructed using reference frames and motion vectors. In H.264, motion estimation & compensation process is modified compared to previous standards. Variable block-size, quarter-pel accurate motion estimation & compensation is performed with ability of using multiple reference frames.

In H.264, multiple reference frames are used in order to obtain a better representation of a block to be coded. In previous standards, there was only a single reference frame for prediction. This technique results higher compression performance for which a larger memory buffer is required.

In H.264, prediction block size is used instead of fixed block size. Block size varies according to detail in motion. For detailed motion this approach has significant effect on compression performance since detailed motion can be represented better. As illustrated in Figure 2-8 [9], there are 7 prediction blocks. A two level method is used to select the appropriate block. In first level, 16x16, 8x16, 16x8 and 8x8 blocks are used for motion estimation. If the best residual is obtained by 8x8 in the second level 8x4, 4x8 and 4x4 modes are also tried for better representation. The mode with the representation which results minimum SAD for residual is selected.

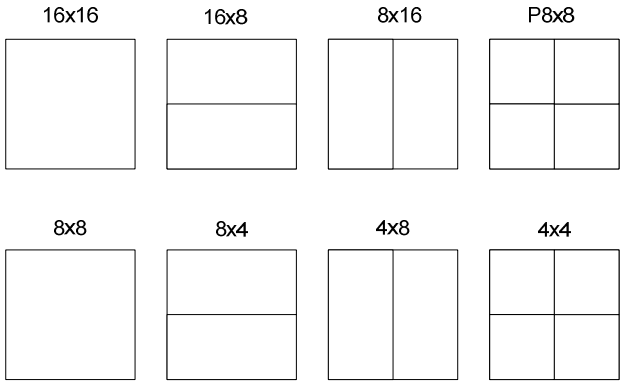

Figure 2-8. Variable prediction block sizes

In H.264 up to quarter-pixel accurate motion estimation is used. As shown in Figure 2-9, prediction values at half-pixel sample points (*ab*, *cd*, *ce*, *jk* etc.) are interpolated using FIR filtering of full-pixels (A, B, C, D, etc.). Similarly quarter-pixel samples are found by averaging half and full-pel sample values. Then motion estimation is conducted on interpolated samples. This approach provides better prediction and results lower bitrate.

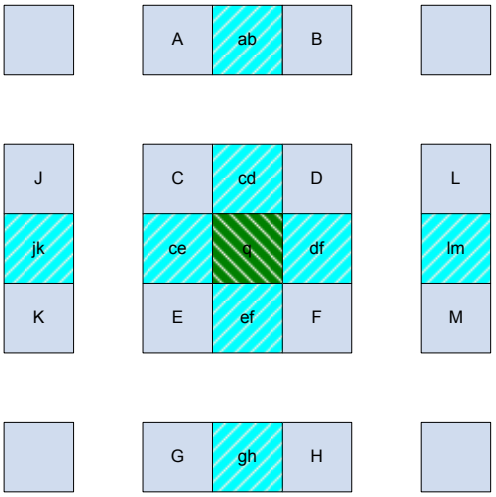

Figure 2-9. Interpolation for half-pixel accurate motion estimation

2.7 Transform & Quantization

Transform coding is conducted to reduce spatial correlation between residual samples to obtain a lower bit count representation for entropy coding. In H.264, 4x4 integer discrete cosine transform (DCT) is used whereas 8x8 classical DCT is used for previous standards. Smaller block size reduces blocking artifacts and is compatible with small prediction blocks. One another improvement of H.264 is that an integer approximation of 4x4 DCT is used. Transform operation can be implemented by add and shift operations. Integer arithmetic also provides exact calculation for all processing platforms which is not the case for floating point arithmetic. Integer transform is almost identical to original in terms of compression performance. Its inverse is similar and calculated with integer arithmetic without division.

Smooth areas in a frame are usually coded in intra16x16 mode. For this case, DC coefficients obtained after transform coding are still correlated and need to be further transformed. Hadamard Transform is performed on DC coefficients of each

4x4 luma block in a macroblock which modifies the energy concentration of DC coefficients, so that lower amount of bits is adequate for representation. Also 2x2 Hadamard transform is applied to chroma DC coefficients of 4x4 blocks in any macroblock, since chroma samples also have smooth behavior. Hadamard Transform can be performed by add and shift operations and its inverse is exactly the same.

In H.264, scalar quantization is used whereas vector quantization is used for some coding standards. Transformed coefficients are quantized before entropy coding. A parameter called, Quantization Parameter (*QP*) is defined to adjust the quantization step size for desired bitrate and quality performance. *QP* ranges from 0 to 51 so that a wider range of quantization step sizes can be supported for fine bitrate and quality adjustment. *QP* can be defined for entire video, frame and slice and even for each macroblock. More detailed explanation will be given in Chapter 3.

2.8 Deblocking Filter

Blocking artifacts due to coarse quantization of residual blocks and motion compensated blocks are filtered with an adaptive filter, called deblocking filter. Deblocking filter both operates as an in-loop filter at encoder side, to filter reference frames before buffering and as a post-loop filter at decoder side, to filter output video frames. In-loop filtering also improves compression performance since it provides better prediction whereas post-loop filtering is just effective on the subjective quality of decoded frame.

In order to discriminate real edges from artificial blocking edges, QP-dependent thresholds are defined by H.264 and pixel-differences near the block edges are compared with these thresholds for deblocking filtering requirement. Thresholds and filtering structures that will be applied also depend on the block boundary type. Type of the boundary is determined by consideration of corresponding prediction

block type, macroblock edges, quantized coefficient, number of reference frames and motion vectors. More detailed explanation will be given in Chapter 4.

2.9 Entropy Coding

Entropy coding is a lossless process used to represent quantized residual coefficients and syntax elements, such as reference frame index, quantization parameter, motion vectors, block type information and control data, with lower amount of bits. In H.264, context-adaptive entropy coding methods are used to provide lower bitrate compared to prior standards. These are Context Adaptive Variable Length Coding (CAVLC) and Context Adaptive Binary Arithmetic Coding (CABAC)

2.10 Encoding & Decoding Process

As illustrated in Figure 2-1, H.264 encoder structure consists of several processing units. Input frame is first splitted into macroblocks for processing. Macroblocks of input frame are predicted using the macroblocks of reference frame(s), by selected inter or intra prediction mode. Predicted macroblock is subtracted from the original to form the residual block. Residual is transformed and quantized to acquire coefficients that represent the residual. Transformed and quantized coefficients are reordered and sent to entropy encoder with motion vector and control data information. Entropy encoder forms the output bit stream for transmission or storage. Transformed and quantized coefficients are also inverse quantized and inverse transformed to form back the residual and added to predicted macroblock to form the reference for intra prediction. Reference is also deblocking filtered and loaded to reconstructed frame buffer for future prediction.

As illustrated in Figure 2-2, H.264 decoder gets the input bitstream and decodes to obtain the video output. Decoding operation is exactly the inverse of encoding where input bitstream is first entropy decoded to get syntax elements such as

transformed & quantized coefficients, block types, motion vector information and control data. Coefficient block is reordered, inverse quantized and inverse transformed to form the residual macroblock. Predicted macroblock is formed using the reference macroblocks and prediction mode information in the bitstream. Residual macroblock is added to predicted macroblock to form the unfiltered reconstructed macroblock. The final output, reconstructed macroblock, is obtained after deblocking filtering. Decoded frame is formed by reconstructed macroblocks. Unfiltered reconstructed macroblock is fed back as a reference for intra prediction of neighboring macroblocks.

2.11 Profiles

In H.264, profiles are defined to configure the structure according to different compression requirements. Mainly 4 different profiles are defined as: baseline, main, extended and high.

Baseline profile is the default configuration that supports inter and intra coding with P and I slices, CAVLC method as entropy coding, and redundant slices for data recovery. It is appropriate for video conferencing, wireless and low-power applications which require moderate compression performance but low complex processing. Main profile further supports CABAC, inter-coding using B slices, weighted prediction and interlaced video to be used for interlaced video and improve the compression performance. It is commonly used for video storage and television broadcasting. Extended profile adds B-slice and weighted prediction support to baseline profile, additionally with SP and SI slice support and data partitioning for better error-resilience. This profile is more appropriate for streaming applications for which error resilience is critical. High profile supports various sampling formats with higher chroma resolution and higher pixel bit-depth. It is mainly used for professional applications.

CHAPTER 3

FORWARD AND INVERSE TRANSFORM & QUANTIZATION

3.1 Introduction

Residual data obtained by inter and intra prediction still has redundancy for entropy coding. It contains correlated values and therefore represented by similar number of bits resulting high bitrate. In order to form a data that still represents the residual but is decorrelated, transform coding is used. The purpose of transform coding is to obtain a few large numbers that have large portion of total energy and plenty of small numbers that have the rest energy. At entropy coder, by representing a few large values with high bit count and plenty of small values with low bit count, total bitrate is reduced.

Transform coding methods are grouped into two as the block-based and image-based transforms. The popular block-based methods are Karhunen-Loeve Transform (KLT), Single Value Decomposition and Discrete Cosine Transform. All of these are applied to *NxN* data blocks. They are applicable because of their lower memory requirement and appropriateness to block-based residual coding. The disadvantage of block-based transform coding is blocking artifacts appearing at the reconstructed frame. Discrete Wavelet Transform (DWT) is an example for image-based transforms, which has higher performance than block-based

transforms. Although it results a higher quality reconstruction, DWT is not suitable for block-based standards and requires high memory for buffering. [9]

Still image compression standards, such as JPEG and JPEG2000 use DCT and DWT for transform coding respectively. DWT was preferred by JPEG2000 to improve the quality of compression, but not by video coding standards because of the reason mentioned above. MPEG-1, MPEG-2, MPEG-4 and H.263 are some of the first video coding standards that use DCT for transform coding. For these standards transform block-size is 8x8. In order to reduce the blocking artifacts caused by large block size and provide transform region for 4x4 prediction blocks, H.264 baseline profile uses 4x4 DCT for residual samples and Hadamard Transform for DC coefficients obtained by DCT. Reduced block size and DC coefficient transform improves compression performance in terms of bitrate and quality.

Prior to H.264, video coding standards used the exact DCT expression. In order to ease the implementation for different processing platforms (general purpose processor, DSP, VLSI etc.), an approximated integer transform is proposed and used by H.264. The purpose is not just the elimination of floating point operations but also to develop a transformation that is exactly invertible and give the same results at different platforms, which was not the case for prior standards. The reason is that floating point operations need the same floating-point representation and rounding to be standard at all platforms, which is not possible and each decoder design obtains different results. These modifications also reduced the need for arithmetic processing word-length from 32-bit to 16-bit as compared to prior standards. [9]

Before coding of transformed residual data, data is quantized by defined quantization levels. H.264 uses a scalar quantizer where some MPEG standards such as MPEG-4 use vector quantization. Transformed coefficients are quantized at encoder side and scaled (inverse quantized) at the decoder. This stage is critical,

where significant compression loss occurs, since quantized coefficients do not contain the complete information of transformed coefficients.

3.2 Theoretical Background

3.2.1 Complete Transform & Quantization Process in H.264

3.2.1.1 Transform & Quantization

As illustrated in Figure 3-1, every 4x4 block in input macroblock, is first forward transformed using 4x4 integer DCT. If macroblock is predicted in intra16x16 mode, the 4x4 matrix formed by DC coefficients of 4x4 luma coefficient blocks is Hadamard transformed and quantized. For all macroblocks, 2x2 version of Hadamard transform is applied to 2x2 matrix formed by DC coefficients of 4x4 chroma blocks. Each quantized DC coefficient is inserted to corresponding 4x4 post-scaled and quantized coefficient block. For macroblocks, which are predicted in different modes than intra16x16, 4x4 luma DC coefficient transform & quantization is not applied.

Transform & quantization is applied at the encoder side, to obtain uncorrelated and quantized coefficients, with *QP* as a rate-distortion controlling parameter. Obtained coefficients are entropy coded to form the encoded bitstream.

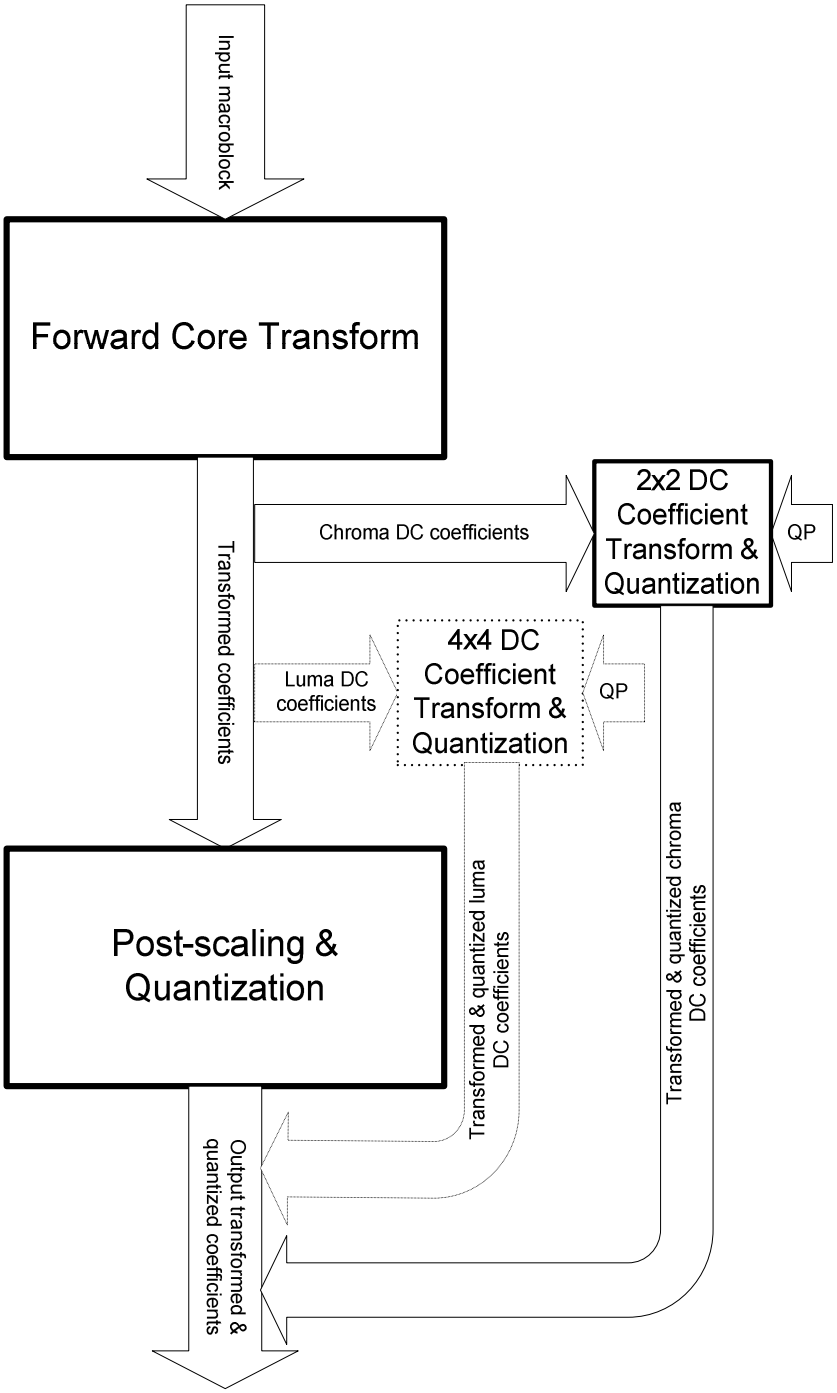

Figure 3-1. Transform and quantization process

3.2.1.2 Scaling & Inverse Transform

As illustrated in Figure 3-2, firstly, if macroblock is predicted in intra16x16 mode, inverse Hadamard transform and scaling is applied to luma DC coefficients of input quantized coefficients. Scaling and 2x2 inverse Hadamard transform is applied to chroma DC coefficients. Other 4x4 coefficient blocks are inverse quantized and then pre-scaled, to avoid rounding errors, before inverse core transform. Scaled and inverse Hadamard transformed DC coefficients are inserted to corresponding DC positions of these coefficient blocks. Resultant 4x4 blocks are inverse core transformed. Obtained samples are finally post-scaled by 64 and rounded to eliminate pre-scaling. For macroblocks that are predicted in different modes than intra16x16, luma DC coefficient scaling and inverse transform is not applied.

Inverse quantization & inverse transform is applied both at encoder and decoder sides. At decoder side, entropy decoded coefficients are scaled and inverse transformed to obtain residual macroblocks. *QP* and prediction mode in the bitstream are the controlling parameters for scaling and inverse DC coefficient transform respectively. At encoder side, transformed and quantized coefficients are scaled and inverse transformed to be fed back to prediction modules as “previously encoded” reference macroblocks.

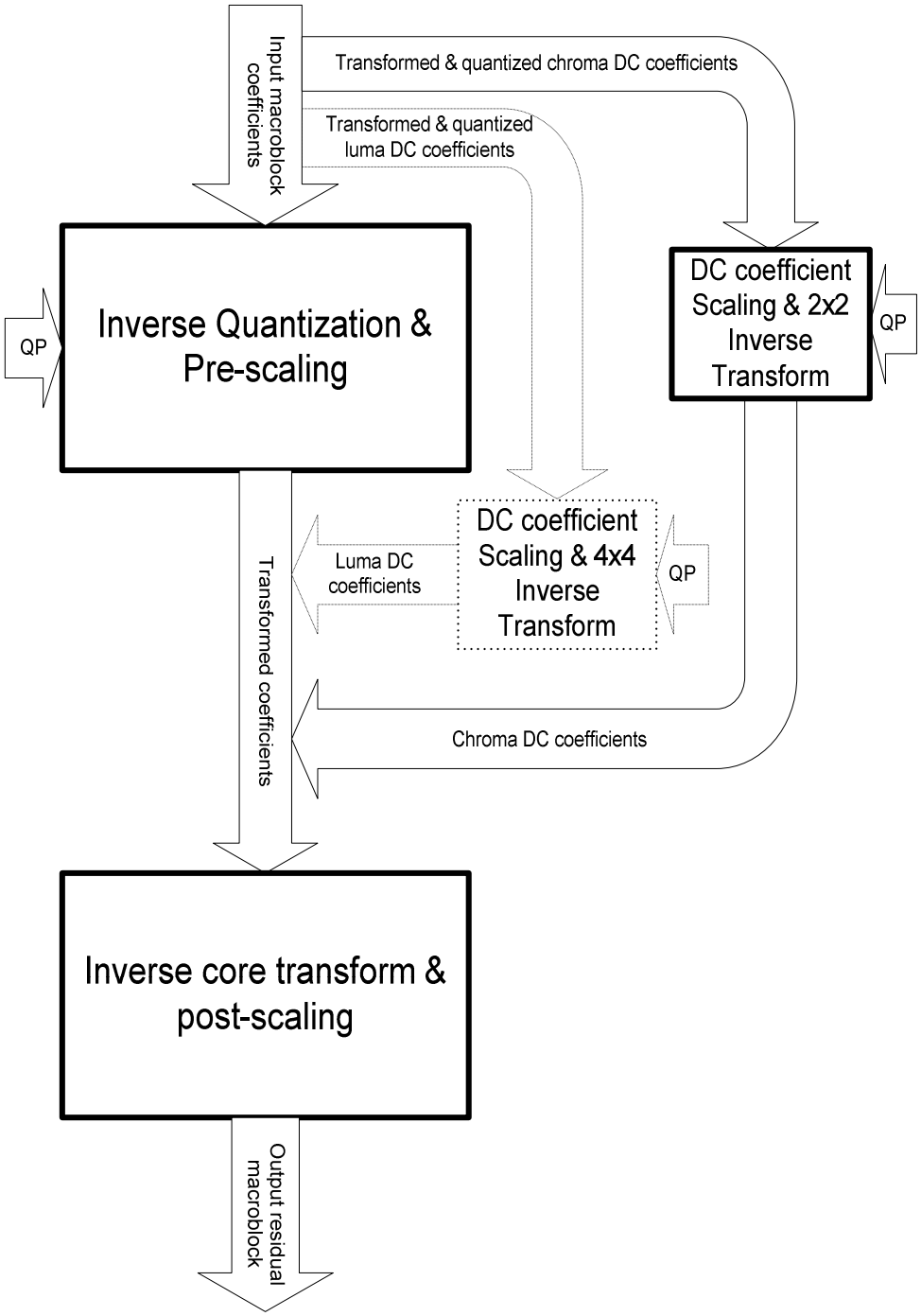

Figure 3-2. Scaling and inverse transform process

3.2.2 Transform & Quantization of a Macroblock

H.264 baseline profile uses three types of transform for each macroblock. A 4x4 integer DCT is used for each 4x4 block. For DC coefficients obtained by DCT, a special type of transform is applied to DC coefficients of 4x4 transformed luma blocks that are predicted in intra16x16 mode and DC coefficients of chroma blocks. The transmission order of blocks is as illustrated in Figure 3-3. For intra16x16 mode, first DC coefficients of luma blocks are transmitted. Then 4x4 luma blocks and DC coefficients of chroma blocks are transmitted. Finally 4x4 chroma blocks are transmitted. [9]

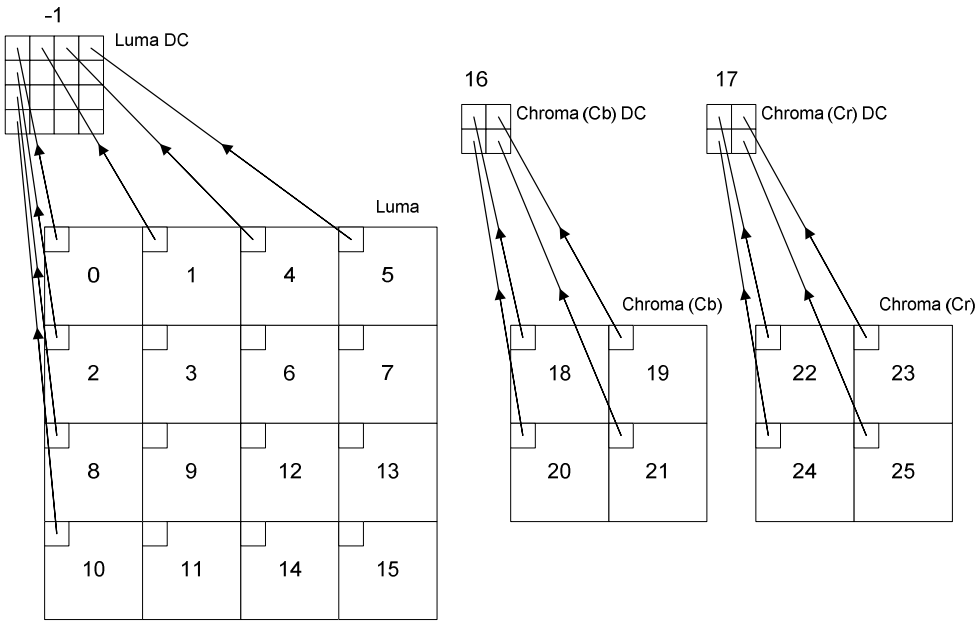

Figure 3-3. Transmission order of a macroblock. [9]

3.2.2.1 Discrete Cosine Transform

The classical DCT expression is a linear transformation [10]:

$$Y = H_{DCT}x \tag{3.1}$$

where x is a length-N vector and H_{DCT} is a linear transformation matrix that maps x to Y with entries defined as:

$$H_{kn} = c_k\sqrt{\frac{2}{N}}\cos\left[\left(n+\frac{1}{2}\right)\frac{k\pi}{N}\right] \tag{3.2}$$

for *kth* row and *nth* column of H_{DCT}, with $c_0 = \sqrt{2}$ and $c_k = 1$.

Hence for $N = 4x4$, H_{DCT} becomes:

$$H_{DCT} = \begin{bmatrix} \frac{1}{2}\cos(0) & \frac{1}{2}\cos(0) & \frac{1}{2}\cos(0) & \frac{1}{2}\cos(0) \\ \sqrt{\frac{1}{2}}\cos(\frac{\pi}{8}) & \sqrt{\frac{1}{2}}\cos(\frac{3\pi}{8}) & \sqrt{\frac{1}{2}}\cos(\frac{5\pi}{8}) & \sqrt{\frac{1}{2}}\cos(\frac{7\pi}{8}) \\ \sqrt{\frac{1}{2}}\cos(\frac{2\pi}{8}) & \sqrt{\frac{1}{2}}\cos(\frac{6\pi}{8}) & \sqrt{\frac{1}{2}}\cos(\frac{10\pi}{8}) & \sqrt{\frac{1}{2}}\cos(\frac{14\pi}{8}) \\ \sqrt{\frac{1}{2}}\cos(\frac{3\pi}{8}) & \sqrt{\frac{1}{2}}\cos(\frac{9\pi}{8}) & \sqrt{\frac{1}{2}}\cos(\frac{15\pi}{8}) & \sqrt{\frac{1}{2}}\cos(\frac{21\pi}{8}) \end{bmatrix} \tag{3.3}$$

And H_{DCT} matrix can be rewritten as:

$$H_{DCT} = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & c \end{bmatrix} \tag{3.4}$$

where and $a = 1/2$, $b = \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right)$ $c = \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right)$

Using the DCT matrix, the classical 4x4 DCT expression is given by:

$$Y = H_{DCT} X H_{DCT}{}^{T} = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix} [X] \begin{bmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{bmatrix} \tag{3.5}$$

The $Y_{(0,0)}$ value is called the "DC coefficient" of transformed *X*. Other values in *Y* represent frequency components of *X*.

3.2.2.2 Forward & Inverse Transform of 4x4 Blocks

4x4 blocks, numbered 0-15 and 18-28 in Figure 3-3, are transformed using DCT after inter/intra prediction. The classical expression of DCT has irrational entries as seen from (3.5). In order to obtain a compact integer expression, [10] replaced H_{DCT} with an orthogonal integer matrix. This is known as the integer DCT in literature. In H.264 standard [6], for 4x4 blocks, instead of classical DCT transform, integer DCT and scalar multiplication with a scaling matrix is conducted. Scaling matrix is further integrated into the quantization stage, to reduce the amount of multiplications. [9]

To obtain integer expression, (3.5) is factorized as:

$$Y = (CXC^{T}) \otimes E = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{bmatrix} [X] \begin{bmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \tag{3.6}$$

where CXC^{T} is defined as "forward core transform" by [9], E is a scaling matrix and $d = c/b$. When *d* is approximated as 0.5 and 2nd and 4th rows of matrix C is scaled by 2 to avoid division, integer DCT expression is obtained. *b* is also

modified for transform to remain orthogonal. The expression of final forward transform becomes:

$$Y = C_f X C_f^T \otimes E_f = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} [X] \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{bmatrix} \tag{3.7}$$

where $a = \frac{1}{2}$, $b = \sqrt{\frac{2}{5}}$ and $W = C_f X C_f{}^T$ is the integer forward core transform.

Obtained final expression is an approximation to classical 4x4 DCT. [10] states that the difference between the original and approximated transforms is clear but there is not significant loss in terms of compression performance. Coding gain loss is calculated by [10] to be less than 0.01 dB which doesn't cause any performance penalty. Approximated expression is advantageous, such that only addition, subtraction and shift operation is needed for core transform calculation, 16-bit arithmetic can be used and scalar multiplication by E_f can be inserted to quantization process.

H.264 standard [6] defines the inverse transform explicitly as:

$$X = C_i^T (Y \otimes E_i) C_i = \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} \left([Y] \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix} \tag{3.8}$$

In the expression since Y is pre-scaled by multiplying with scaling matrix E_i, multiplication by $1/2$ in the inverse core transform expression can be implemented by a right-shift without accuracy loss.

3.2.2.3 Quantization

Quantization in H.264 is designed to be computationally efficient. In order to provide this, division operation is avoided and scaling matrices in transform expressions are integrated to quantization, as explained in previous sections.

The forward quantization operation is defined as [9]:

$$Z_{ij} = round(\frac{Y_{ij}}{Qstep}) \tag{3.9}$$

where Y_{ij} is a transformed coefficient, $Qstep$ is the quantization step size and Z_{ij} is the quantized transform coefficient.

Quantization step size is effective to determine the amount of information contained in quantized coefficients, which determines the quality and bitrate. $Qstep$ is controlled by a controller parameter, called Quantization Parameter. This parameter is a critical controller for rate-distortion performance. Quantization step size doubles for every increment of *QP*. H.264 standard [6] defines values for *QP* as 0 to 51 and therefore 52 step sizes are defined for quantization, as shown in Table 3-1. With increasing *QP*, bitrate which is approximately proportional with quantization step size, also increases. [7] emphasizes that every increment of *QP*, results approximately 12% increase for quantization step size and that much bitrate reduction, as illustrated in Figure 3-4.

Table 3-1. Quantization step size values for each *QP* [6]

QP	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Qstep	0,625	0,6875	0,8125	0,875	1	1,125	1,25	1,375	1,625	1,75	2	2,25	2,5
QP	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
Qstep	2,75	3,25	3,5	4	4,5	5	5,5	6,5	7	8	9	10	11
QP	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**
Qstep	13	14	16	18	20	22	26	28	32	36	40	44	52
QP	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**
Qstep	56	64	72	80	88	104	112	128	144	160	176	208	224

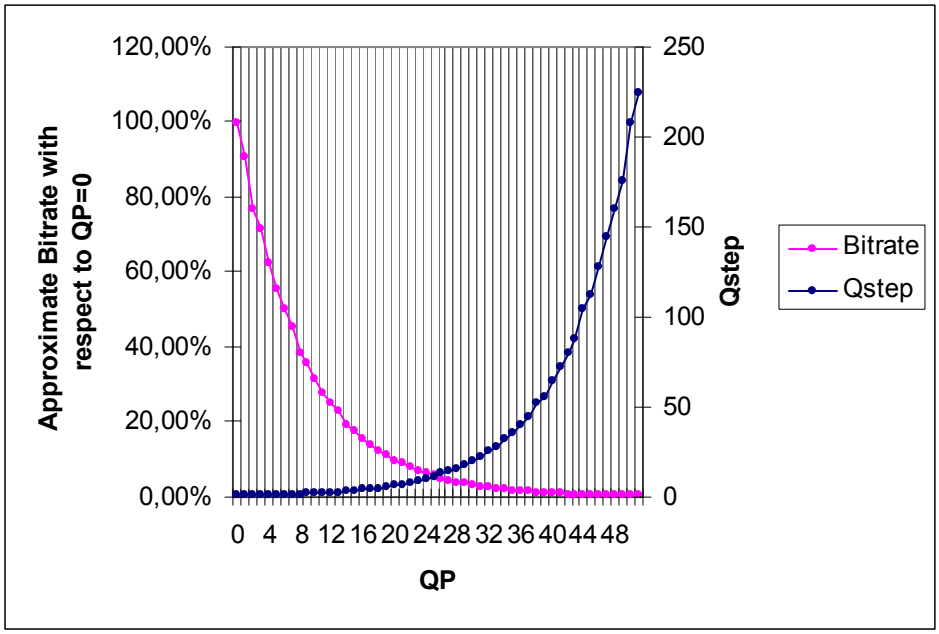

Figure 3-4. Quantization step size and approximate bitrate with respect to *QP=0*

Different *QP* values can be assigned to luma and chroma components. In H.264 standard [6], by default, *QP* for chroma is derived from *QP* for luma. For values of *QP* for luma greater than 30, chroma *QP* is set to be less than luma *QP*. [9]

Scaling matrix values can be combined with quantizer calculation. Post-scaling matrix E_f in (3.7) is integrated to forward quantizer and (3.9) becomes:

$$Z_{ij} = round(\frac{Y_{ij}}{Qstep}) = round\left(W_{ij}\frac{PF_f}{Qstep}\right) \tag{3.10}$$

where *i* and *j* represents row and column numbers of respective matrices and PF_f is the corresponding entry of E_f . Hence PF_f values are:

$$PF_f = \begin{cases} a^2, & (i,j) = (0,0),(2,0),(0,2),(2,2) \\ b^2/4, & (i,j) = (1,1),(1,3),(3,1),(3,3) \\ ab/2, & else \end{cases} \tag{3.11}$$

Reference model software [1] implements $\left(\frac{PF_f}{Qstep}\right)$ as a multiplication and a right-shift operation as:

$$\left(\frac{PF_f}{Qstep}\right) = \left(\frac{MF}{2^{qbits}}\right) \Rightarrow Z_{ij} = round\left(W_{ij}\frac{MF}{2^{qbits}}\right) \tag{3.12}$$

where $qbits = 15 + floor(QP/6)$ (3.13)

and *MF* is a multiplication factor for each coefficient position. *MF* values used by the reference software are shown in Table 3-2. *MF* values sometimes can be slightly changed in order to improve perceptual quality [9].

Table 3-2. *MF* values for given *QP* and position [9]

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

As illustrated in Table 3-2, *MF* values are given for $QP < 6$. For higher *QP* values *MF* values are calculated from values given in Table 3-2, since:

$$MF = MF(floor(QP / 6)) \tag{3.14}$$

The final expression of forward quantizer for integer arithmetic becomes:

$$\begin{aligned} &|Z_{ij}| = \left(|W_{ij}| \cdot MF + f\right) / 2^{qbits} \\ &sign(Z_{ij}) = sign(W_{ij}) \end{aligned} \tag{3.15}$$

f can be implemented as [1]:

$$f = \begin{cases} \dfrac{2^{qbits}}{3}, & for\ Intra \\ \dfrac{2^{qbits}}{6}, & for\ Inter \end{cases} \tag{3.16}$$

3.2.2.4 Inverse Quantization (Scaling)

Inverse quantization operation is done to obtain transformed coefficients before inverse transform. The operation is basically:

$$\hat{Y}_{ij} = Z_{ij} Qstep \tag{3.17}$$

where $\hat{Y}_{ij}$ is the estimated transform coefficient.

As in the case for quantization which integrates the scalar multiplication matrix E_f of transform expression, scaling operation integrates pre-scaling factor E_i of inverse transform, together with a constant scaling factor to avoid errors caused by rounding [9]:

$$\hat{W}_{ij} = Z_{ij} Qstep \cdot PF_i \cdot 64 \tag{3.18}$$

where i and j represents row and column numbers of respective matrices and PF_i is the corresponding entry of E_i. At the output of inverse transform, result is divided by 64 to remove the scaling factor. H.264 standard [6] defines a parameter V for this operation [9]:

$$V = Qstep \cdot PF_i \cdot 64 \tag{3.19}$$

V is effective to reduce number of multiplications required for each coefficient. As in the case for MF, V is also defined for $QP < 6$, as shown in Table 3-3, and further scaling factors are computed by doubling for every QP increment of 6:

$$\hat{W}_{ij} = Z_{ij} V_{ij} 2^{floor(QP/6)} \tag{3.20}$$

Table 3-3. *V* values for given *QP* and matrix position [6]

QP	Positions (0,0), (2,0), (2,2), (0,2)	Positions (1,1), (1,3), (3,1), (3,3)	Other Positions
0	10	16	13
1	11	18	14
2	13	20	16
3	14	23	18
4	16	25	20
5	18	29	23

3.2.2.5 Hadamard Transform

Hadamard Transform is a symmetric, linear and orthogonal transformation that has relationship with multidimensional DFT. It transforms a $2^m \text{x} 2^m$ vector x_k to $2^m \text{x} 2^m$ vector X_k. [10]

$$X_k = H_m x_k \tag{3.21}$$

Hadamard transformation matrix is recursively defined as [10]:

$$H_m = \sqrt{\frac{1}{2}} \begin{pmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{pmatrix} \tag{3.22}$$

where $H_0 = 1$ by identity. H_1 and H_2 can be calculated by (3.22) as:

$$H_1 = \sqrt{\frac{1}{2}} \begin{pmatrix} H_0 & H_0 \\ H_0 & -H_0 \end{pmatrix} = \sqrt{\frac{1}{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.23}$$

$$H_2 = \sqrt{\frac{1}{2}} \begin{pmatrix} H_1 & H_1 \\ H_1 & -H_1 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \tag{3.24}$$

H_1 and H_2 are used for 2x2 and 4x4 Hadamard Transform respectively. Scalar multiplication in the expression can be omitted. In H.264 standard [6] a different form of H_2 is used, which is obtained by substitution of 2nd, 3rd and 4th rows with each other and is still orthogonal and symmetric:

$$H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \tag{3.25}$$

3.2.2.6 4x4 Luma DC Coefficient Transform & Quantization

As in the case for 4x4 blocks, DC coefficients obtained by DCT of each 4x4 block are also correlated, when the macroblock is predicted in intra16x16 mode. The reason is the concentration of energy in DC coefficients for smooth surfaces which are predicted in intra16x16 mode. In order to decorrelate DC coefficients, Hadamard Transform is used. 4x4 matrix of DC coefficients, which is numbered "-1" in Figure 3-3, is 4x4 Hadamard transformed and divided by 2 with rounding [9]:

$$Y_{DC} = round\left\{ 1/2 \cdot \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} [W_{DC}] \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \right\} \tag{3.26}$$

Where W_{DC} and Y_{DC} are 4x4 matrices formed by DC coefficients of W and Y, respectively. Y_{DC} entries are further quantized similarly, to obtain quantized DC coefficients:

$$\left|Z_{DC(i,j)}\right| = \left(\left|Y_{DC(i,j)}\right| \cdot MF_{(0,0)} + 2f \right) / 2^{(qbits+1)} \tag{3.27}$$

$$sign\left(Z_{DC(i,j)}\right) = sign\left(Y_{DC(i,j)}\right) \tag{3.28}$$

Since DC coefficients are quantized, *MF* value that corresponds to (0,0) position is used.

Inverse transform is similar to (3.26):

$$W_{QDC} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} [Z_{DC}] \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \tag{3.29}$$

Inverse quantization is performed using *V* parameter for (0, 0) position [9]:

for $QP \geq 12$:

$$\hat{W}_{DC(i,j)} = W_{QDC(i,j)} \cdot V_{(0,0)} \cdot 2^{\left(floor\left(QP/6 \right) - 2 \right)} \tag{3.30}$$

Else:

$$\hat{W}_{DC(i,j)} = \left[W_{QDC(i,j)} \cdot V_{(0,0)} + 2^{\left(1 - floor\left(QP/6 \right) \right)} \right] / 2^{\left(2 - floor\left(QP/6 \right) \right)} \tag{3.31}$$

Obtained scaled DC coefficients $\hat{W}_{DC(i,j)}$, are inserted to (0,0) positions of each 4x4 block in a macroblock, then 4x4 inverse DCT is applied.

3.2.2.7 2x2 Chroma DC Coefficient Transform & Quantization

DC coefficients of 4x4 chroma blocks are also correlated, since chroma samples generally have smooth behavior. For each chroma component, block size in a macroblock is 8x8, for default 4:2:0 color sampling in H.264. Therefore a 2x2 matrix is formed by DC coefficients of 4x4 chroma blocks, numbered 16 and 17 in Figure 3-3. 2x2 version of Hadamard Transform, (3.23) is used for this case [9]:

$$Y_{DC} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} [W_{DC}] \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.32}$$

Where $\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ is again orthogonal and consequently its inverse is the same. Transformed DC coefficients are quantized similarly by (3.27) and (3.28).

Inverse transform is expressed as:

$$W_{QDC} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} [Z_{DC}] \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.33}$$

Inverse quantization is performed by [9]:

For $QP \geq 6$

$$W'_{DC(i,j)} = W_{QDC(i,j)} \cdot V_{(0,0)} \cdot 2^{floor(QP/6)-1} \tag{3.34}$$

Else:

$$W'_{DC(i,j)} = \left(W_{QDC(i,j)} \cdot V_{(0,0)} \right) / 2 \tag{3.35}$$

Obtained scaled DC coefficients are inserted to DC coefficient positions of 4x4 chroma blocks.

3.3 Hardware Architectures in Literature

Forward and inverse transform & quantization is one of the critical sections in H.264 encoder/decoder system. Due to parallel arithmetic operations requirement for forward and inverse matrix transformations and quantization, hardware architectures became popular for forward and inverse transform and quantization. These architectures allow parallel processing for matrix operations in DCT, Hadamard Transform and shift operations for divisions and multiplications in forward and inverse quantization. Also hardware solutions provide pipelined processing which effectively utilize the processing stages and reduces effective processing duration of a macroblock in a complete frame. These features made them indispensable for low-power and high-performance H.264 decoders.

There are various approaches for hardware implementation of forward and inverse transform & quantization. Most of them are architecture solutions for standard forward and inverse transform & quantization defined by the standard [4]. Some of them propose a modified algorithm to reduce processing cycle time.

I. Amer et. al. [12] proposed an architecture for 8x8 transform & quantization to be used for FRExt (Fidelity Range Extensions) extension of H.264 [8]. The architecture takes 8x8 samples directly as inputs, perform processing and outputs 8x8 quantized coefficients directly. The architecture is implemented on FPGA with extremely large logic resources to be inserted into a H.264 decoder system. The architecture seems to be designed just to obtain high throughput of 1 macroblock/clock cycle, but resultant I/O and logic count is extremely high.

E. P. Hong et. al. [32] and T. C. Wang [27] proposed 4x4 forward and inverse transform architectures based on bit-extension. They aimed to obtain architecture with lower logic utilization by reducing adder bit-width for different stages of forward and inverse integer DCT calculation. The method seems to be effective to reduce transform architecture complexity but not sufficient considering complete

transform and quantization architecture in terms of power consumption and complexity.

W. Hwangbo et. al. [33] proposed a high performance inverse transform architecture with modifying the calculation steps. They used permutation matrices to implement inverse transform operations with fewer additions. The purpose is to obtain lower critical path delay for higher throughput.

Z. Y. Cheng et. al. [29] proposed a high throughput forward and inverse DCT and Hadamard Transform architectures based on 1-D transform kernels with addition/subtraction operations for input and output samples.

R. Kordasiewicz et. al. [30] proposed a 4x4 forward transform and quantization architecture with totally 4 cycle delay for transform and quantization calculations. The design suffers from high amount of 16 multiplier requirement for quantization operation and I/O transfer latencies.

H. Y. Lin et. al. [31] proposed combined architectures for forward and inverse transform & quantization. Both forward and inverse transform operations are splitted into two parts. Quantization and inverse quantization operations are inserted into transform stages to decrease number of multiplications.

O. Tasdizen et. al. [26] proposed a forward and inverse transform & quantization hardware with reconfigurable architecture for forward and inverse transform. Input and output register files are also combined with the architecture to evaluate macroblock processing performance different from the previously mentioned works which only focus on the transform & quantization core. Nevertheless processing speed performance of the designed core is extremely high for intra16x16 prediction mode case.

3.4 Proposed Implementation

3.4.1 Proposed Representation for Coefficients

Most of the architectures in literature focus on the processing part of forward and inverse transform & quantization architecture and try to reduce the gate count, increase processing speed etc. The cost of input and output data transfer in terms of speed and power is usually not taken into account.

As illustrated before in Figure 3-3, there are mainly 4 types of input data to be processed by inverse transform & quantization architecture. These are luma DC coefficients (for intra16x16 mode), luma coefficients, chroma DC coefficients and chroma coefficients.

Forward transform & quantization takes residual samples as input and obtains transformed and quantized coefficients. Generally 16-bit bit-width is used to represent incoming residual samples and output coefficients. Input residual samples are in the range (-255, 255) which can be represented by 9-bits actually but in order to be conformable to general 16-bit structure represented by 16-bits.

The range for luma DC coefficients can be found by applying residual macroblock input with white samples:

$$X = \begin{bmatrix} 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \end{bmatrix} \quad (3.36)$$

After forward DCT applied to each 4x4 block in macroblock, coefficients and are obtained:

$$Y = \begin{bmatrix} 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 & 4080 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (3.37)$$

Only DC coefficients are nonzero, since input macroblock has constant sample values and therefore frequency terms in DCT are eliminated. In intra16x16 mode, DC coefficients are Hadamard transformed to be decorrelated:

$$Y_{DC} = \begin{bmatrix} 4080 & 4080 & 4080 & 4080 \\ 4080 & 4080 & 4080 & 4080 \\ 4080 & 4080 & 4080 & 4080 \\ 4080 & 4080 & 4080 & 4080 \end{bmatrix} \tag{3.38}$$

$$Z_{DC} = \tfrac{1}{2} H Y_{DC} H^T = \begin{bmatrix} 32640 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.39}$$

Since only DC coefficient in (0,0) position is nonzero, only it is quantized. The highest possible quantized coefficient value is obtained for *QP=0* for which quantization step size is minimum.

$$qbits = 15 + floor(QP/6) = 15 \tag{3.40}$$

$$MF_{(0,0)} = 13107 \text{ for } QP=0 \tag{3.41}$$

$$f = 2^{qbits}/3 = 10922.666 \tag{3.42}$$

$$|W_{DC}| = \left(|Z_{DC}| MF + 2f\right) / \left(2^{qbits+1}\right) = \begin{bmatrix} 6528.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 \end{bmatrix} \tag{3.43}$$

$$W_{DC} = \begin{bmatrix} 6528 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.44}$$

Similarly for macroblock with most negative (-255) values, -6528 is obtained as minimum DC coefficient value. Therefore transformed and quantized DC coefficient values are in the range (-6528, 6528) and represented by 16-bits. Chroma DC coefficients are also included in this range.

For prediction modes other than intra16x16 transformed and quantized luma DC coefficients are not separately transmitted as mentioned before. They are transmitted in the same blocks with other coefficients. The range of these coefficients can be similarly found by disabling Hadamard Transform and applying quantization simply for $QP=0$:

$$W_{DC} = \begin{bmatrix} 1632 & 1632 & 1632 & 1632 \\ 1632 & 1632 & 1632 & 1632 \\ 1632 & 1632 & 1632 & 1632 \\ 1632 & 1632 & 1632 & 1632 \end{bmatrix} \tag{3.45}$$

So the range for these coefficients is (-1632, 1632).

The limit for coefficients corresponding to frequency components can also be found applying transform & quantization to a macroblock with checkerboard pattern which has the highest possible frequency component value:

$$X = \begin{bmatrix}
255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 \\
-255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 \\
255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 \\
-255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 \\
255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 \\
-255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 \\
255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 \\
-255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 \\
255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 \\
-255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 \\
255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 \\
-255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 \\
255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 \\
-255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 \\
255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 \\
-255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255 & -255 & 255
\end{bmatrix} \tag{3.46}$$

This input macroblock is not very realistic for a practical H.264 encoder/decoder system but helps to find the theoretical limit for the frequency component range. After DCT is applied DC and AC components are obtained. DC terms are zero since input has zero mean:

$$Y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 & 0 & 1020 & 0 & 3060 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 & 0 & 3060 & 0 & 9180 \end{bmatrix} \quad (3.47)$$

After quantization is applied for *QP=0* case:

$$Y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 & 0 & 163 & 0 & 489 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 & 0 & 489 & 0 & 1469 \end{bmatrix} \quad (3.48)$$

Highest value in quantized coefficients is 1469. Therefore the range of AC coefficients is (-1469, 1469) and represented by 16-bits.

In order to obtain the limits of AC coefficients and DC coefficients for each QP and prediction modes different than intra16x16, the maximum value of transformed & quantized coefficients are calculated, as illustrated in Table 3-4. For *QP>21* luma DC coefficients (for prediction modes different than intra16x16) and all AC coefficients are in the range (-127,127) and therefore can be represented by 8-bits. For these *QP,* luma DC coefficients for intra16x16 prediction mode should still be represented by 16-bits. This "*QP* coverage" is advantageous because of less amount of access and storage required to access 8-bit coefficient values instead of 16-bit coefficient values.

Also for *QP>21* and intra16x16 prediction mode, luma DC coefficients are in the range (-512, 512) and can be represented by 9-bits signed representation. This effectively reduces arithmetic logic required to implement Hadamard Transform from 16-bits to 9-bits, for this *QP* range. *QP>33* would be a further limitation to obtain 8-bit representation also for luma DC coefficients in intra16x16 mode but that *QP* range would cause significant quality loss, considering the number of these coefficients compared to other coefficients in residual macroblock.

For a low power H.264 decoder, bitrate and complexity considerations are more important than quality. Baseline profile is mainly used for low-power decoder implementations [11, 28] instead of other profiles with high demand of processing power. A low-complex decoder with ability to decode low-bitrate, moderate quality bitstream is the description of low-power H.264 decoder for mobile application. Therefore input bitstream with higher *QP,* to support lower bitrate is not a surprise for a low-power H.264 decoder. A simple software decoder (which consists of simple motion estimation, transform & quantization operations) is designed to encode an input frame and decode the residual. Obtained decoded frame for QP=22 doesn't lose much in terms of subjective quality, as shown in Figure 3-5 and 3-6 and has PSNR-Y value of 36.62 dB. Therefore this limitation does not degrade performance of decoder in terms of quality.

Table 3-4. Maximum values of coefficients and PSNR for given *QP*

QP	Luma DC coeff. (intra16x16)	Luma DC coeff. (other)	AC coeff.	PSNR-Y (dB)
0	6528	1632	1469	49.22
1	5934	1483	1335	49.18
2	5021	1255	1130	48.80
3	4663	1166	1049	48.42
4	4080	1020	918	48.30
5	3627	907	816	48.20
6	3264	816	734	47.75
7	2967	742	667	47.57
8	2511	628	565	46.82
9	2331	583	524	46.41
10	2040	510	459	46.11
11	1813	453	408	45.61
12	1632	408	367	44.98
13	1483	371	334	44.58
14	1255	314	282	43.11
15	1166	291	262	42.25
16	1020	255	229	41.65
17	907	227	204	41.39
18	816	204	183	40.45
19	742	185	167	39.76
20	628	157	141	37.98
21	583	146	131	37.72
22	510	127	115	36.62
23	453	113	102	36.15
24	408	102	92	35.4
25	371	93	83	34.65
26	314	78	70	33.19
27	291	73	65	32.95
28	255	64	57	31.88
29	227	57	51	31.20
30	204	51	46	30.46
31	185	46	42	30.14
32	157	39	35	29.00
33	146	36	33	28.34
34	127	32	29	27.84
35	113	28	25	27.12
36	102	25	23	26.44
37	93	23	21	26.12
38	78	19	17	25.42
39	73	18	16	24.86
40	64	16	14	24.06
41	57	14	13	23.77
42	51	13	11	23.49
43	46	11	10	22.98
44	39	10	9	22.57
45	36	9	8	22.05
46	32	8	7	21.45
47	28	7	6	20.88
48	25	6	6	20.74
49	23	6	5	20.09
50	19	5	4	18.69
51	18	4	4	18.33

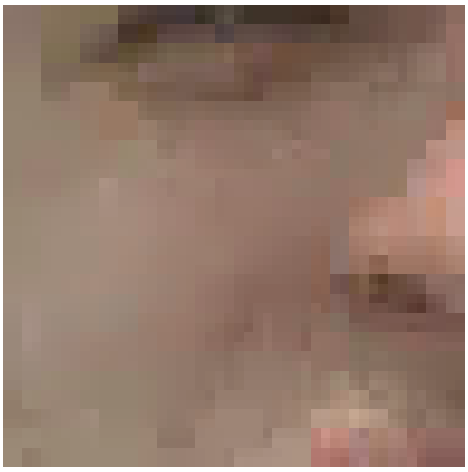

Figure 3-5. Part of 4:2:0 CIF input frame taken from “foreman” sequence

Figure 3-6. Output of decoder for QP=22 (left) and QP=35 (right)

3.4.2 System Architecture

During the design process of inverse transform & quantization architecture, proposed QP limitation is used to obtain a low-complexity design. Input memory organization and complexity of input buffers and Hadamard Transform unit is directly affected from this limitation.

Inverse transform & quantization architecture is implemented on Spartan-3 series XC3S2000 FPGA on evaluation board. Features of the evaluation board are explained in Appendix B.

System architecture is as illustrated in Figure 3-7. Input CIF frame of transformed and quantized coefficients, with prediction modes and QP are sent through serial channel to on-chip embedded processor and then loaded to external SRAM. Then each macroblock is read from SRAM and written to a internal dual port memory where inverse transform & quantization hardware reads input coefficients, processes them and writes the result back to another internal dual-port memory. Obtained results are loaded back to SRAM and finally complete frame is sent back to PC. Serial communication between the board and PC is handled by software designed in MATLAB®. Embedded processor on FPGA is used to control the flow of data between inverse transform & quantization hardware and PC.

Inverse transform & quantization hardware is the only component in the system architecture which is designed to be used in a complete H.264 hardware decoder. Other components are just used to control the flow of data between inverse transform & quantization hardware and software. Actually their purpose is to emulate the data transfer between inverse transform & quantization unit and other processing blocks in a complete H.264 decoder architecture. Therefore their performance in terms of memory use, processing speed, gate count etc. is ignored during evaluation of inverse transform & quantization performance.

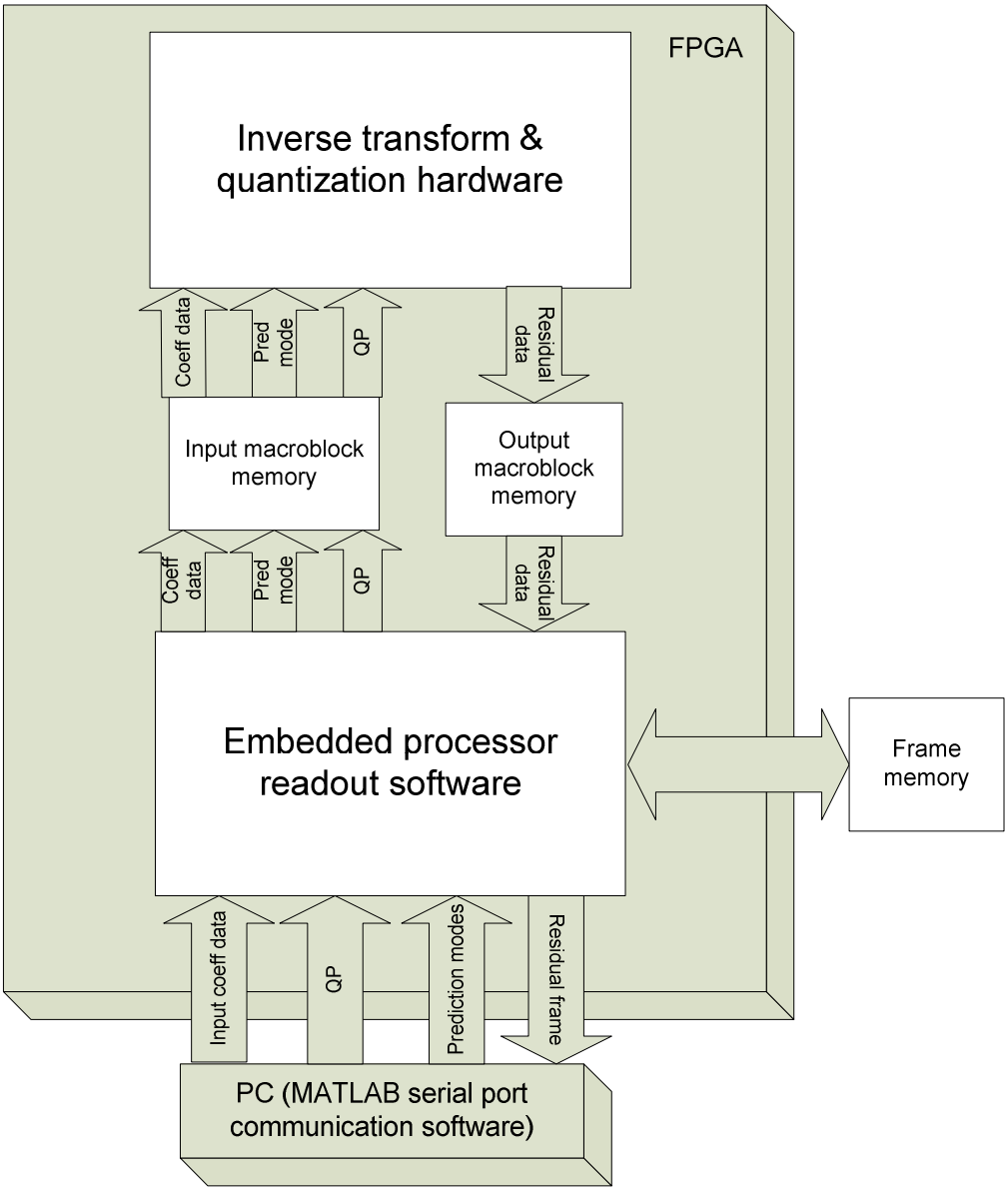

Figure 3-7. System architecture

3.4.3 Preparation of Input Macroblock Coefficients

In order to obtain transformed and quantized macroblock coefficients to be processed by hardware, software is developed in MATLAB®. Transform and quantization of a macroblock with given *QP* and prediction mode (whether it is intra16x16 or not) is implemented and obtained transformed and quantized coefficients, prediction mode and *QP* are then sent to hardware. The software simply implements the processing flow as illustrated before in Figure 3-1.

3.4.4 Serial Data Transfer

Transformed & quantized coefficient data, prediction mode type and *QP* of a single CIF frame are sent through serial port using serial communication software. Transformed and quantized luma DC coefficients, luma coefficients, chroma DC coefficients, chroma coefficients, quantization parameter and prediction mode are sent in sequential order. The software running on embedded processor inside FPGA reads the frame data and send acknowledge for each data group to control the flow.

The purpose of embedded processor implemented in FPGA is just to control the data transfer between serial communication software and designed inverse transform & quantization hardware with its UART and memory controller interfaces. Embedded processor reads a frame of quantized coefficients, prediction modes of macroblocks in the frame and *QP* from UART. This information is loaded to external SRAM on board. For each macroblock, luma and chroma quantized coefficients, corresponding prediction mode and *QP* is read from SRAM by embedded processor and loaded to input memory for processing. Inverse transform & quantization hardware is triggered to do the processing and results are read from output memory and loaded back to SRAM. After all data of a frame is processed, the results residing in SRAM are sent back to PC through serial channel.

Embedded processor software has no task assigned related to any signal processing, since pure hardware solution is to be designed for inverse transform & quantization.

3.4.5 Memory Organization

There are three types of memory in the system to implement inverse transform & quantization of a complete CIF frame.

Input and output memories are used to give quantized coefficient macroblock as an input to the inverse transform & quantization hardware and store the output to be read by embedded processor and sent back to PC. They are dual-port memories inserted to allow concurrent access from both embedded processor and inverse transform & quantization hardware. With proposed representation for input data, that is, 16-bit representation for DC coefficients and 8-bit representation for other coefficients, for a single macroblock input and control parameters, input and output memories are organized as follows:

16 Luma DC coefficients: $16 \cdot 16 = 256\ bits = 32 bytes$
256 Luma coefficients: $256 \cdot 8 = 256\ bytes$
8 Chroma DC coefficients: $8 \cdot 16 = 128\ bits = 16 bytes$
128 Chroma coefficients: $128 \cdot 8 = 128\ bytes$
Quantization parameter: 1 byte
Prediction mode: 1 byte
Total required memory size = $32 + 256 + 16 + 128 + 2 = 434 \Rightarrow 512$ bytes

Table 3-5. Input and output memory organization

Data type	Memory window base address	Memory window size
Luma coefficients	0x0	256bytes
Chroma coefficients	0x100	128bytes
Luma DC coefficients	0x180	32bytes
Chroma DC coefficients	0x1A0	16bytes
Quantization parameter	0x1B0	16bytes (1byte is used)
Prediction mode	0x1C0	16bytes (1byte is used)

Input and output memories have 32-bit access for both ports. Since DC coefficients are represented by 16-bits, 2 DC coefficients are accessed at each read/write transaction. On the other hand, 8-bit representation is proposed to be used for this architecture so that, 4 coefficients are accessed at each read/write access. These dual-port memories are constructed using the limited memory resources inside FPGA. Core Generator® tool is used for this purpose. Their ports are directly connected to inverse transform & quantization hardware and memory controller of embedded processor. Detailed information about memory resources of FPGA is given in Appendix A.

Frame memory is an external SRAM device connected to FPGA. It is a single port 2MB memory used to store complete CIF (352x288) frame coefficient data. Data stored in frame memory is organized as follows:

Number of macroblocks: $22 \cdot 18 = 396$

16 Luma DC coefficients for each MB: $32bytes \cdot 396 = 12kbytes$

256 Luma coefficients for each MB: $256bytes \cdot 396 = 96kbytes$

8 Chroma DC coefficients for each MB: $16bytes \cdot 396 = 6kbytes$

128 Chroma coefficients for each MB: $128bytes \cdot 396 = 48kbytes$

Quantization parameter: $1byte$

Prediction mode of each MB = $396bytes$

Table 3-6. Frame memory organization

Data type	Memory window base address	Memory window size
Luma coefficients	0x0	1MB (96KB is used)
Chroma coefficients	0x100000	512KB (48KB is used)
Luma DC coefficients	0x180000	256KB (12KB is used)
Chroma DC coefficients	0x1C0000	128KB (8KB is used)
Prediction modes	0x1E0000	64KB (396 bytes is used)
QP	0x1F0000	16bytes (1byte is used)

3.4.6 Inverse Transform & Quantization Architecture

The architecture consists of processing units which are responsible for data transfer, inverse transform and inverse quantization. Block diagram of designed architecture is illustrated in Figure 3-8.

3.4.6.1 Input Data Buffering Unit

Input data buffering is the most complex and critical part of inverse transform & quantization architecture. The reason is that it should read input coefficients from input memory in transmission order, load them to a buffer, trigger Hadamard Transform Unit and send DC coefficients for intra16x16 prediction mode, read the results back and replace DC coefficients in the buffer with transformed ones. Also this unit handles luma and chroma cases for loading input coefficients and Hadamard transform selection.

There is a 32-bit interface between input data buffering unit and dual-port input memory. In transmission order of a macroblock, DC coefficients of luma samples are firstly transmitted. Then other luma coefficients and DC coefficients of chroma samples are transmitted. Finally DC coefficients of chroma samples are transmitted.

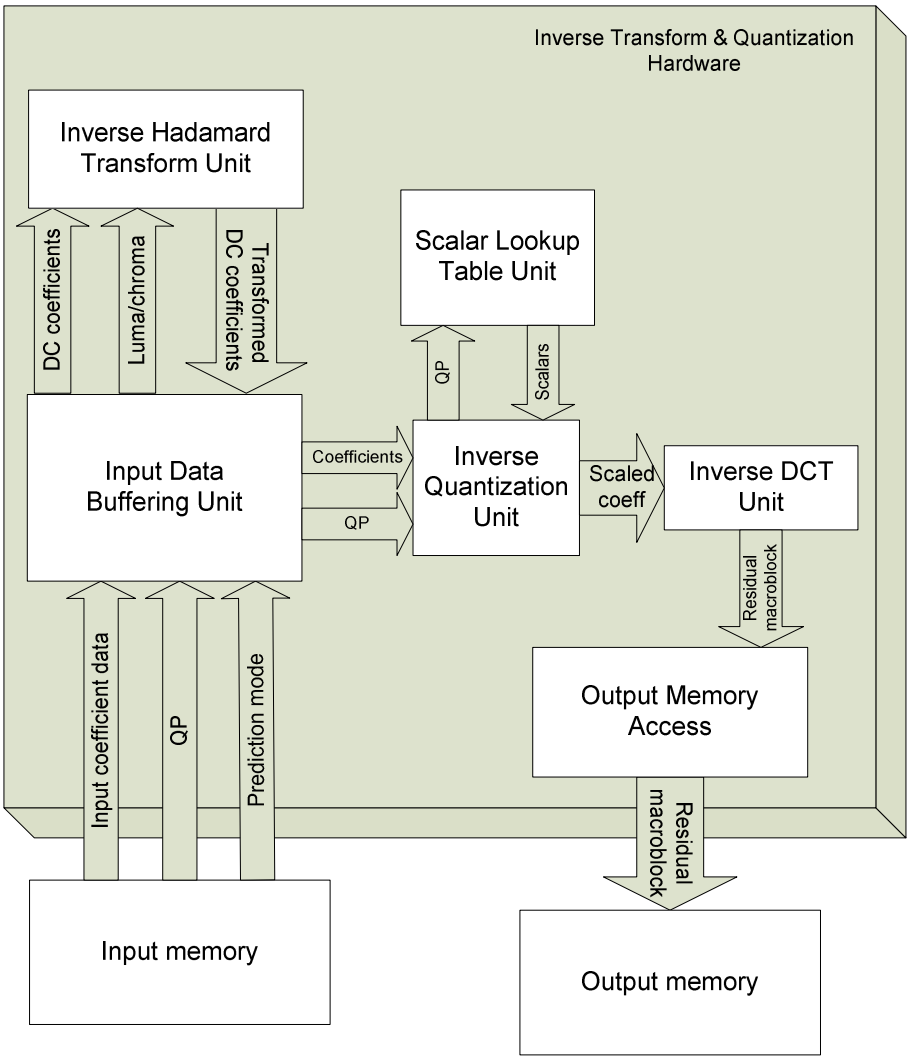

Figure 3-8. Processing Hardware Architecture

Input coefficients are read from different locations of input memory in "transmission order" defined by the standard [6] and inverse transform & quantization results are written to assigned locations of output memory in the same order. By this way input and output buffering units obey transmission order for controlling internal processing units. So the design can easily be immigrated to a complete H.264 hardware decoder system with small changes in input buffering unit to provide interface to other processing blocks in decoder hardware.

Input buffering unit is mainly composed of two state machines. The first one reads input coefficients from input memory in transmission order. It handles the addressing required to read input memory for obeying transmission order. Coefficients are stored by embedded processor to input memory in the organization, as previously shown in Table 3-5. The addressing control reads from these locations in transmission order.

The second state machine loads read input data to a buffer, called *input_buffer*, controls the Hadamard Transform Unit to start the right transform operation (for 4x4 luma or 2x2 chroma) and reads transformed coefficient results. *input_buffer* is actually a shift-register filled by shifting read input coefficients. in*put_buffer* has 160-bit size used for different coefficient types. 10-bit representation is used for luma DC coefficients in intra16x16 prediction mode, since *QP>21*. 10-bit Luma DC coefficient block are loaded to this buffer as shown in Figure 3-9.

DC_coeff(0,0)	DC_coeff(0,1)	DC_coeff(0,2)	DC_coeff(0,3)
DC_coeff(1,0)	DC_coeff(1,1)	DC_coeff(1,2)	DC_coeff(1,3)
DC_coeff(2,0)	DC_coeff(2,1)	DC_coeff(2,2)	DC_coeff(2,3)
DC_coeff(3,0)	DC_coeff(3,1)	DC_coeff(3,2)	DC_coeff(3,3)

DC_coeff(0,0)	DC_coeff(0,1)	DC_coeff(0,2)	DC_coeff(0,3)	….	DC_coeff(3,2)	DC_coeff(3,3)
10-bit	10-bit	10-bit	10-bit		10-bit	10-bit

Figure 3-9. Upper: Input DC coefficient block. Lower: 160-bit buffer content

If Hadamard Transform is required, for intra16x16 prediction mode or chroma, Hadamard Transform Unit is triggered and reads *input_buffer* content. Transformation result is from a different 160-bit shift-register called *hadamard_buffer*. Then 8-bit represented 4x4 Luma coefficients are read and loaded to 128-bit (16x8-bit coefficients) portion of *input_buffer*. The first 10-bit entry of *input_buffer*, which is the DC coefficient position, is replaced by corresponding 10-bit *hadamard_buffer* entry. Obtained 136-bit content is read by inverse quantization and transform units for further processing. For the next 4x4

block, new coefficient values are loaded to *input_buffer* and *hadamard_buffer* is shifted 16 bits to point next corresponding DC coefficient for replacement. This continues until all luma blocks are read and DC coefficients are replaced with Hadamard transformed ones. For prediction modes other than intra16x16, Hadamard Transform Unit is not triggered for luma and loaded 128-bit *input_buffer* content is directly used by other processing units.

3.4.6.2 Inverse Hadamard Transform Unit

As previously mentioned, 4x4 luma DC coefficients are Hadamard transformed at encoder side for macroblocks predicted in intra16x16 mode. Transformed DC coefficients are inverse transformed at the decoder side, which is exactly the same expression with forward transform. 2x2 inverse Hadamard transform is also used for chroma DC coefficients. A state machine is designed to support both transforms and controlled by input buffering unit. It lasts only 2 clock cycles to finish the transform operation and send ready signal *hadamard_done*, to controlling state machine of input buffering unit.

As previously mentioned, 4x4 inverse Hadamard transform is simply a matrix transformation operation:

$$W_{QDC} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} [Z_{DC}] \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \tag{3.49}$$

For the implementation, this multiplication is splitted into two parts as (3.50) and (3.51):

$$W_{QDC}{}' = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} [Z_{DC}] \tag{3.50}$$

$$W_{QDC} = W_{QDC}{}' \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \tag{3.51}$$

Obtained (3.50) is a single matrix multiplication and can be implemented by 4 different vector multiplications such as:

$$W_{QDC(0,k)}{}' = [1 \quad 1 \quad 1 \quad 1] \begin{bmatrix} Z_{DC(0,k)} & Z_{DC(1,k)} & Z_{DC(2,k)} & Z_{DC(3,k)} \end{bmatrix}^T \tag{3.52}$$

$$W_{QDC(1,k)}{}' = [1 \quad 1 \quad -1 \quad -1] \begin{bmatrix} Z_{DC(0,k)} & Z_{DC(1,k)} & Z_{DC(2,k)} & Z_{DC(3,k)} \end{bmatrix}^T \tag{3.53}$$

$$W_{QDC(2,k)}{}' = [1 \quad -1 \quad -1 \quad 1] \begin{bmatrix} Z_{DC(0,k)} & Z_{DC(1,k)} & Z_{DC(2,k)} & Z_{DC(3,k)} \end{bmatrix}^T \tag{3.54}$$

$$W_{QDC(3,k)}{}' = [1 \quad -1 \quad 1 \quad -1] \begin{bmatrix} Z_{DC(0,k)} & Z_{DC(1,k)} & Z_{DC(2,k)} & Z_{DC(3,k)} \end{bmatrix}^T \tag{3.55}$$

where $k = 0,1,2,3$

Totally 16 vector multiplications are required for (3.50). These are implemented using 10-bit addition structures illustrated in Figure 3-11. Coefficient values are loaded into 10-bit registers and additions are performed on these registers. 4 copies of each of these structures are used to completely implement (3.50). In (3.51) matrix multiplication is exactly the same with (3.50) and therefore copies of the same addition structures are used. All additions for (3.50) and (3.51) are performed with parallel structures with reduced adder count.

In order to make Hadamard transform controllable by input buffering unit, a state machine is designed, as illustrated in Figure 3-10. Hadamard transform operation starts when *dc_ready* signal is asserted by input buffering unit when buffering of

DC coefficients from input memory is completed. When this signal is detected by Hadamard transform unit, it is checked whether they are luma or chroma DC coefficients. If they are luma coefficients, (3.50) is implemented using addition structures with 10-bit registers loaded with coefficients and results are loaded to 10-bit registers. In the next state, (3.51) is implemented on these registers with similar structures. The final result is loaded to a 160-bit shift-register, *hadamard_buffer* (16x10-bit) to be read by input buffering & control unit with asserting *hadamard_done* signal. If 2x2 transformation is to be done for chroma, smaller addition structures with 2 entries are used and the result is loaded to 40-bit (4x10-bit) portion of *hadamard_buffer*.

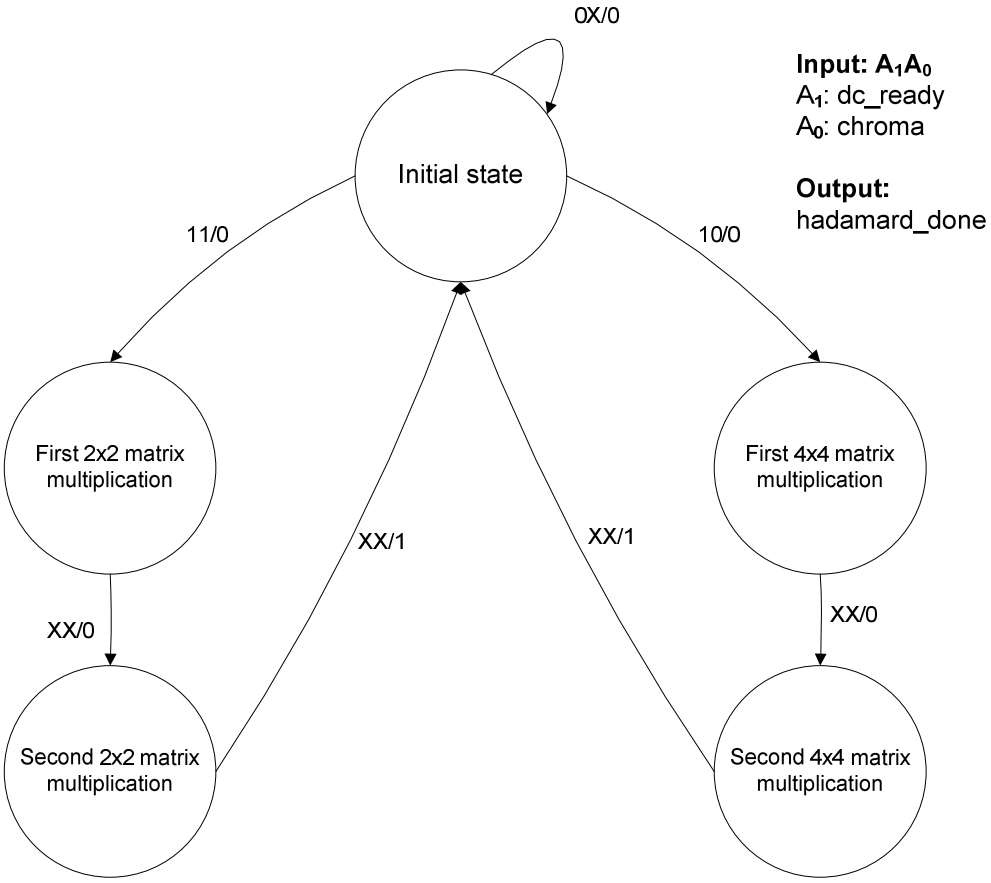

Figure 3-10. Inverse Hadamard transform unit state transition diagram

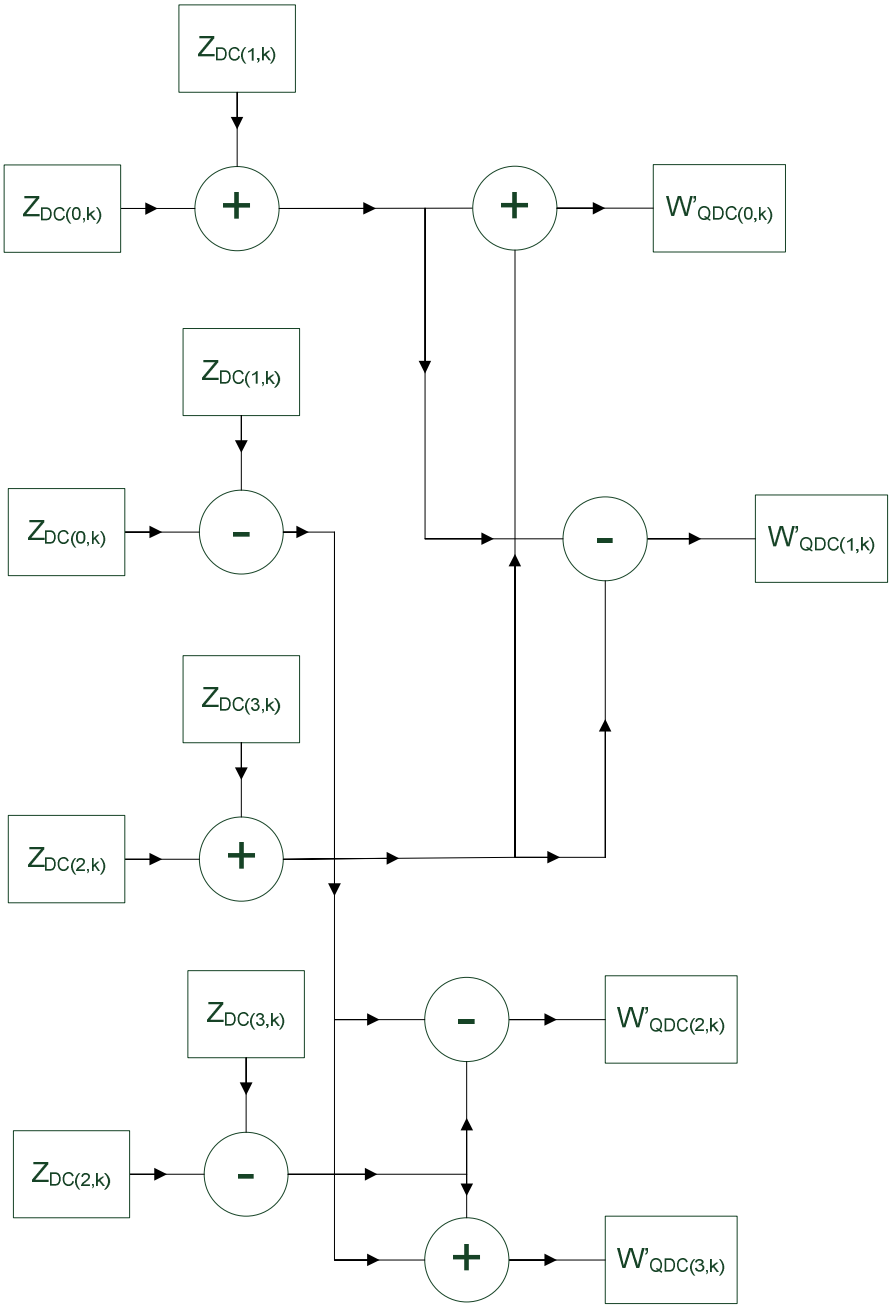

Figure 3-11. Addition structures for (3.50)

3.4.6.3 Inverse Quantization Unit

Inverse quantization is a scaling operation to obtain transformed residual samples from quantized coefficients. Scaling factors depend on the *QP* and matrix position of the coefficient to be scaled. Derivation of inverse quantization expression is previously explained in detail.

There are 3 different V values corresponding to 3 different matrix positions and *QP<6*, as illustrated in Table 3-3. In the implementation V_{ij} in (3.20) is combined with exponent term to form:

$$V_{ij}' = V_{ij} 2^{floor(QP/6)} \tag{3.56}$$

$$W_{ij}' = Z_{ij} V_{ij}' \tag{3.57}$$

The contents of V_{ij}' are named as *V1*, *V2* and *V3* where;

$$V' = \begin{bmatrix} V1 & V3 & V1 & V3 \\ V3 & V2 & V3 & V2 \\ V1 & V3 & V1 & V3 \\ V3 & V2 & V3 & V2 \end{bmatrix} \tag{3.58}$$

For inverse quantization of luma and chroma DC coefficients, expressions in (3.30) and (3.31) reduce to (3.59), (3.34) and (3.35) reduce to (3.60) respectively and easily implemented with same structure used for other coefficients:

$$W_{ij}' = Z_{ij} V1/4 \tag{3.59}$$

$$W_{ij}' = Z_{ij} V1/2 \tag{3.60}$$

V Values for *QP>6*, are obtained by doubling for each *QP* increment of 6. In order to get rid of shifting operation depending on the *QP*, which would slow the operation, lookup tables are formed by all *QP* values. These LUTs are combined in a scalar lookup table unit. It consists of 3 lookup tables for *V1*, *V2* and *V3* values. All *V* values can be read from lookup table without any delay. Structure of the scalar lookup table unit is as illustrated in Figure 3-12. *V1*, *V2* and *V3* values are represented by 18-bit entries and selected by 6-bit *QP* input.

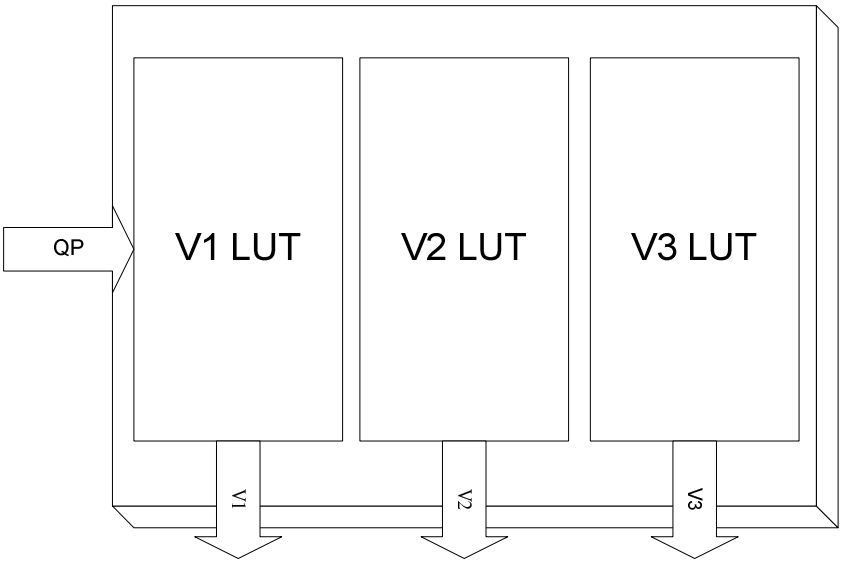

Figure 3-12. Scalar look-up table unit

Multiplication factors read from LUTs and buffered coefficients are loaded to 4 hardware multiplier units. Hardware multipliers are limited resources of FPGAs that can be used in arithmetic building blocks of designed architectures. They have 18-bit inputs, 36-bit output and have capability to handle signed numbers represented in 1's complement form. They are combinatorial elements with zero arithmetic processing delay to produce multiplication result. Throughout the inverse transform & quantization architecture, 2's complement representation is used. For multiplication, coefficients are converted 1's complement representation and multiplication results are then converted back to 2's complement form by a simple logic.

Two controller state machines are designed to handle inverse quantization of 4x4 blocks. The state transition diagram for these state machines is as illustrated in Figure 3-14. The first state machine is triggered by Input Buffering Unit when buffering is completed for each buffered 4x4 coefficient block (with replaced Hadamard transformed DC coefficients for intra16x16 prediction mode). In each clock cycle, 4 coefficients are loaded to multiplier registers and obtained results are loaded registers simultaneously. Since there are 16 coefficients in each 4x4 block, 4 clock cycles is required to obtain inverse quantization result of a 4x4 coefficient block. Since *V3* is used twice times more than others 2 multipliers are used for *V3* multiplication as illustrated in Figure 3-13. Second state machine is triggered at the same time with first state machine and synchronized to read the multiplication results from multipliers and load them to registers with triggering Output Memory Access unit at each 4 clock cycle. Simulation results for multiplication and loading of results to registers is shown in Figure 3-15 and 3-16.

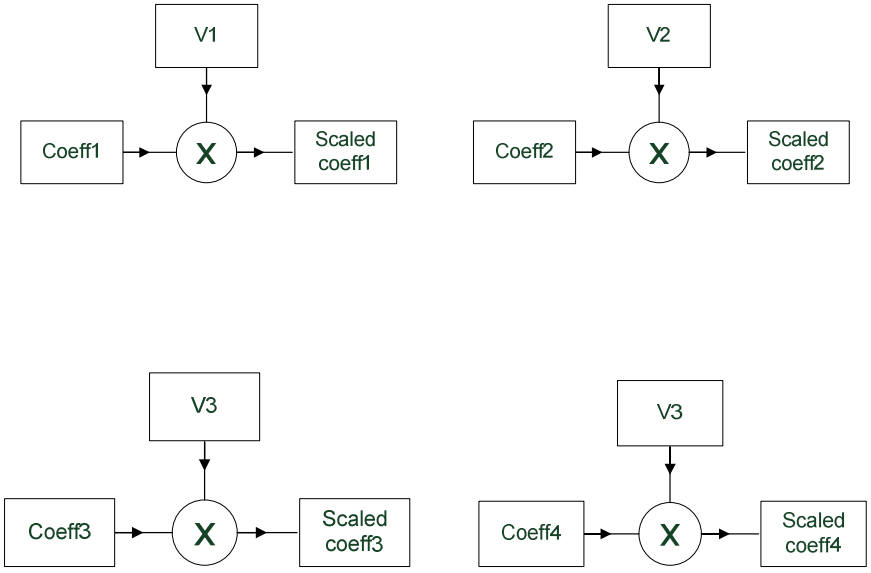

Figure 3-13. Multipliers for scaling coefficients

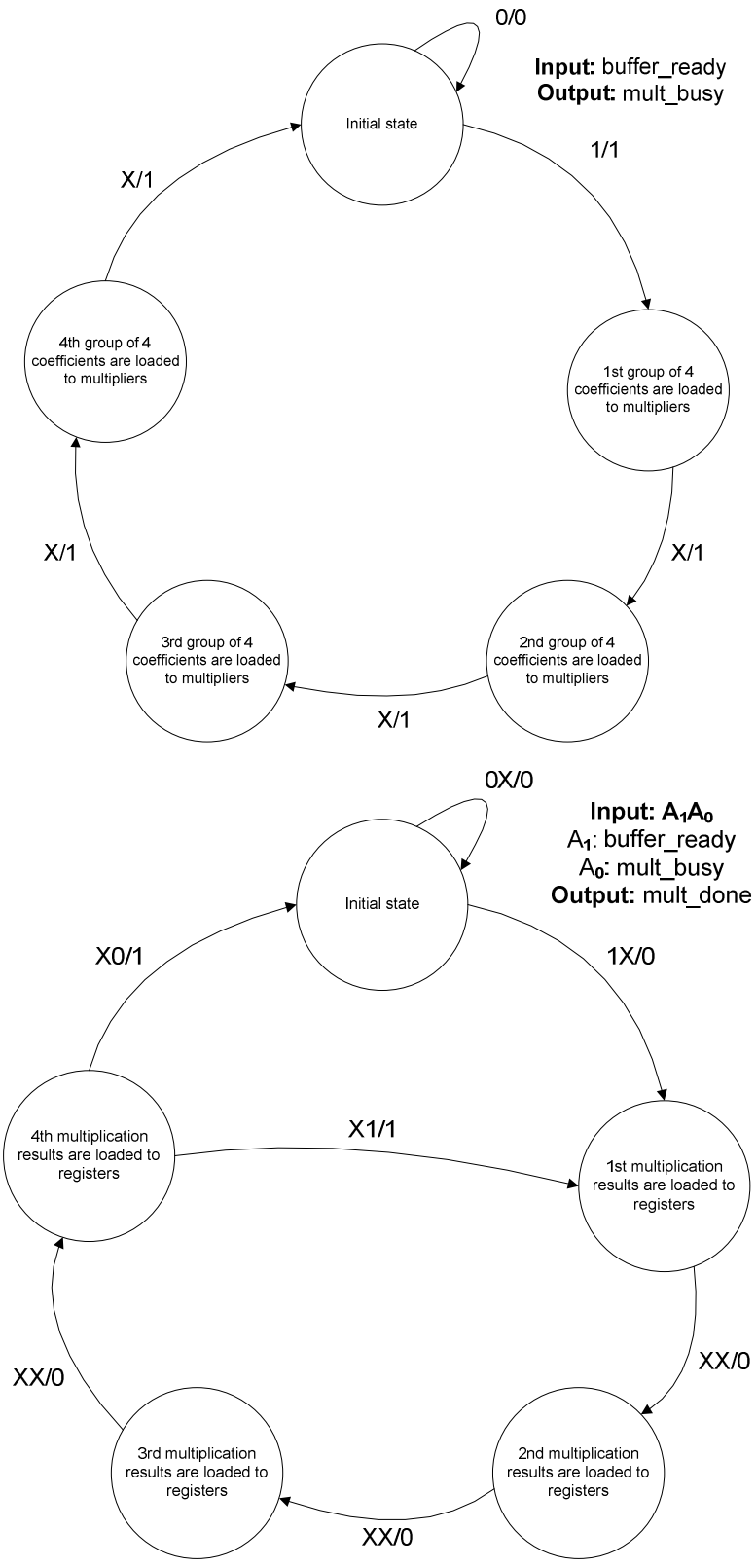

Figure 3-14. Controlling state machines for inverse quantization

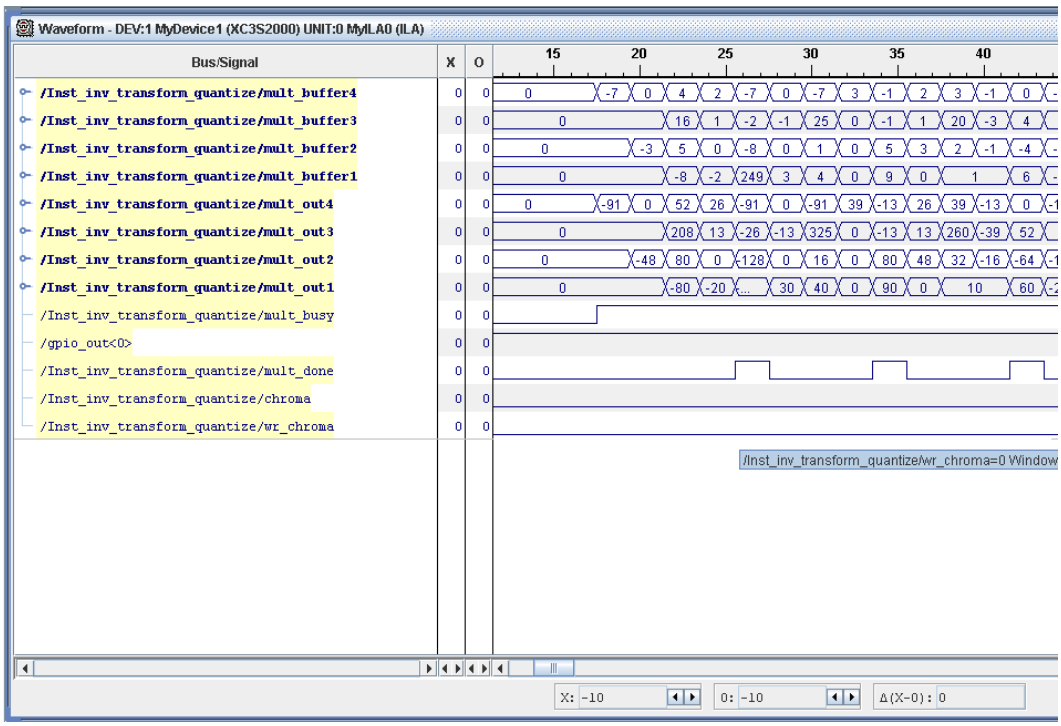

Figure 3-15. Multiplication with *V* scalars

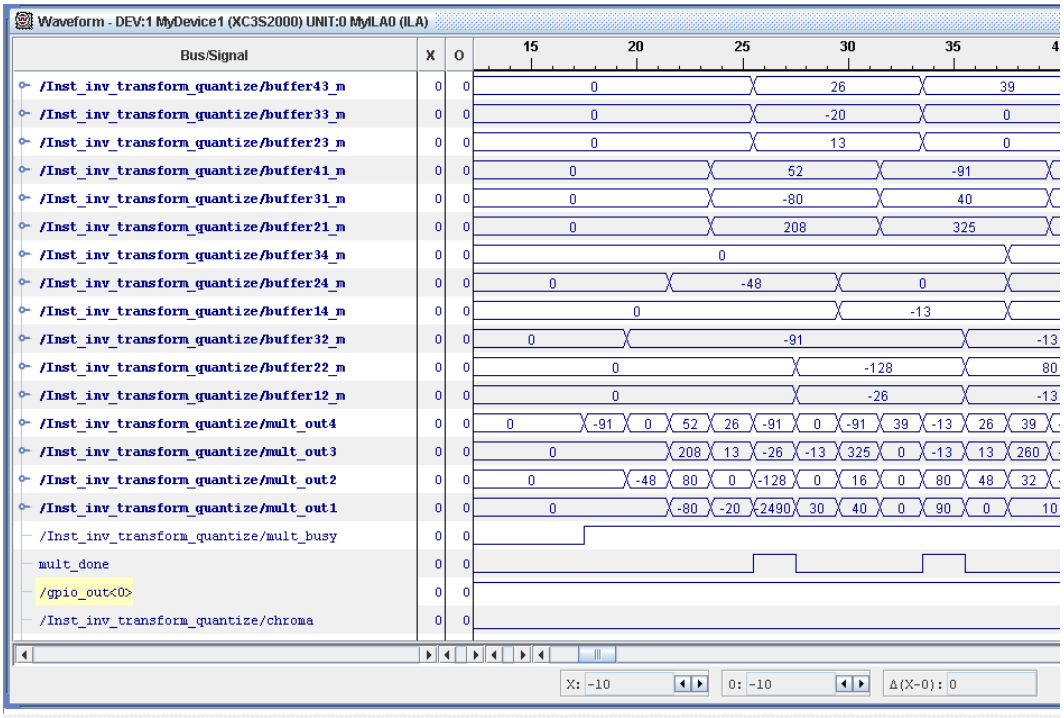

Figure 3-16. Loading registers with multiplication results

3.4.6.4 Inverse Integer DCT Unit

Integer form of Discrete Cosine Transform is used by H.264 to ease the hardware implementation and provide exactly equal results on different processing platforms which is not the case for floating-point operations. Derivation of integer DCT is explained in previous sections.

For each 4x4 coefficient block, Integer DCT is implemented using similar arithmetic structures with the ones used for Hadamard transform. Required multiplications by $1/2$ are implemented by left-shift operations. Since coefficients may have negative values they are represented in 2's complement form and therefore shifting operation is performed in 2's complement form.

Scalar multiplication with E_i is integrated into inverse quantization operation. The remaining "core inverse transform" is implemented by designed inverse transform unit:

$$X = C_i^T Y_{scaled} C_i = \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} Y_{scaled} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix} \tag{3.61}$$

As it is performed for inverse Hadamard transform, core inverse transform expression is also splitted into two matrix multiplications for implementation:

$$X' = C_i^T Y_{scaled} = \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} \left[Y_{scaled} \right] \tag{3.62}$$

$$X = X'C_i = X' \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix} \tag{3.63}$$

(3.62) is a single matrix multiplication and can be implemented by 4 different vector multiplications such as:

$$X'_{(0,k)} = \left[1 \quad 1 \quad 1 \quad 1/2 \right] \left[Y_{scaled(0,k)} \quad Y_{scaled(1,k)} \quad Y_{scaled(2,k)} \quad Y_{scaled(3,k)} \right]^T \tag{3.64}$$

$$X'_{(1,k)} = \left[1 \quad 1/2 \quad -1 \quad -1 \right] \left[Y_{scaled(0,k)} \quad Y_{scaled(1,k)} \quad Y_{scaled(2,k)} \quad Y_{scaled(3,k)} \right]^T \tag{3.65}$$

$$X'_{(2,k)} = \left[1 \quad -1/2 \quad -1 \quad 1 \right] \left[Y_{scaled(0,k)} \quad Y_{scaled(1,k)} \quad Y_{scaled(2,k)} \quad Y_{scaled(3,k)} \right]^T \tag{3.66}$$

$$X'_{(3,k)} = \left[1 \quad -1 \quad 1 \quad -1/2 \right] \left[Y_{scaled(0,k)} \quad Y_{scaled(1,k)} \quad Y_{scaled(2,k)} \quad Y_{scaled(3,k)} \right]^T \tag{3.67}$$

where $k = 0,1,2,3$

Totally 16 vector multiplications are required for (3.62). These are implemented using addition structures, including shift operations, as illustrated in Figure 3-17. Coefficient values are loaded into 16-bit registers and addition operations are performed on these registers. 4 copies of each of these structures are used to completely implement (3.62). (3.63) is also a single matrix multiplication and can be implemented as 4 different vector multiplications such as:

$$X_{(k,0)} = \begin{bmatrix} X'_{(k,0)} & X'_{(k,1)} & X'_{(k,2)} & X'_{(k,3)} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1/2 \end{bmatrix}^T \quad (3.68)$$

$$X_{(k,1)} = \begin{bmatrix} X'_{(k,0)} & X'_{(k,1)} & X'_{(k,2)} & X'_{(k,3)} \end{bmatrix} \begin{bmatrix} 1 & 1/2 & -1 & -1 \end{bmatrix}^T \quad (3.69)$$

$$X_{(k,2)} = \begin{bmatrix} X'_{(k,0)} & X'_{(k,1)} & X'_{(k,2)} & X'_{(k,3)} \end{bmatrix} \begin{bmatrix} 1 & -1/2 & -1 & 1 \end{bmatrix}^T \quad (3.70)$$

$$X_{(k,3)} = \begin{bmatrix} X'_{(k,0)} & X'_{(k,1)} & X'_{(k,2)} & X'_{(k,3)} \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 & -1/2 \end{bmatrix}^T \quad (3.71)$$

where $k = 0,1,2,3$

Similarly 16 vector multiplications are required for (3.63). These are implemented using similar structures, as illustrated in Figure 3-18. Results of (3.62) are loaded into 16-bit registers and addition operations are performed on these registers. 4 copies of each of these structures are used to completely implement (3.63). Obtained inverse transform result is loaded to a 256-bit buffer (16x16-bit).

For the design of inverse transform stages, simple 16-bit arithmetic is used with two stages of 1-D DCT operations by (3.62) and (3.63). There are algorithms in the literature that decrease logic utilization of DCT by using different techniques like bit-width reduction, division of matrix transformation into several stages etc. These techniques are effective but have negligible effect considering the total logic utilization of inverse transform & quantization architecture, therefore a simple implementation is used.

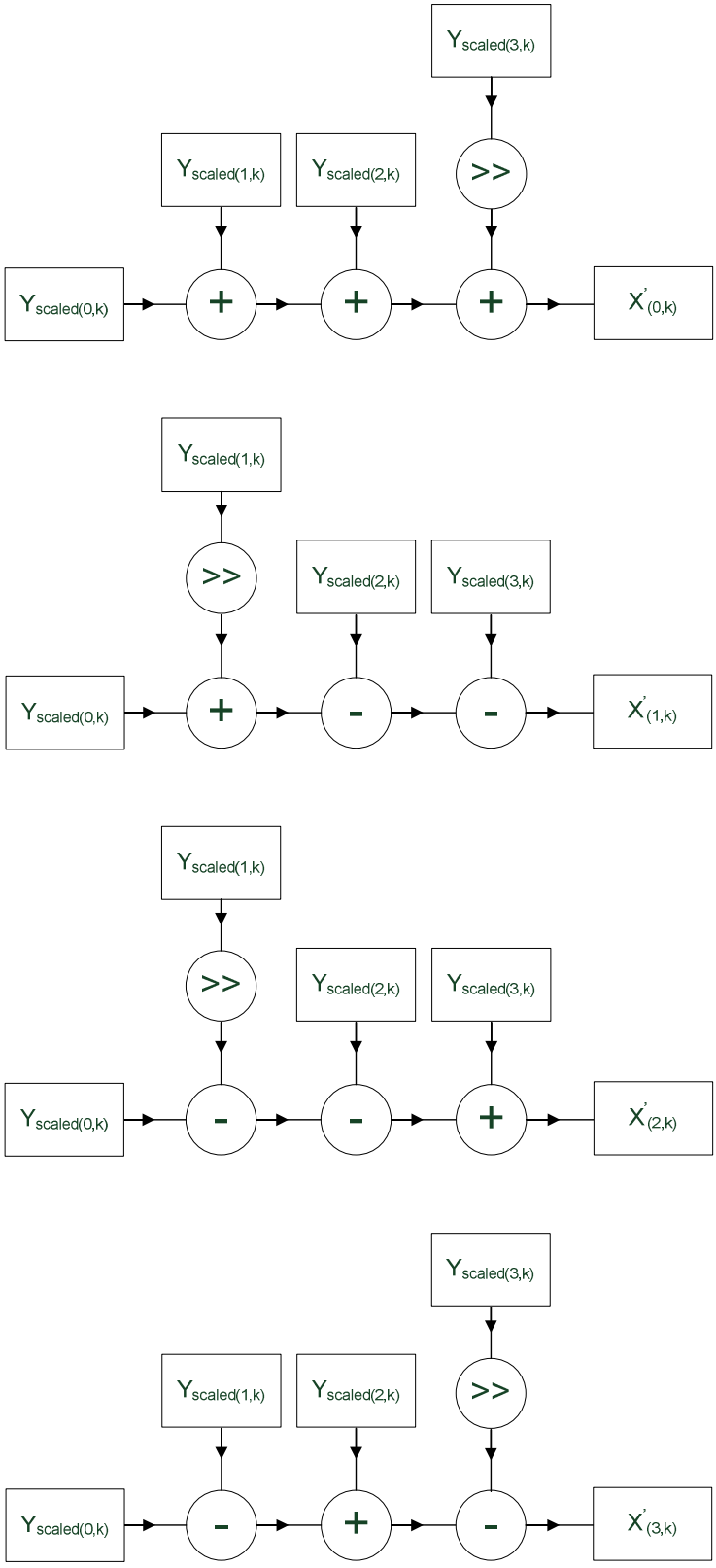

Figure 3-17. Addition structures for (3.62)

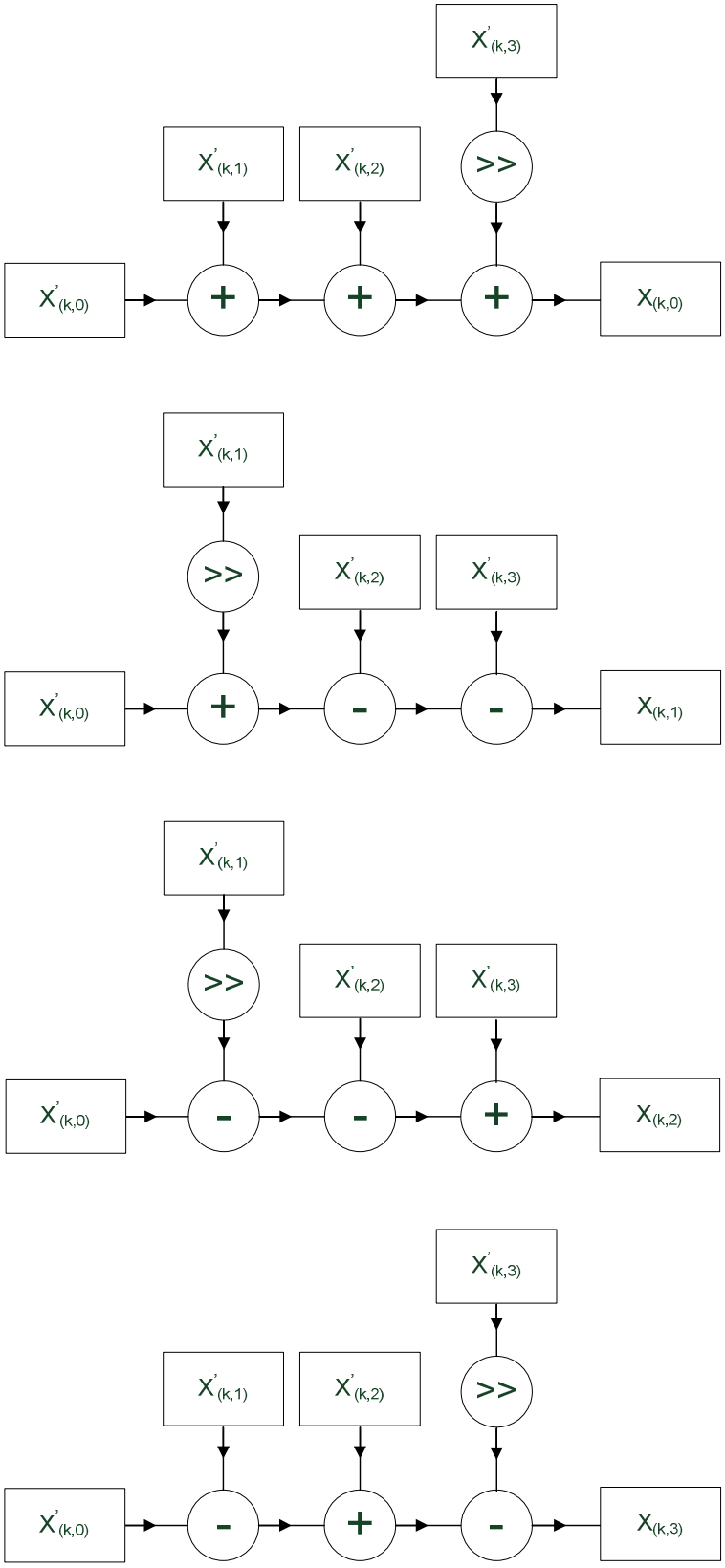

Figure 3-18. Addition structures for (3.63)

3.4.6.5 Output Memory Access Unit

The purpose of this unit is to write the final inverse transform results into output memory, to be read by embedded processor and sent to PC. Before loading inverse transform results they are post-scaled by 64 and rounded since they were pre-scaled in inverse quantization.

Inverse transformed samples are loaded to a 256-bit register containing 16 16-bit entries, as shown in Figure 3-19.

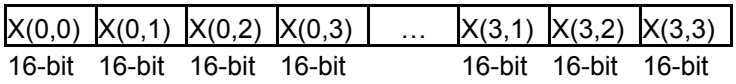

Figure 3-19. Inverse transformed samples (X) loaded to 256-bit register

Each 16-bit sample is right-shifted 6 times to obtain post-scaled (by 64) results, as illustrated in Figure 3-20.

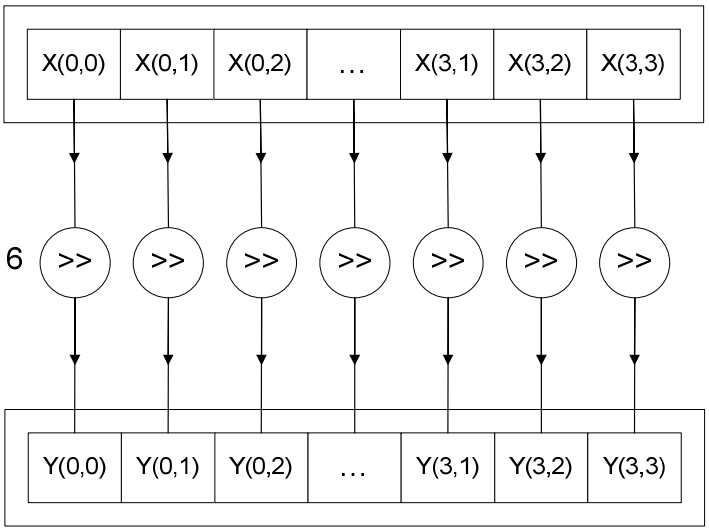

Figure 3-20. Right-shift operation to obtain post scaled samples

Right-shift operation is an easy way to implement division by powers of 2 but insufficient to implement rounding. In order to round post-scaled samples, a simple logic is designed to eliminate rounding errors caused by shift operation. Designed logic increments post-scaled values by 1 in conditions shown in Figure 3-21.

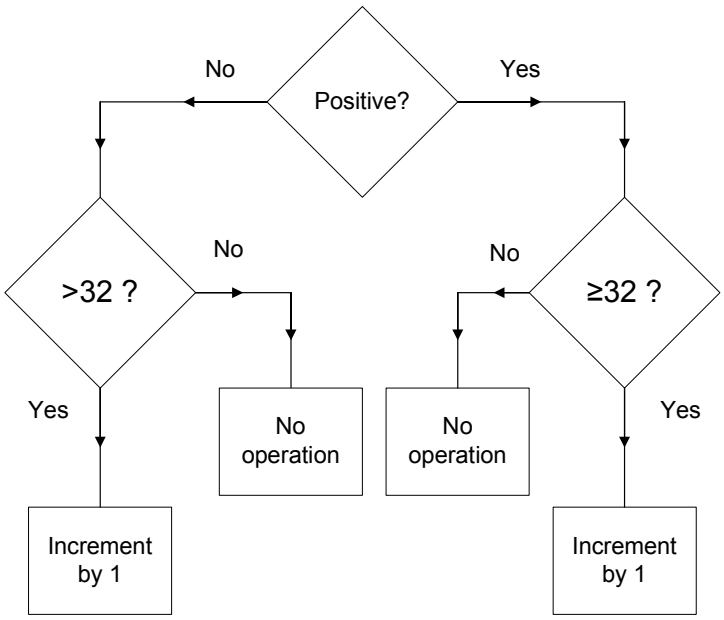

Figure 3-21. Condition check for rounding.

Obtained post-scaled and rounded results have 10-bit representation. For default 8-bit pixel representation, obtained results for residual samples is in the range (-255, 255). Therefore 9-bits are required to represent output samples in 2's complement representation. Designed output memory access is 32-bits. Modifying it to 36-bits to write 4 samples at each access would be a solution but 36-bit access is not supported by memory controller of embedded processor which is rarely used, whereas 8, 16, 32, 64-bit memory accesses are commonly used by various logic interfaces. Therefore output values are decided to be splitted into two parts, such as the sign bit and 8-bit value term. Value terms are grouped into 32-bit registers and written to memory at each clock cycle, synchronized with the inverse quantization state machine.

Sign bits of all luma residual samples are loaded to a 256-bit register and written to memory after 8-bit values of all luma residual samples are written, as shown in Figure 3-22. Since output memory access unit is synchronized with state machines which are responsible for multiplications in inverse quantization, it should not lose synchronization by writing sign information. Sign information is designed to be written to output memory after all luma samples are inverse quantized and transformed and chroma samples are started to be read from input memory. During that period inverse quantization state machines are in idle state waiting for input buffering to be completed and output memory access unit has enough time (8 clock cycles) to write 256-bit register containing sign bits. 8 clock cycles is enough because chroma DC coefficients are firstly read, inverse Hadamard transformed and first 4x4 chroma coefficient block is read before inverse quantization state machines start to operate leaving idle state.

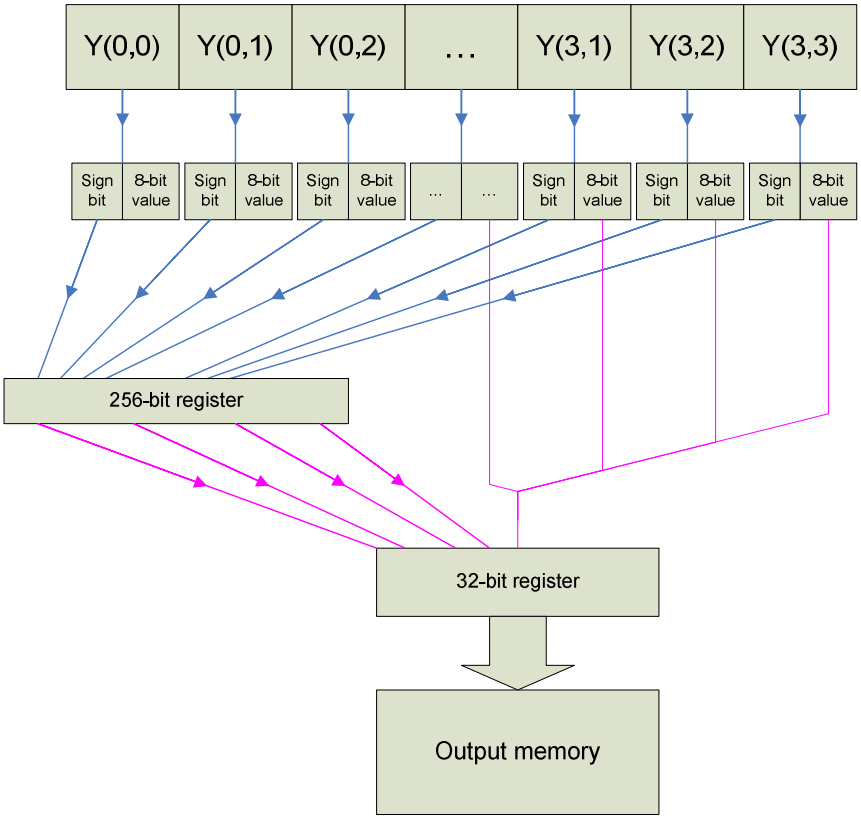

Figure 3-22. Write access for obtained residual samples

3.5 Results

Designed architecture is tested with serial communication software and verified by comparing the results with inverse transform & quantization software developed in MATLAB®.

Overall processing time for a single macroblock is 145 clock cycles for worst case (intra16x16 prediction mode) where Hadamard Transform is required for luma DC coefficients and 135 clock cycles for other cases. These are observed by sample differences between start of operation and completed signal assertion in Xilinx ChipScope® analysis software, shown in Figure 3-23 and 3-24. Obtained sample difference is divided by 2 since 66MHz clock is used as a sampling clock for observation of signals.

If *QP* was not limited with same design, almost twice more memory access would be required to read input coefficients. This would result higher logic utilization and higher power consumption.

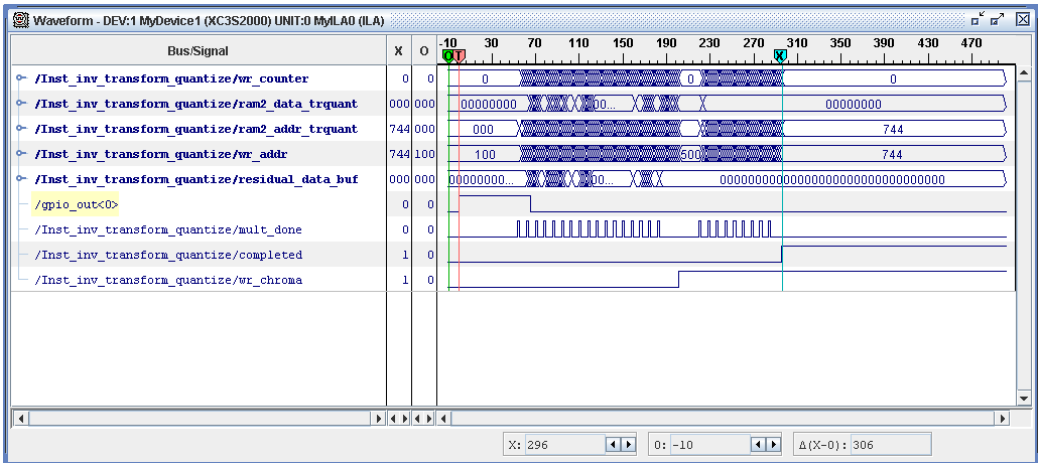

Figure 3-23. System performance evaluation for intra16x16 mode

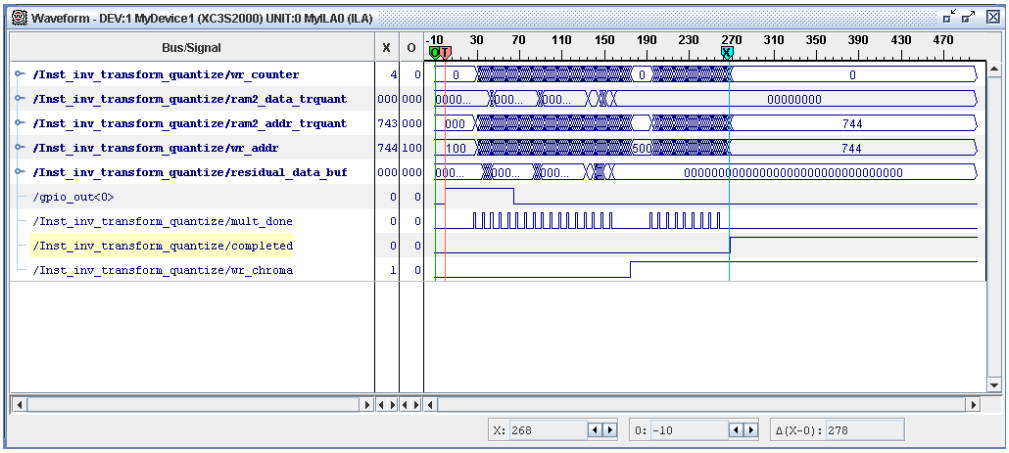

Figure 3-24. System performance evaluation for other modes

With this processing speed with 33MHz operating clock, a CIF (352x288) coefficient frame, which contains 396 macroblocks, can be processed in 1.72 ms. In other words, 581 CIF frames/second processing throughput can be achieved. Even for higher resolution of 1408x1152, a frame would be processed just in 27.52 ms that is a throughput of 36 fps.

For integrating designed system to a low-power H.264 decoder, the right clock frequency should be selected to support desired resolution and provide low power consumption. Some of the recommended clock frequencies that satisfy different resolutions are given in Table 3-7.

Table 3-7. Recommended operating clock frequency values

resolution	frame rate	clock frequency
176x144	30fps	500kHz
352x288	30fps	2MHz
704x562	30fps	8MHz
1408x1152	30fps	32MHz

Hardware utilization of system implemented in XC3S2000 series FPGA is illustrated in Figure 3-25. During evaluation of utilization, memory blocks and embedded processor are taken out from the design and only inverse transform & quantization architecture is synthesized and implemented by Xilinx ISE® software. Obtained values give a general idea of logic utilization of designed system but it is not right to compare it with an ASIC design, since it may give more accurate logic utilization results.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	1,612	40,960	3%	
Number of 4 input LUTs	6,150	40,960	15%	
Logic Distribution				
Number of occupied Slices	3,365	20,480	16%	
Number of Slices containing only related logic	3,365	3,365	100%	
Number of Slices containing unrelated logic	0	3,365	0%	
Total Number of 4 input LUTs	6,194	40,960	15%	
Number used as logic	6,150			
Number used as a route-thru	44			
Number of bonded IOBs	94	489	19%	
IOB Flip Flops	9			
Number of MULT18X18s	4	40	10%	
Number of GCLKs	1	8	12%	

Figure 3-25. Hardware utilization in FPGA

CHAPTER 4

DEBLOCKING FILTERING

4.1 Introduction

At first, when the output of H.264 decoder was analyzed, blocking artifacts were observed at the decoded frame. The artifacts are mostly due to the coarse quantization of transformed residual samples and imperfect fit of motion compensated blocks. An image-based transform would be a solution to reduce the artifacts but is already not suitable for block-processing video coding and requires frame buffering. These artifacts affect the compression performance in terms of subjective quality. In order to obtain a perceptually better decoded video sequence, output is filtered by an adaptive filter, named *post filter*, to reduce the blocking artifacts. It is called a post filter since it is applied after the decoding operation. On the other hand, since filtering provides a "better representation" of the frame, insertion as an *in-loop filter* for reference frame is found to be effective. Better reference frame provided better inter-picture prediction and therefore slightly improved objective quality and 5-10% bitrate. Deblocking filtering was an optional feature for H.263+, but is standardized in H.264 for in-loop filtering and is optional for decoder post-filtering. [15]

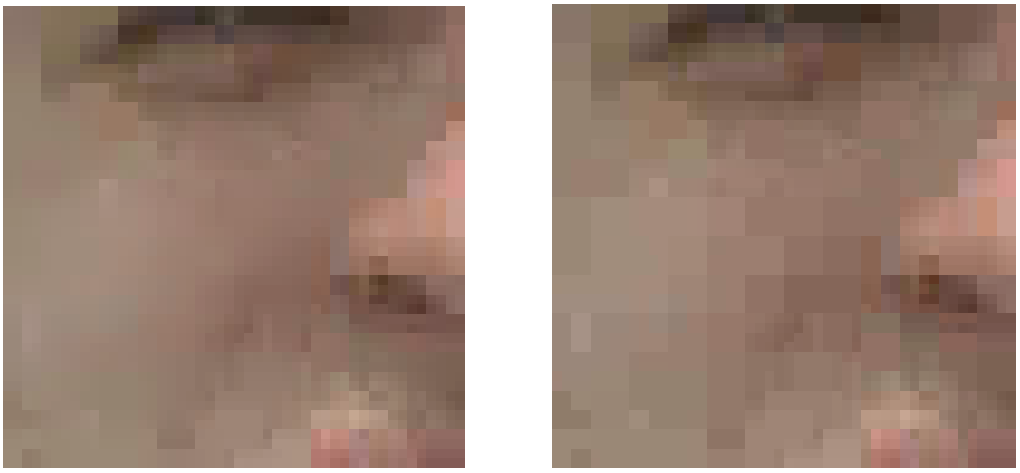

Figure 4-1. Original and reconstructed part of frame “foreman” for QP=28

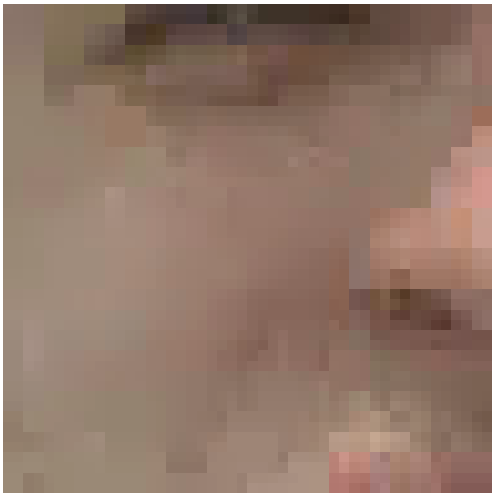

Figure 4-2. Part of frame after deblocking filtering

As illustrated in Figure 4-1, reconstructed frame has blocking artifacts compared to original frame mostly on smooth background and moving parts. Deblocking filter smoothes these accidental blocking edges and results a closer frame to original one, as shown in Figure 4-2. True edges on the frame are not affected because of the adaptive behavior of filtering.

4.2 Theoretical Background

In H.264 standard [6], deblocking filtering is applied to every 4x4 block edge in a macroblock. As shown in Figure 4-3 and Figure 4-4, filtering is applied to both horizontal and vertical edges. Luma and chroma block edges are independently filtered. Totally 32 luma (16 vertical and 16 horizontal edges) and 16 chroma block edges are filtered (8 vertical and 8 horizontal edges). For each macroblock, filtering starts from left-most edge to right-most for vertical edges and from upper-most to lower for horizontal edges. Horizontal edge filtering is required to be applied to vertical edge filtered blocks. Therefore every 4x4 block in macroblock is first filtered by vertical edge filtering and the filtered by horizontal edge filtering. This process is similarly implemented for luma (Y) and chroma (Cb, Cr) components.

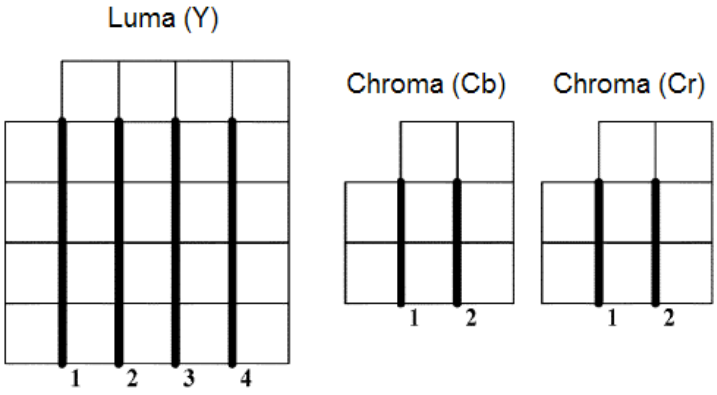

Figure 4-3. Vertical 4x4 block edges of a macroblock

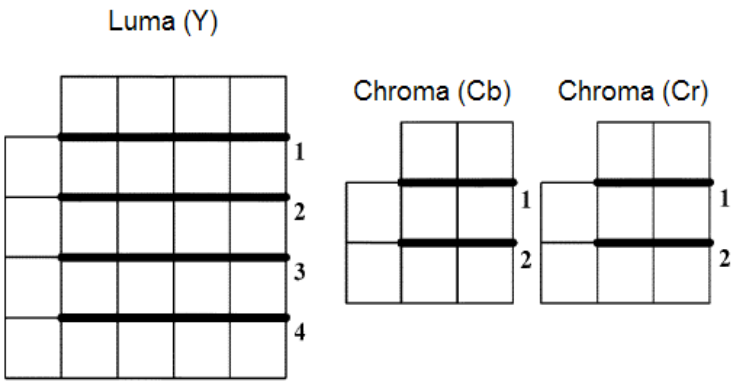

Figure 4-4. Horizontal 4x4 block edges of a macroblock

The filter is adaptive in order to discriminate artificial blocking artifacts from real edges. Therefore every 4x4 block edge is checked before filtering, to prohibit blurring of real edges. On the other hand, blocking artifacts are classified to select the appropriate type of filtering. Deblocking filtering is also adjusted by the quantization parameter (*QP*) to control the filter strength with general quality of the reconstructed frame.

4.2.1 Edge Level Adaptivity

A Boundary-strength (*bS*) parameter is set for every 4x4 block edge in a macroblock. *bS* determines the type and limit of the filtering. Conditions shown in Table 4-1 are checked from top to bottom for each 4x4 block edge to determine the corresponding *bS* value. For intra coded macroblock edges, generally used for smooth surfaces, blocking artifacts are more annoying and *bS* parameter is set to 4 to allow strong filtering. If any other condition is satisfied *bS* is set to 1 to 3, where standard filtering is applied and *bS* determines the "limits of filtering". If none of the conditions is satisfied *bS* is set to zero to disable filtering for that edge. As illustrated in Table 4-1, *bS* determination requires block types, motion vectors and transformed coefficients corresponding to each block to check the conditions. *bS* values for chroma edges are set directly from their corresponding luma block edges, without any extra check. [15]

Table 4-1. *bS* determination conditions [6]

Conditions	***bS***	**Filtering type**
One of the blocks is intra coded and the edge is at macroblock boundary	4	Strong
One of the blocks is intra coded	3	Standard
One of the blocks has coded coefficients	2	Standard
Difference of block motion ≥ 1 luma samples	1	Standard
Motion compensation from different reference frames	1	Standard
Else	0	None

4.2.2 Sample Level Adaptivity

In order to protect real edges from filtering, samples around each edge are analyzed. The analysis is done by comparison of edge sample differences with predefined threshold coefficients. These coefficients, α and β are defined by the standard after empirical tests to result visually pleasing results. [15] Coefficients are functions of *QP* to select the right threshold for given quality metric of the reconstructed frame.

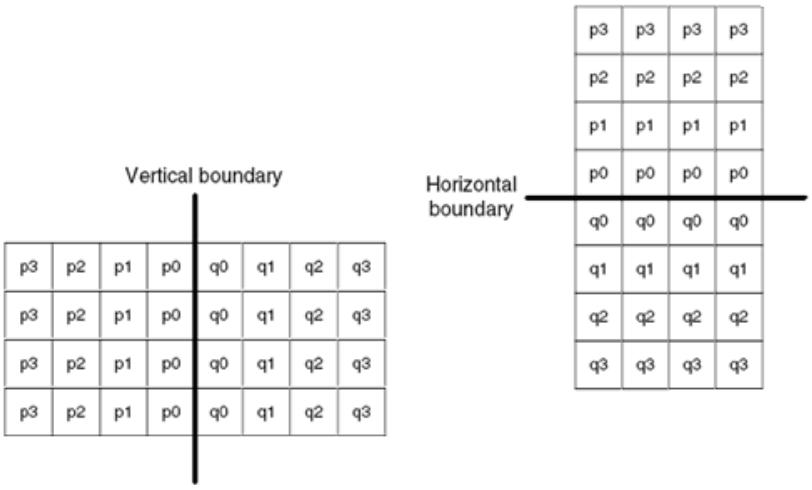

Figure 4-5. Samples around 4x4 block edge boundaries

As illustrated in Figure 4-5, edge samples are generally denoted by p_0, p_1, p_2, p_3 and q_0, q_1, q_2, q_3 . Mainly three pixel difference constraints are defined for edge filtering.

$$|p_0 - q_0| < \alpha(Index_A) \tag{4.1}$$

$$|p_1 - p_0| < \beta(Index_B) \tag{4.2}$$

$$|q_1 - q_0| < \beta(Index_B) \tag{4.3}$$

$Index_A$ and $Index_B$ are table index values for α and β and calculated as

$$Index_A = \min\left[\max(0, QP + Offset_A), 51\right] \tag{4.4}$$

$$Index_B = \min\left[\max(0, QP + Offset_B), 51\right] \tag{4.5}$$

where QP is in the range 0 to 51 and $Offset_A$ and $Offset_B$ are the offset QP values of a slice for selecting α and β [6].

α is generally greater than β, since it has a quadratic behavior with increasing QP, whereas β is truncated linear. The purpose of QP dependency is to adjust strength of filtering with quality of the frame. With increasing QP, quantization dependent distortion exponentially increases and more samples are to be filtered. For lower QP, quantization step size is lower and quantization dependent distortion is lower, therefore threshold values are defined lower to disable undesired filtering. For $Index_A < 16$ and $Index_B < 16$, α and β are set to zero and filtering is effectively disabled.

Table 4-2. Values of α and β coefficients [6]

	IndexA (for α) or IndexB (for β)												
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
α	0	0	0	0	0	0	0	0	0	0	0	0	0
β	0	0	0	0	0	0	0	0	0	0	0	0	0

	IndexA (for α) or IndexB (for β)												
	13	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
α	0	0	0	4	4	5	6	7	8	9	10	12	13
β	0	0	0	2	2	2	3	3	3	3	4	4	4

	IndexA (for α) or IndexB (for β)												
	26	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**
α	15	17	20	22	25	28	32	36	40	45	50	56	63
β	6	6	7	7	8	8	9	9	10	10	11	11	12

	IndexA (for α) or IndexB (for β)												
	39	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**
α	71	80	90	101	113	127	144	162	182	203	226	255	255
β	12	13	13	14	14	15	15	16	16	17	17	18	18

4.2.3 Slice Level Adaptivity

The offset values for the *QP* in equations 4.4 and 4.5 can be determined for every slice to adjust the amount of filtering by setting α and β coefficients. The purpose may be to discriminate areas with high detailed content from smooth surfaces. For uniform *QP* through the entire frame, offset values are set to zero and table index values become directly equal to *QP* value.

If $Offset_A = 0$ and $Offset_B = 0$

$$Index_A = \min\left[\max(0, QP), 51\right] = QP \tag{4.6}$$

$$Index_B = \min\left[\max(0, QP), 51\right] = QP \tag{4.7}$$

4.2.4 Filtering Structures

There are mainly two filtering structures used for deblocking filter. All filter structures are FIR filters and can be implemented by add and shift operations instead of multiplication and division to ease the implementation.

4.2.4.1 Standard Filtering

Standard filtering is used for block edges with boundary strength value from 1 to 3. This type of filtering modifies maximum 4 pixels near a block edge (2 near left, 2 near right), with a value that is limited by the coefficient t_{c0}, which depends on *bS* value and *QP*. Therefore, pixel modification limit effectively depends on *bS* and *QP,* as illustrated in Figure 4-6. It is named as clipping by the standard and applied to avoid too much low-pass filtering which can cause blurring. [15] The value of t_{c0} for given *QP* and *bS* determines the limiting value of Δ_0, Δ_{p1} and Δ_{q1}.

Table 4-3. t_{c0} coefficient values for given *QP* [6]

	QP																									
t_{c0}	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
***bS* = 1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
***bS* = 2**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
***bS* = 3**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1

	QP																									
t_{c0}	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**
***bS* = 1**	1	1	1	1	1	1	1	2	2	2	2	3	3	3	4	4	4	5	6	6	7	8	9	10	11	13
***bS* = 2**	1	1	1	1	1	2	2	2	2	3	3	3	4	4	5	5	6	7	8	8	10	11	12	13	15	17
***bS* = 3**	1	2	2	2	2	3	3	3	4	4	4	5	6	6	7	8	9	10	11	13	14	16	18	20	23	25

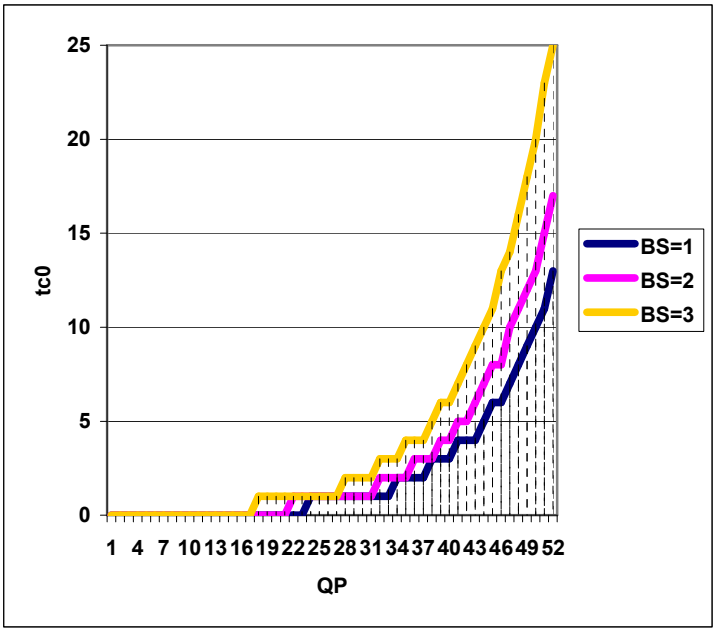

Figure 4-6. Characteristic of t_{c0} with increasing QP and for given bS

Let p_2, p_1, p_0 and q_0, q_1, q_2 be the samples near a block edge with $bS < 4$ and $bS \neq 0$. Filtered samples, p_0' and q_0' are obtained by the equations [15]:

$$p_0' = p_0 + \Delta_0 \tag{4.8}$$

$$q_0' = q_0 - \Delta_0 \tag{4.9}$$

The change factor, Δ_0, is calculated in two steps [15]:

1) The initial value of Δ_0 is calculated by 4-tap filtering of p_0, p_1, q_0 and q_1.

$$\Delta_{0i} = \left[4\left(q_0 - p_0\right) + p_1 + q_1 + 4 \right] >> 3 \tag{4.10}$$

2) Obtained value is clipped by t_{c0}'. t_{c0}' is first set equal to t_{c0} and incremented by 1 for each satisfying condition (4.2) and (4.3). For chrominance filtering, t_{c0}' is set equal to t_{c0} and directly incremented by 1.

$$\Delta_0 = \min\left[\max\left(-tc_0', \Delta_{0i}\right), tc_0'\right] \tag{4.11}$$

Filtered sample p_1' or q_1' is obtained similarly if (4.2) or (4.3) is true respectively. For chrominance filtering, p_1 and q_1 are left unfiltered.

$$If\ \left|p_1 - p_0\right| < \beta\left(Index_B\right) \Rightarrow p_1' = p_1 + \Delta_{p1} \tag{4.12}$$

$$If\ \left|q_1 - q_0\right| < \beta\left(Index_B\right) \Rightarrow q_1' = q_1 + \Delta_{q1} \tag{4.13}$$

The change factor, Δ_{p1}, is similarly calculated in two steps [15]:

1) The initial value of Δ_{p1} and Δ_{q1} are calculated by 4-tap filtering [15]:

$$\Delta_{p1i} = \left\{p_2 + \left[\left(p_0 + q_0 + 1\right) >> 1\right] - 2p_1\right\} >> 1 \tag{4.14}$$

$$\Delta_{q1i} = \left\{q_2 + \left[\left(p_0 + q_0 + 1\right) >> 1\right] - 2q_1\right\} >> 1 \tag{4.15}$$

2) Obtained values are clipped by t_{c0}.

$$\Delta_{p1} = \min\left[\max\left(-t_{c0}, \Delta_{p1i}\right), t_{c0}\right] \tag{4.16}$$

$$\Delta_{q1} = \min\left[\max\left(-t_{c0}, \Delta_{q1i}\right), t_{c0}\right] \tag{4.17}$$

4.2.4.2 Strong Filtering

Strong filtering is a special type of filtering applied to block edges with $bS = 4$. The purpose of this type of filtering is to filter "tiling effects" at smooth surfaces, near intra macroblock boundaries. [15] 4 different FIR filter structures are used conditionally. Filtering can modify maximum 6 pixels near an edge (3 at left and 3 at right). Filtering structures are applied depending on the tighter pixel difference constraint and previous (4.2) and (4.3).

$$\left|p_0 - q_0\right| < \left(\alpha\left(Index_A\right) >> 2\right) + 2 \tag{4.18}$$

If (4.18) and (4.2) is satisfied, p_0, p_1 and p_2 luma samples are filtered by 5-tap and 4-tap FIR filters [15]:

$$p_0' = \left(p_2 + 2p_1 + 2p_0 + 2q_0 + q_1 + 4\right) >> 3 \tag{4.19}$$

$$p_1' = \left(p_2 + p_1 + p_0 + q_0 + 2\right) >> 2 \tag{4.20}$$

$$p_2' = \left(2p_3 + 3p_2 + p_1 + p_0 + q_0 + 4\right) >> 3 \tag{4.21}$$

Else if (4.18) or (4.2) is not satisfied, or chroma filtering is implemented, p_0 is filtered by a weaker 3-tap FIR filter [15]:

$$p_0' = \left(2p_1 + p_0 + q_1 + 2\right) >> 2 \tag{4.22}$$

Similarly if (4.18) and (4.3) are satisfied, q_0, q_1 and q_2 luma samples are filtered:

$$q_0' = \left(q_2 + 2q_1 + 2p_0 + 2q_0 + p_1 + 4\right) >> 3 \tag{4.23}$$

$$q_1' = \left(q_2 + q_1 + p_0 + q_0 + 2\right) >> 2 \tag{4.24}$$

$$q_2' = \left(2q_3 + 3q_2 + q_1 + q_0 + p_0 + 4\right) >> 3 \tag{4.25}$$

Else if (4.18) or (4.3) is not satisfied, or chroma filtering is implemented, q_0 is filtered similarly.

$$q_0' = (2q_1 + q_0 + p_1 + 2) >> 2 \tag{4.26}$$

In Figure 4-7, the flow chart of filtering decision for each edge is illustrated.

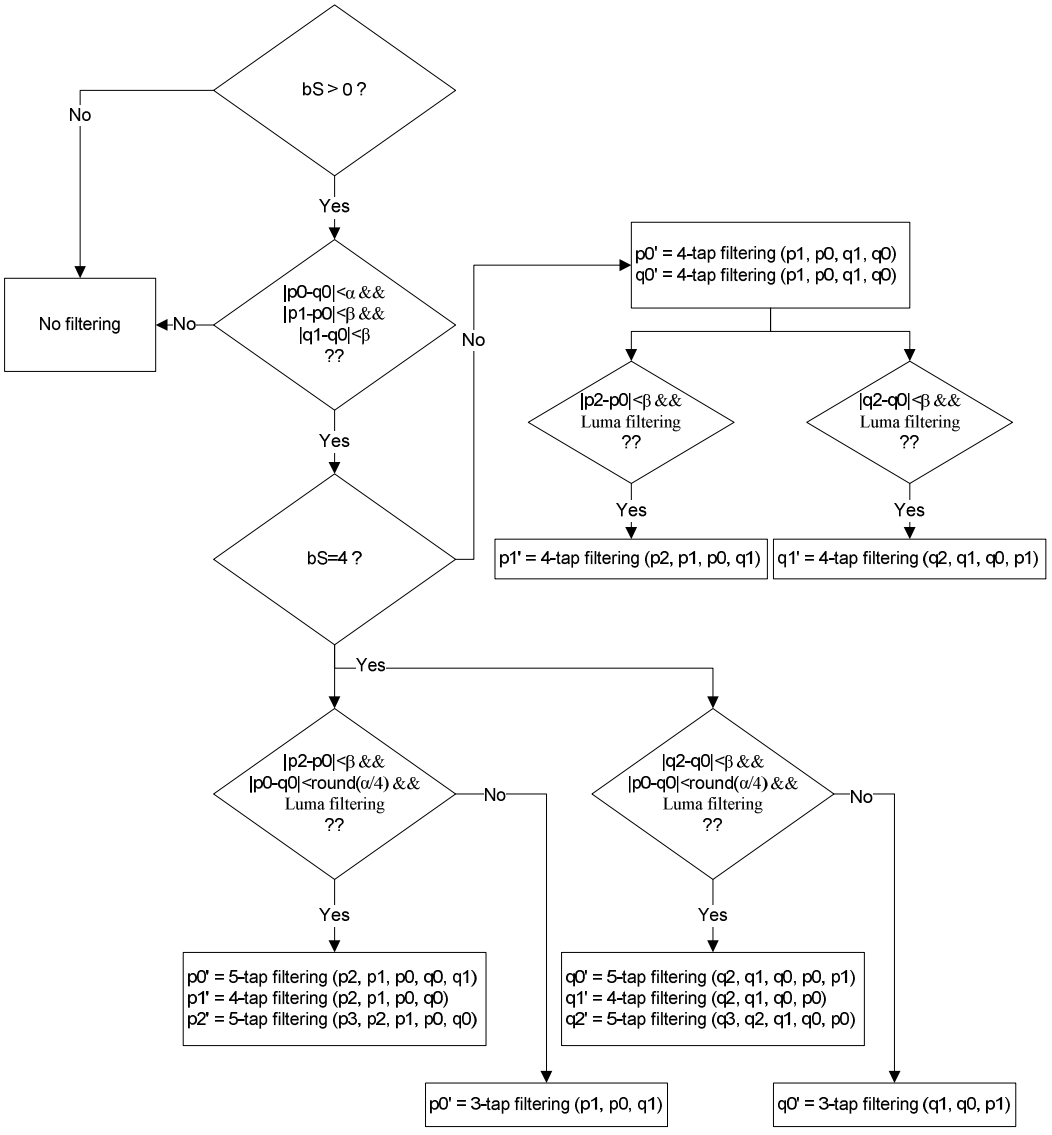

Figure 4-7. Flow chart of filtering decision for each edge

4.3 Deblocking Filter Architectures in Literature

Deblocking filtering is being interested after standardized in H.264/AVC as an in-loop filter. Computational load of deblocking filtering, which is approximately one-third of total decoder complexity [25], made researchers focus on efficient algorithms to implement deblocking filtering.

On the other hand, improvements for video compression efficiency became saturated and solutions for H.264 decoder hardware began to be interested. Deblocking filter is naturally a part of the decoder hardware to be used for various consumer video coding applications. In order to obtain a design with high performance to support high resolution, with less power consumption or less logic utilization, efficient hardware architectures started to be developed since 2005.

B. Sheng *et. al.* [16] proposed a deblocking filter architecture which has a high performance but its internal memory requirement is extremely high. Also the gate count of the design is considerably high for a VLSI design.

B. J. Kim *et al.* [23] proposed a low power architecture for H.264 deblocking filter. They used a frame memory to read input pixels and registers to buffer filtered pixel blocks. They disabled filtering for $QP<16$ and threshold conditions not satisfied. The architecture was simple and effective in terms of gate count and power consumption but throughput was not satisfactory for moderate resolutions even with a high frequency processing clock.

M. Sima *et. al.* [22] proposed a deblocking filter architecture with instruction and parameter interface and 64-byte internal memory which can be embedded into general purpose processor and DSP as a hardware accelerator software codec.

G. Khurana *et. al.* [13] proposed a pipelined hardware architecture which obtained brilliant throughput to be used for high resolution video. The architecture is seemed to be effective to reduce internal memory usage and gate count.

K. Xu *et. al.* [35] proposed a 5-stage pipelined architecture with single-port SRAM use as an internal memory and high processing speed to allow high resolutions.

C. C. Cheng *et. al.* [21] proposed an improved version of their previous architecture [18] for which data reuse is effective and only 64 byte of internal memory is required with degraded processing speed.

T. M. Liu *et. al.* [20] proposed an architecture which is compatible with MPEG-4 and H.264 decoders. The architecture has capability to work in-loop and post-loop. They used a hybrid processing order and tried to reduce memory accesses.

M. Parlak *et. al.* [25] proposed a different processing order compatible with the transmission order of inverse transform & quantization unit to increase processing performance. Two different architectures are proposed for high performance and low power applications. Obtained architectures have low gate count, but have large amount of memory access and insufficient processing speed.

4.3.1 Data Reuse Analysis of Processing Orders in Literature

Processing order selection for deblocking filtering is effective on the architecture design and data reuse performance. Several processing orders are used in proposed architectures mentioned above. These processing orders are analyzed in terms of data reuse to determine their performance for an architecture with a register set for buffering intermediate filtering results instead of "power and chip area consuming" internal memory.

In the analysis, data access types and number of registers required to support "memory-free design" are determined. In tables, "*direct*" represents data transfer of a previously filtered block to next filtering stage. "*reg1, reg2 ...*" represent load and store accesses from the register set. "*ext*" represents an access to external memory. External memory accesses are restricted to be allowed only for reading new blocks to be filtered or writing blocks with all edges filtered. The purpose is to minimize external memory access and use only register set for intermediate results buffering to obtain a low power implementation. Consequently, the processing order with minimum external memory access and register requirement is to be determined for implementation.

4.3.1.1 Basic Processing Order

Basic processing order is the simplest method to implement the deblocking filter, where all vertical edges are filtered first in a sequential order and horizontal edges are filtered next. Processing order of edges do not take data reuse into account but the controller architecture is simple and do not require complex hardware. Basic processing order, as shown in Figure 4-8, is specified in H.264 standard [6] and used by [17, 22], which the architecture is a base point for further improvements in terms of processing speed and memory use.

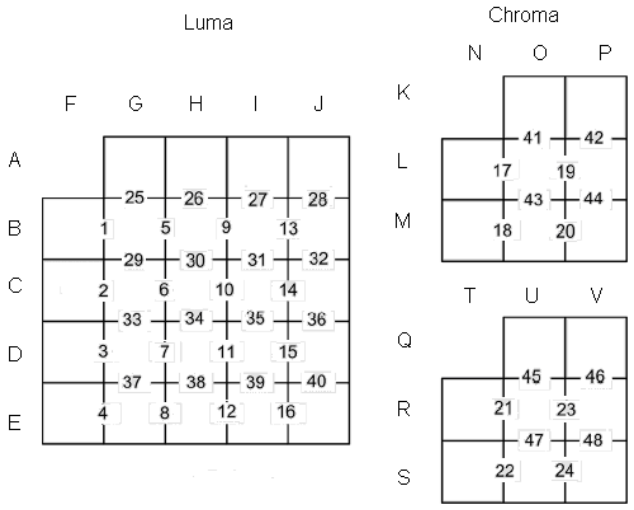

Figure 4-8. Basic processing order

As illustrated in Table 4-4, for this processing order:

# of read access from external memory = 24 luma + 16 chroma = 40

# of write access to external memory = 24 luma + 16 chroma = 40

# of direct read access from previous filtering result = 0

# of registers required = 24 (or 24x128bit internal memory)

# of read and write access to registers = 56 + 56 = 112

If chroma filtering was performed after luma filtering is totally finished, 16 registers would be adequate, but still the architecture would need a large internal memory for buffering instead of large number of registers.

Table 4-4. Data access types of 4x4 blocks with basic processing order

Edge #	Block1	Block2	load1	load2	store1	store2
1	BF	BG	ext	ext	ext	reg1
2	CF	CG	ext	ext	ext	reg2
3	DF	DG	ext	ext	ext	reg3
4	EF	EG	ext	ext	ext	reg4
5	BG	BH	reg1	ext	reg1	reg5
6	CG	CH	reg2	ext	reg2	reg6
7	DG	DH	reg3	ext	reg3	reg7
8	EG	EH	reg4	ext	reg4	reg8
9	BH	BI	reg5	ext	reg5	reg9
10	CH	CI	reg6	ext	reg6	reg10
11	DH	DI	reg7	ext	reg7	reg11
12	EH	EI	reg8	ext	reg8	reg12
13	BI	BJ	reg9	ext	reg9	reg13
14	CI	CJ	reg10	ext	reg10	reg14
15	DI	DJ	reg11	ext	reg11	reg15
16	EI	EJ	reg12	ext	reg12	reg16
17	LN	LO	ext	ext	ext	reg17
18	MN	MO	ext	ext	ext	reg18
19	LO	LP	reg17	ext	reg17	reg19
20	MO	MP	reg18	ext	reg18	reg20
21	RT	RU	ext	ext	ext	reg21
22	ST	SU	ext	ext	ext	reg22
23	RU	RV	reg21	ext	reg21	reg23
24	SU	SV	reg22	ext	reg22	reg24
25	AG	BG	ext	reg1	ext	reg1
26	AH	BH	ext	reg5	ext	reg5
27	AI	BI	ext	reg9	ext	reg9
28	AJ	BJ	ext	reg13	ext	reg13
29	BG	CG	reg1	reg2	ext	reg2
30	BH	CH	reg5	reg6	ext	reg6
31	BI	CI	reg9	reg10	ext	reg10
32	BJ	CJ	reg13	reg14	ext	reg14
33	CG	DG	reg2	reg3	ext	reg3
34	CH	DH	reg6	reg7	ext	reg7
35	CI	DI	reg10	reg11	ext	reg11
36	CJ	DJ	reg14	reg15	ext	reg15
37	DG	EG	reg3	reg4	ext	ext
38	DH	EH	reg7	reg8	ext	ext
39	DI	EI	reg11	reg12	ext	ext
40	DJ	EJ	reg15	reg16	ext	ext
41	KO	LO	ext	reg17	ext	reg17
42	KP	LP	ext	reg19	ext	reg19
43	LO	MO	reg17	reg18	ext	ext
44	LP	MP	reg19	reg20	ext	ext
45	QU	RU	ext	reg21	ext	reg21
46	QV	RV	ext	reg23	ext	reg23
47	RU	SU	reg21	reg22	ext	ext
48	RV	SV	reg23	reg24	ext	ext

4.3.1.2 1-D Processing Order Proposed by [18]

1-D processing order takes more care on date reuse than the previous one so that each edge being filtered is transferred to next filtering stage. This approach is advantageous so that some of intermediate filtering results can be used instantaneously without storing and reloading. The method is called 1-D and used by [18, 23], since data reuse is only in horizontal direction for vertical edge filtering and vertical direction for horizontal edge filtering.

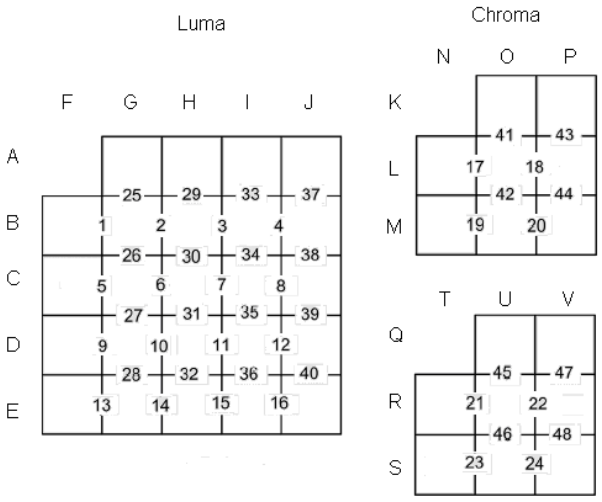

Figure 4-9. 1-D processing order proposed by [18]

As illustrated in Table 4-5, for this processing order:

# of read access from external memory = 24 luma + 16 chroma = 40

# of write access to external memory = 24 luma + 16 chroma = 40

# of direct read access from previous filtering result = 24

# of registers required = 24 (or 24x128bit internal memory)

# of read and write access to registers = 32 + 32 = 64

Although there is data reuse with direct transfer of some of previously filtered blocks, architecture is still inappropriate for a memory-free design because of large number of registers. Like the previous case, 16 registers would be enough if chroma filtering is implemented after luma filtering is completely finished.

Table 4-5. Data access types for 1-D processing order proposed by [18]

Edge #	Block1	Block2	load1	load2	store1	store2
1	BF	BG	ext	ext	ext	direct
2	BG	BH	direct	ext	reg1	direct
3	BH	BI	direct	ext	reg2	direct
4	BI	BJ	direct	ext	reg3	reg4
5	CF	CG	ext	ext	ext	direct
6	CG	CH	direct	ext	reg5	direct
7	CH	CI	direct	ext	reg6	direct
8	CI	CJ	direct	ext	reg7	reg8
9	DF	DG	ext	ext	ext	direct
10	DG	DH	direct	ext	reg9	direct
11	DH	DI	direct	ext	reg10	direct
12	DI	DJ	direct	ext	reg11	reg12
13	EF	EG	ext	ext	ext	direct
14	EG	EH	direct	ext	reg13	direct
15	EH	EI	direct	ext	reg14	direct
16	EI	EJ	direct	ext	reg15	reg16
17	LN	LO	ext	ext	ext	direct
18	LO	LP	direct	ext	reg17	reg18
19	MN	MO	ext	ext	ext	direct
20	MO	MP	direct	ext	reg19	reg20
21	RT	RU	ext	ext	ext	direct
22	RU	RV	direct	ext	reg21	reg22
23	ST	SU	ext	ext	ext	direct
24	SU	SV	direct	ext	reg23	reg24
25	AG	BG	ext	reg1	ext	direct
26	BG	CG	direct	reg5	ext	direct
27	CG	DG	direct	reg9	ext	direct
28	DG	EG	direct	reg13	ext	ext
29	AH	BH	ext	reg2	ext	direct
30	BH	CH	direct	reg6	ext	direct
31	CH	DH	direct	reg10	ext	direct
32	DH	EH	direct	reg14	ext	ext
33	AI	BI	ext	reg3	ext	direct
34	BI	CI	direct	reg7	ext	direct
35	CI	DI	direct	reg11	ext	direct
36	DI	EI	direct	reg15	ext	ext
37	AJ	BJ	ext	reg4	ext	direct
38	BJ	CJ	direct	reg8	ext	direct
39	CJ	DJ	direct	reg12	ext	direct
40	DJ	EJ	direct	reg16	ext	ext
41	KO	LO	ext	reg17	ext	direct
42	LO	MO	direct	reg19	ext	ext
43	KP	LP	ext	reg18	ext	direct
44	LP	MP	direct	reg20	ext	ext
45	QU	RU	ext	reg21	ext	direct
46	RU	SU	direct	reg23	ext	ext
47	QV	RV	ext	reg22	ext	direct
48	RV	SV	direct	reg24	ext	ext

4.3.1.3 1-D Processing Order Proposed by [13]

This processing order is proposed by [13], to be used for a pipelined architecture which outperforms the previous ones in terms of data reuse and allows pipelined processing since horizontal and vertical edge filtering can be implemented simultaneously in different stages.

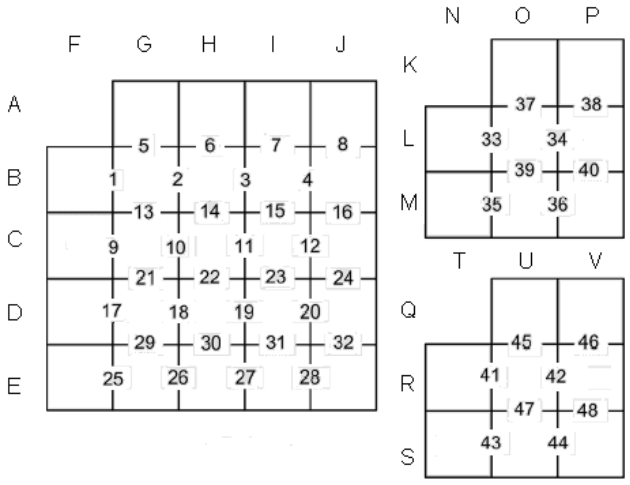

Figure 4-10. 1-D processing order proposed by [13]

As illustrated in Table 4-6, for this processing order:

# of read access from external memory = 24 luma + 16 chroma = 40

# of write access to external memory = 24 luma + 16 chroma = 40

# of direct read access from previous filtering result = 16

# of registers required = 8 (or 8x128bit internal memory)

# of read and write access to registers = 40 + 40 = 80

For this processing order number of registers required is much more less than previous ones. 8 128-bit registers, each holding a 4x4 block with 8-bit pixel depth, can be used as a buffer for intermediate filtering results instead of a 8x128 (128byte) memory.

Table 4-6. Data access types of 1-D processing order proposed by [13]

Edge #	Block1	Block2	load1	load2	store1	store2
1	BF	BG	ext	ext	ext	direct
2	BG	BH	direct	ext	reg1	direct
3	BH	BI	direct	ext	reg2	direct
4	BI	BJ	direct	ext	reg3	reg4
5	AG	BG	ext	reg1	ext	reg1
6	AH	BH	ext	reg2	ext	reg2
7	AI	BI	ext	reg3	ext	reg3
8	AJ	BJ	ext	reg4	ext	reg4
9	CF	CG	ext	ext	ext	direct
10	CG	CH	direct	ext	reg5	direct
11	CH	CI	direct	ext	reg6	direct
12	CI	CJ	direct	ext	reg7	reg8
13	BG	CG	reg1	reg5	ext	reg5
14	BH	CH	reg2	reg6	ext	reg6
15	BI	CI	reg3	reg7	ext	reg7
16	BJ	CJ	reg4	reg8	ext	reg8
17	DF	DG	ext	ext	ext	direct
18	DG	DH	direct	ext	reg1	direct
19	DH	DI	direct	ext	reg2	direct
20	DI	DJ	direct	ext	reg3	reg4
21	CG	DG	reg5	reg1	ext	reg1
22	CH	DH	reg6	reg2	ext	reg2
23	CI	DI	reg7	reg3	ext	reg3
24	CJ	DJ	reg8	reg4	ext	reg4
25	EF	EG	ext	ext	ext	direct
26	EG	EH	direct	ext	reg5	direct
27	EH	EI	direct	ext	reg6	direct
28	EI	EJ	direct	ext	reg7	reg8
29	DG	EG	reg1	reg5	ext	ext
30	DH	EH	reg2	reg6	ext	ext
31	DI	EI	reg3	reg7	ext	ext
32	DJ	EJ	reg4	reg8	ext	ext
33	LN	LO	ext	ext	ext	direct
34	LO	LP	direct	ext	reg1	reg2
35	MN	MO	ext	ext	ext	direct
36	MO	MP	direct	ext	reg3	reg4
37	KO	LO	ext	reg1	ext	reg1
38	KP	LP	ext	reg2	ext	reg2
39	LO	MO	reg1	reg3	ext	ext
40	LP	MP	reg2	reg4	ext	ext
41	RT	RU	ext	ext	ext	direct
42	RU	RV	direct	ext	reg1	reg2
43	ST	SU	ext	ext	ext	direct
44	SU	SV	direct	ext	reg3	reg4
45	QU	RU	ext	reg1	ext	reg1
46	QV	RV	ext	reg2	ext	reg2
47	RU	SU	reg1	reg3	ext	ext
48	RV	SV	reg2	reg4	ext	ext

4.3.1.4 2-D Processing Order Proposed by [20]

2-D processing orders outperform 1-D, in terms of data reuse. As the name implies, data reuse is in both dimensions and therefore less memory is required for intermediate buffering.

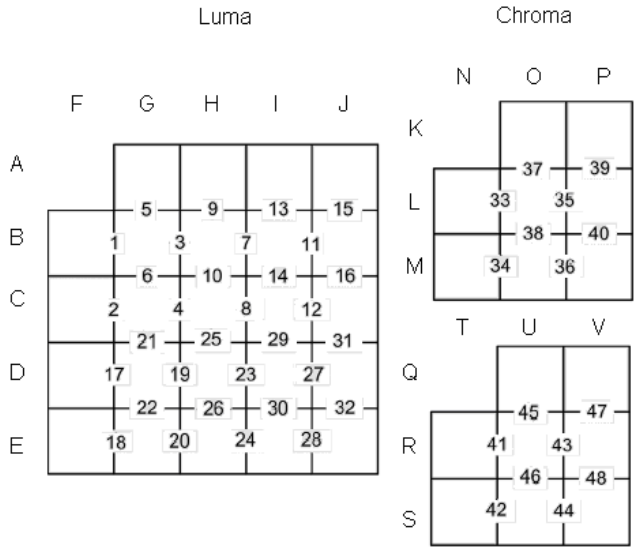

Figure 4-11. 2-D processing order proposed by [20]

As illustrated in Table 4-7, for this processing order:

# of read access from external memory = 24 luma + 16 chroma = 40
# of write access to external memory = 24 luma + 16 chroma = 40
# of direct read access from previous filtering result = 12
# of registers required = 8 (or 8x128bit internal memory)
# of read and write access to registers = 44 + 44 = 88

Table 4-7. Data access types for 2-D processing order proposed by [20]

Edge #	Block1	Block2	load1	load2	store1	store2
1	BF	BG	ext	ext	ext	reg1
2	CF	CG	ext	ext	ext	reg2
3	BG	BH	reg1	ext	reg1	reg3
4	CG	CH	reg2	ext	reg2	reg4
5	AG	BG	ext	reg1	ext	direct
6	BG	CG	direct	reg2	ext	reg2
7	BH	BI	reg3	ext	reg3	reg1
8	CH	CI	reg4	ext	reg4	reg5
9	AH	BH	ext	reg3	ext	direct
10	BH	CH	direct	reg4	ext	reg4
11	BI	BJ	reg1	ext	reg1	reg3
12	CI	CJ	reg5	ext	reg5	reg6
13	AI	BI	ext	reg1	ext	direct
14	BI	CI	direct	reg5	ext	reg5
15	AJ	BJ	ext	reg3	ext	direct
16	BJ	CJ	direct	reg6	ext	reg6
17	DF	DG	ext	ext	ext	reg1
18	EF	EG	ext	ext	ext	reg3
19	DG	DH	reg1	ext	reg1	reg7
20	EG	EH	reg3	ext	reg3	reg8
21	CG	DG	reg2	reg1	ext	direct
22	DG	EG	direct	reg3	ext	ext
23	DH	DI	reg7	ext	reg7	reg1
24	EH	EI	reg8	ext	reg8	reg2
25	CH	DH	reg4	reg7	ext	direct
26	DH	EH	direct	reg8	ext	ext
27	DI	DJ	reg1	ext	reg1	reg3
28	EI	EJ	reg2	ext	reg2	reg4
29	CI	DI	reg5	reg1	ext	direct
30	DI	EI	direct	reg2	ext	ext
31	CJ	DJ	reg6	reg3	ext	direct
32	DJ	EJ	direct	reg4	ext	ext
33	LN	LO	ext	ext	ext	reg1
34	MN	MO	ext	ext	ext	reg2
35	LO	LP	reg1	ext	reg1	reg3
36	MO	MP	reg2	ext	reg2	reg4
37	KO	LO	ext	reg1	ext	direct
38	LO	MO	direct	reg2	ext	ext
39	KP	LP	ext	reg3	ext	direct
40	LP	MP	direct	reg4	ext	ext
41	RT	RU	ext	ext	ext	reg1
42	ST	SU	ext	ext	ext	reg2
43	RU	RV	reg1	ext	reg1	reg3
44	SU	SV	reg2	ext	reg2	reg4
45	QU	RU	ext	reg1	ext	direct
46	RU	SU	direct	reg2	ext	ext
47	QV	RV	ext	reg3	ext	direct
48	RV	SV	direct	reg4	ext	ext

4.3.1.5 2-D Processing Order Proposed by [35]

This is another version of 2-D processing order proposed by [35].

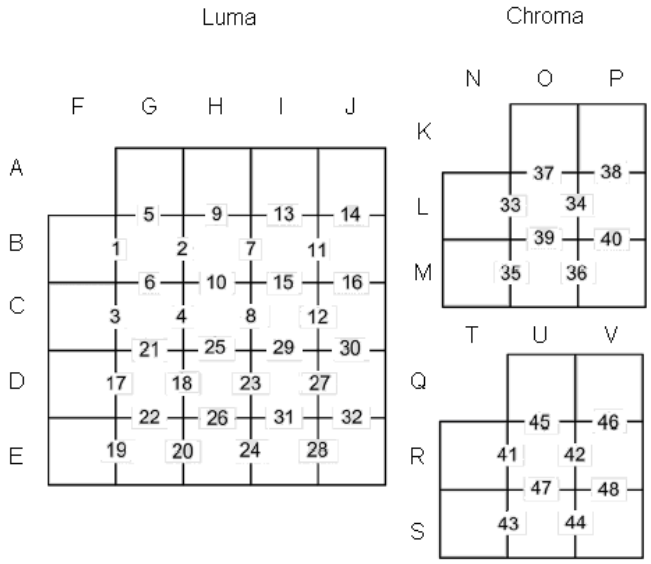

Figure 4-12. 2-D processing order proposed by [35]

As illustrated in Table 4-8, for this processing order:

# of read access from external memory = 24 luma + 16 chroma = 40
# of write access to external memory = 24 luma + 16 chroma = 40
# of direct read access from previous filtering result = 12
# of registers required = 8 (or 8x128bit internal memory)
# of read and write access to registers = 44 + 44 = 88

Although data reuse is well done, number of required registers is still high.

Table 4-8. Data access types for 2-D processing order proposed by [35]

Edge #	Block1	Block2	load1	load2	store1	store2
1	BF	BG	ext	ext	ext	direct
2	BG	BH	direct	ext	reg1	reg2
3	CF	CG	ext	ext	ext	direct
4	CG	CH	direct	ext	reg3	reg4
5	AG	BG	ext	reg1	ext	direct
6	BG	CG	direct	reg3	ext	reg3
7	BH	BI	reg2	ext	reg2	reg1
8	CH	CI	reg4	ext	reg4	reg5
9	AH	BH	ext	reg2	ext	direct
10	BH	CH	direct	reg4	ext	reg4
11	BI	BJ	reg1	ext	reg1	reg2
12	CI	CJ	reg5	ext	reg5	reg6
13	AI	BI	ext	reg1	ext	reg1
14	AJ	BJ	ext	reg2	ext	reg2
15	BI	CI	reg1	reg5	ext	reg5
16	BJ	CJ	reg2	reg6	ext	reg6
17	DF	DG	ext	ext	ext	direct
18	DG	DH	direct	ext	reg1	reg2
19	EF	EG	ext	ext	ext	direct
20	EG	EH	direct	ext	reg7	reg8
21	CG	DG	reg3	reg1	ext	direct
22	DG	EG	direct	reg7	ext	ext
23	DH	DI	reg2	ext	reg2	reg1
24	EH	EI	reg8	ext	reg8	reg3
25	CH	DH	reg4	reg2	ext	direct
26	DH	EH	direct	reg8	ext	ext
27	DI	DJ	reg1	ext	reg1	reg2
28	EI	EJ	reg3	ext	reg3	reg4
29	CI	DI	reg5	reg1	ext	reg1
30	CJ	DJ	reg6	reg2	ext	reg2
31	DI	EI	reg1	reg3	ext	ext
32	DJ	EJ	reg2	reg4	ext	ext
33	LN	LO	ext	ext	ext	direct
34	LO	LP	direct	ext	reg1	reg2
35	MN	MO	ext	ext	ext	direct
36	MO	MP	direct	ext	reg3	reg4
37	KO	LO	ext	reg1	ext	reg1
38	KP	LP	ext	reg2	ext	reg2
39	LO	MO	reg1	reg3	ext	ext
40	LP	MP	reg2	reg4	ext	ext
41	RT	RU	ext	ext	ext	direct
42	RU	RV	direct	ext	reg1	reg2
43	ST	SU	ext	ext	ext	direct
44	SU	SV	direct	ext	reg3	reg4
45	QU	RU	ext	reg1	ext	reg1
46	QV	RV	ext	reg2	ext	reg2
47	RU	SU	reg1	reg3	ext	ext
48	RV	SV	reg2	reg4	ext	ext

4.3.1.6 2-D Processing Order Proposed By [34]

It is proposed by [34] and also used in [25], to be compatible with transform and quantization transmission order to increase processing speed.

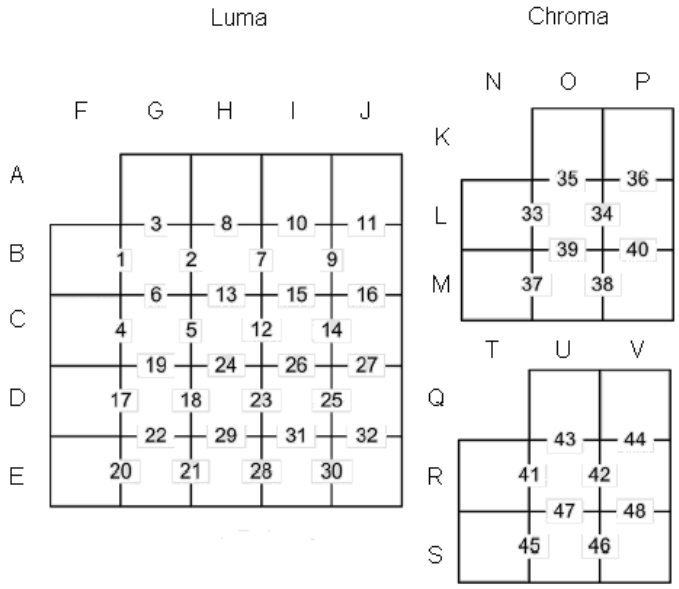

Figure 4-13. 2-D Processing order proposed by [34]

As illustrated in Table 4-9, for this processing order:

# of read access from external memory = 24 luma + 16 chroma = 40
# of write access to external memory = 24 luma + 16 chroma = 40
# of direct read access from previous filtering result = 24
# of registers required = 6 (or 6x128bit internal memory)
# of read and write access to registers = 32 + 32 = 64

Number of registers required is only 6 for this proposed processing order but required controlling hardware seems to be complex. The types of data access for each edge seems not be symmetric with others which may result a complicated state machine.

Table 4-9. Data access types of 4x4 blocks with processing order proposed by [34]

Edge #	Block1	Block2	load1	load2	store1	store2
1	BF	BG	ext	ext	ext	direct
2	BG	BH	direct	ext	direct	reg1
3	AG	BG	ext	direct	ext	reg2
4	CF	CG	ext	ext	ext	direct
5	CG	CH	direct	ext	direct	reg3
6	BG	CG	reg2	direct	ext	reg2
7	BH	BI	reg1	ext	direct	reg1
8	AH	BH	ext	direct	ext	reg4
9	BI	BJ	reg1	ext	direct	reg1
10	AI	BI	ext	direct	ext	reg5
11	AJ	BJ	ext	reg1	ext	reg1
12	CH	CI	reg3	ext	direct	reg3
13	BH	CH	reg4	direct	ext	reg4
14	CI	CJ	reg3	ext	direct	reg3
15	BI	CI	reg5	direct	ext	reg5
16	BJ	CJ	reg1	reg3	ext	reg1
17	DF	DG	ext	ext	ext	direct
18	DG	DH	direct	ext	direct	reg3
19	CG	DG	reg2	direct	ext	reg2
20	EF	EG	ext	ext	ext	direct
21	EG	EH	direct	ext	direct	reg6
22	DG	EG	reg2	direct	ext	ext
23	DH	DI	reg3	ext	direct	reg2
24	CH	DH	reg4	direct	ext	reg3
25	DI	DJ	reg2	ext	direct	reg4
26	CI	DI	reg5	direct	ext	reg5
27	CJ	DJ	reg1	reg4	ext	reg4
28	EH	EI	reg6	ext	direct	reg2
29	DH	EH	reg3	direct	ext	ext
30	EI	EJ	reg2	ext	direct	reg2
31	DI	EI	reg5	direct	ext	ext
32	DJ	EJ	reg4	reg2	ext	ext
33	LN	LO	ext	ext	ext	direct
34	LO	LP	direct	ext	direct	reg1
35	KO	LO	ext	direct	ext	reg2
36	KP	LP	ext	reg1	ext	reg1
37	MN	MO	ext	ext	ext	direct
38	MO	MP	direct	ext	direct	reg3
39	LO	MO	reg2	direct	ext	ext
40	LP	MP	reg1	reg3	ext	ext
41	RT	RU	ext	ext	ext	direct
42	RU	RV	direct	ext	direct	reg1
43	QU	RU	ext	direct	ext	reg2
44	QV	RV	ext	reg1	ext	reg1
45	ST	SU	ext	ext	ext	direct
46	SU	SV	direct	ext	direct	reg3
47	RU	SU	reg2	direct	ext	ext
48	RV	SV	reg1	reg3	ext	ext

4.3.1.7 2-D Processing Order Proposed by [19]

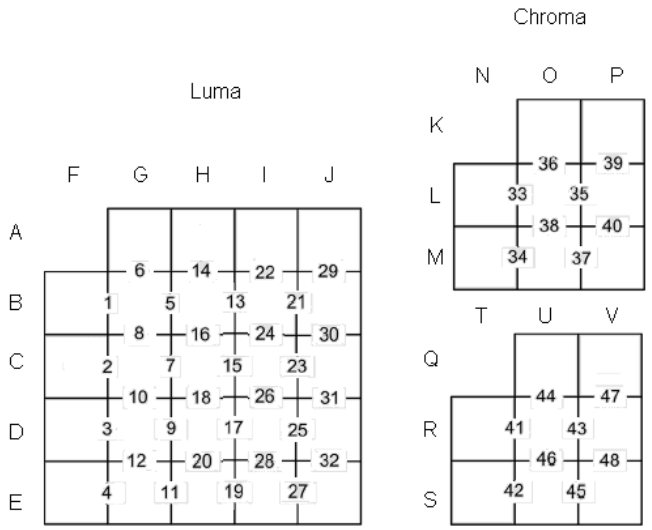

Figure 4-14. 2-D processing order proposed by [19]

As illustrated in Table 4-10, for this processing order:

# of read access from external memory = 24 luma + 16 chroma = 40
# of write access to external memory = 24 luma + 16 chroma = 40
# of direct read access from previous filtering result = 21
# of registers required = 5 (or 8x128bit internal memory)
# of read and write access to registers = 35 + 35 = 70

Only 5 registers are required for this processing order and direct transfers are higher compared to previous ones.

Table 4-10. Data access types for 2-D processing order proposed by [19]

Edge #	Block1	Block2	load1	load2	store1	store2
1	BF	BG	ext	ext	ext	reg1
2	CF	CG	ext	ext	ext	reg2
3	DF	DG	ext	ext	ext	reg3
4	EF	EG	ext	ext	ext	reg4
5	BG	BH	reg1	ext	direct	reg1
6	AG	BG	ext	direct	ext	reg5
7	CG	CH	reg2	ext	direct	reg2
8	BG	CG	reg5	direct	ext	reg5
9	DG	DH	reg3	ext	direct	reg3
10	CG	DG	reg5	direct	ext	reg5
11	EG	EH	reg4	ext	direct	reg4
12	DG	EG	reg5	direct	ext	ext
13	BH	BI	reg1	ext	direct	reg5
14	AH	BH	ext	direct	ext	reg1
15	CH	CI	reg2	ext	direct	reg2
16	BH	CH	reg1	direct	ext	reg1
17	DH	DI	reg3	ext	direct	reg3
18	CH	DH	reg1	direct	ext	reg1
19	EH	EI	reg4	ext	direct	reg4
20	DH	EH	reg1	direct	ext	ext
21	BI	BJ	reg5	ext	direct	reg5
22	AI	BI	ext	direct	ext	reg1
23	CI	CJ	reg2	ext	direct	reg2
24	BI	CI	reg1	direct	ext	reg1
25	DI	DJ	reg3	ext	direct	reg3
26	CI	DI	reg1	direct	ext	reg1
27	EI	EJ	reg4	ext	direct	reg4
28	DI	EI	reg1	direct	ext	ext
29	AJ	BJ	ext	reg5	ext	direct
30	BJ	CJ	direct	reg2	ext	direct
31	CJ	DJ	direct	reg3	ext	direct
32	DJ	EJ	direct	reg4	ext	ext
33	LN	LO	ext	ext	ext	reg1
34	MN	MO	ext	ext	ext	reg2
35	LO	LP	reg1	ext	direct	reg1
36	KO	LO	ext	direct	ext	reg3
37	MO	MP	reg2	ext	direct	reg2
38	LO	MO	reg3	direct	ext	ext
39	KP	LP	ext	reg1	ext	direct
40	LP	MP	direct	reg2	ext	ext
41	RT	RU	ext	ext	ext	reg1
42	ST	SU	ext	ext	ext	reg2
43	RU	RV	reg1	ext	direct	reg1
44	QU	RU	ext	direct	ext	reg3
45	SU	SV	reg2	ext	direct	reg2
46	RU	SU	reg3	direct	ext	ext
47	QV	RV	ext	reg1	ext	direct
48	RV	SV	direct	reg2	ext	ext

4.3.1.8 2-D Processing Order Proposed by [16]

2-D processing order proposed by [16] and also used by [21, 36] is the best in terms of data reuse. A block filtered is next used twice before storing to external memory so that internal memory requirement is minimized. Processing order of edges is illustrated in Figure 4-15.

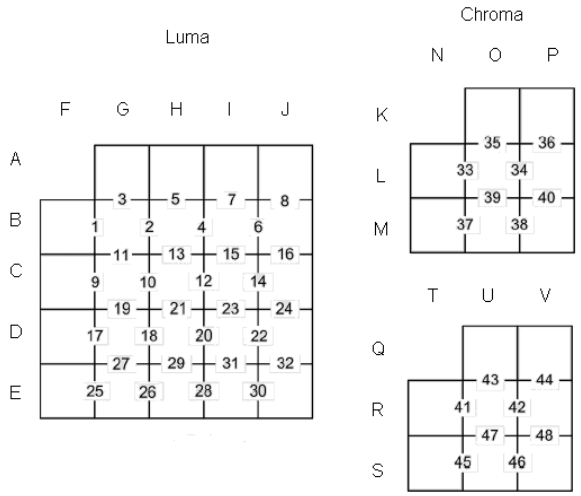

Figure 4-15. 2-D processing order proposed by [16]

As illustrated in Table 4-11, for this processing order:

# of read access from external memory = 24 luma + 16 chroma = 40

# of write access to external memory = 24 luma + 16 chroma = 40

# of direct read access from previous filtering result = 24

# of registers required = 5 (or 5x128bit internal memory)

# of read and write access to registers = 32 + 32 = 64

This processing order again requires 5 registers for memory-free design. The controller hardware seems to be obviously complex than basic processing order case but there is symmetry between data access decisions for most of the edges compared to the one proposed by [19]. So a state-machine that handles the data access case for each edge can be designed without complicated hardware. The design is memory-free and requires only 5x128-bit registers replacing a minimum of 80-bytes internal memory and a lot of memory access.

Table 4-11. Data access types for 2-D processing order proposed by [16]

Edge #	Block1	Block2	load1	load2	store1	store2
1	BF	BG	ext	ext	ext	direct
2	BG	BH	direct	ext	direct	reg1
3	AG	BG	ext	direct	ext	reg2
4	BH	BI	reg1	ext	direct	reg1
5	AH	BH	ext	direct	ext	reg3
6	BI	BJ	reg1	ext	direct	reg1
7	AI	BI	ext	direct	ext	reg4
8	AJ	BJ	ext	reg1	ext	reg5
9	CF	CG	ext	ext	ext	direct
10	CG	CH	direct	ext	direct	reg1
11	BG	CG	reg2	direct	ext	reg2
12	CH	CI	reg1	ext	direct	reg1
13	BH	CH	reg3	direct	ext	reg3
14	CI	CJ	reg1	ext	direct	reg1
15	BI	CI	reg4	direct	ext	reg4
16	BJ	CJ	reg5	reg1	ext	reg5
17	DF	DG	ext	ext	ext	direct
18	DG	DH	direct	ext	direct	reg1
19	CG	DG	reg2	direct	ext	reg2
20	DH	DI	reg1	ext	direct	reg1
21	CH	DH	reg3	direct	ext	reg3
22	DI	DJ	reg1	ext	direct	reg1
23	CI	DI	reg4	direct	ext	reg4
24	CJ	DJ	reg5	reg1	ext	reg5
25	EF	EG	ext	ext	ext	direct
26	EG	EH	direct	ext	direct	reg1
27	DG	EG	reg2	direct	ext	ext
28	EH	EI	reg1	ext	direct	reg1
29	DH	EH	reg3	direct	ext	ext
30	EI	EJ	reg1	ext	direct	reg1
31	DI	EI	reg4	direct	ext	ext
32	DJ	EJ	reg5	reg1	ext	ext
33	LN	LO	ext	ext	ext	direct
34	LO	LP	direct	ext	direct	reg1
35	KO	LO	ext	direct	ext	reg2
36	KP	LP	ext	reg1	ext	reg1
37	MN	MO	ext	ext	ext	direct
38	MO	MP	direct	ext	direct	reg3
39	LO	MO	reg2	direct	ext	ext
40	LP	MP	reg1	reg3	ext	ext
41	RT	RU	ext	ext	ext	direct
42	RU	RV	direct	ext	direct	reg1
43	QU	RU	ext	direct	ext	reg2
44	QV	RV	ext	reg1	ext	reg1
45	ST	SU	ext	ext	ext	direct
46	SU	SV	direct	ext	direct	reg3
47	RU	SU	reg2	direct	ext	ext
48	RV	SV	reg1	reg3	ext	ext

4.4 Implementation

4.4.1 System Architecture

Deblocking filter with 2-D processing order proposed by [16] and internal memory-free architecture is implemented on Spartan-3 series XC3S2000 FPGA on evaluation board.

System architecture is similar with the one used for inverse transform & quantization. It is as illustrated in Figure 4-16. Input frame and control information is sent through serial channel to be loaded to 2MB SRAM on board. Frame parts are sent to on-chip memory of FPGA and deblocking filter architecture processes par of input frame using the control information and writes the result back to memory. Filtered frame parts are written back to SRAM and finally sent to PC through serial channel. Software developed in MATLAB® handles the serial communication between the board and PC. Embedded processor on FPGA is again used to control the flow of data between frame, pixel and parameter memories and PC.

Deblocking filter hardware is the only component which is designed to use in an H.264 hardware decoder. Other components in the system architecture are again used to control the flow of data between deblocking filter and serial communication software. Therefore their performance in terms of memory use, processing speed, gate count etc. is ignored during evaluation of deblocking filter performance.

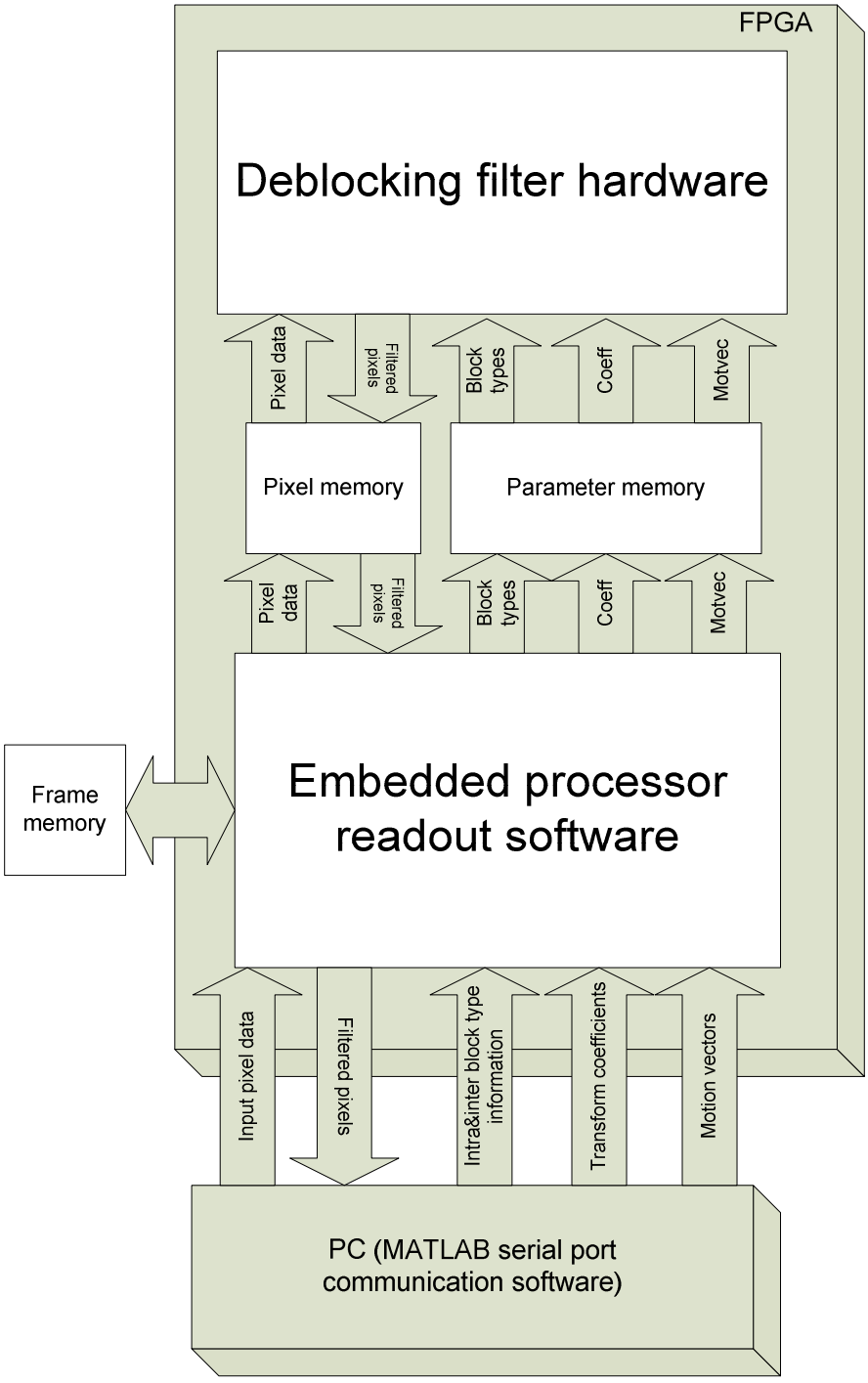

Figure 4-16. System architecture for deblocking filtering

4.4.2 Preparation of Input Frame and Control Parameters

Deblocking filtering is used to filter out blocking artifacts of a reconstructed frame in a H.264 encoder/decoder structure. In order to form input data that is applied to the designed hardware, a simple encoder and decoder software are implemented in MATLAB®. The purpose of the software is to emulate the processing blocks in the H.264 hardware decoder that forms the required inputs to deblocking filter hardware. During the design of this simple codec, bitrate, quality and processing speed considerations are not taken into account. Simple motion estimation and prediction modes are used. Actually the implementations are basic which are not fully conformable with a standard H.264 encoder/decoder system but adequate to generate a frame with blocking artifacts and required control data to a standard H.264 deblocking filter.

4.4.2.1 Encoder

The encoder software takes two adjacent frames as an input, obtains the motion vectors and motion compensated frame to form the residual frame. The output of the encoder is transformed and quantized residual frame. Quantization parameter is an input to the system as a controller for quality loss. Encoder software architecture is illustrated in Figure 4-17.

For motion estimation, simple full-pel block matching method is used in a search window of 12 pixels with fixed 4x4 prediction block size. Obtained motion vectors are used to estimate the second frame from the reference frame. Areas which are not predicted by motion compensation or have very low pixel difference with respect to reference frame are intra coded using three different prediction modes for 4x4 or 16x16 blocks, such as horizontal, vertical and DC modes. The prediction modes are decided considering the SAD for 4x4 or 16x16 blocks, as intra4x4 or intra16x16 respectively. Prediction mode information for other blocks which are predicted using motion compensation is set as inter predicted.

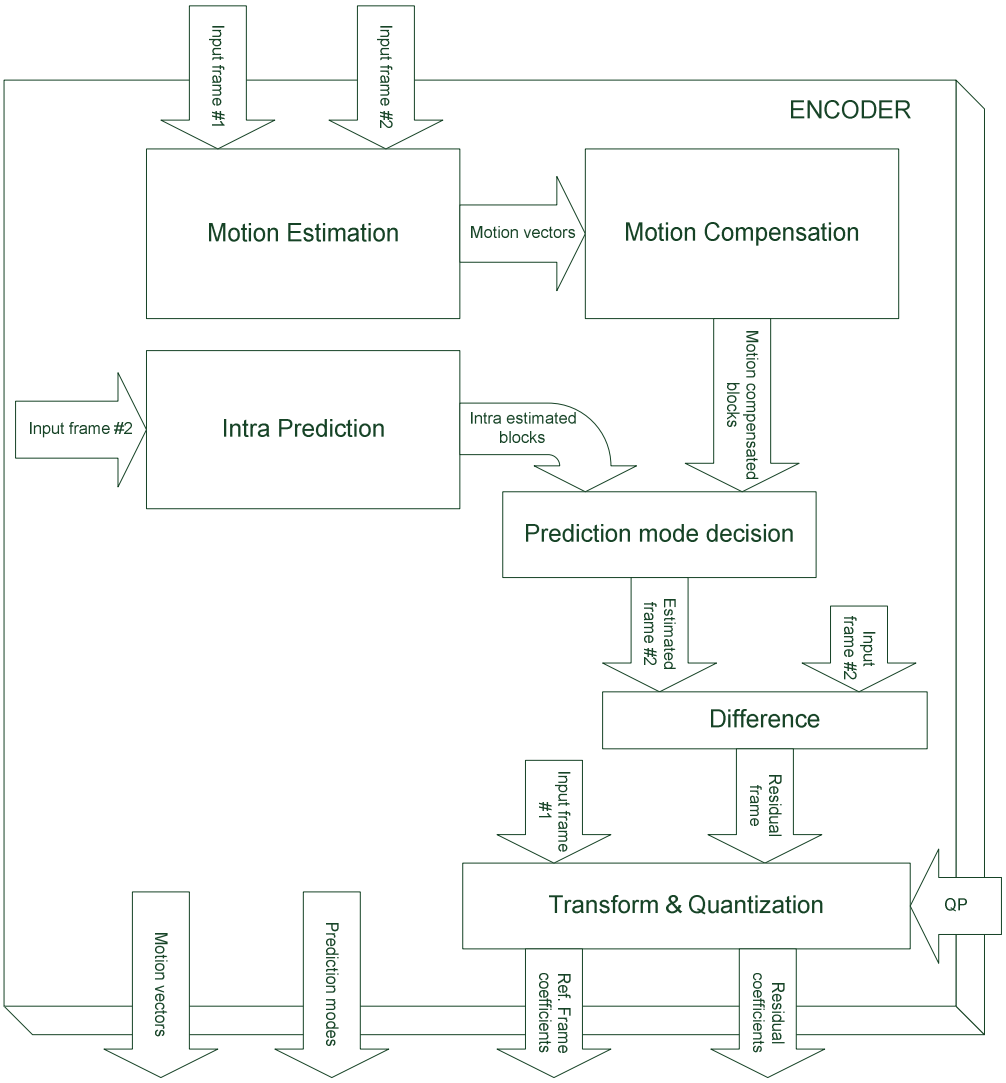

Figure 4-17. Encoder software architecture

4.4.2.2 Decoder

The decoder software simply takes the quantized residual frame coefficients, quantized reference frame coefficients, prediction mode information and motion vectors to obtain the decoded residual. Then it forms the estimated frame using motion vectors and reference frame. Obtained estimated frame is added to decoded residual to form the decoded frame output. Decoder uses the same *QP* with encoder. Decoder software architecture is illustrated in Figure 4-18. Finally obtained decoded frame with blocking artifacts, motion vectors, prediction mode

information and corresponding quantized coefficients are fed to serial communication software to be sent to deblocking filter hardware.

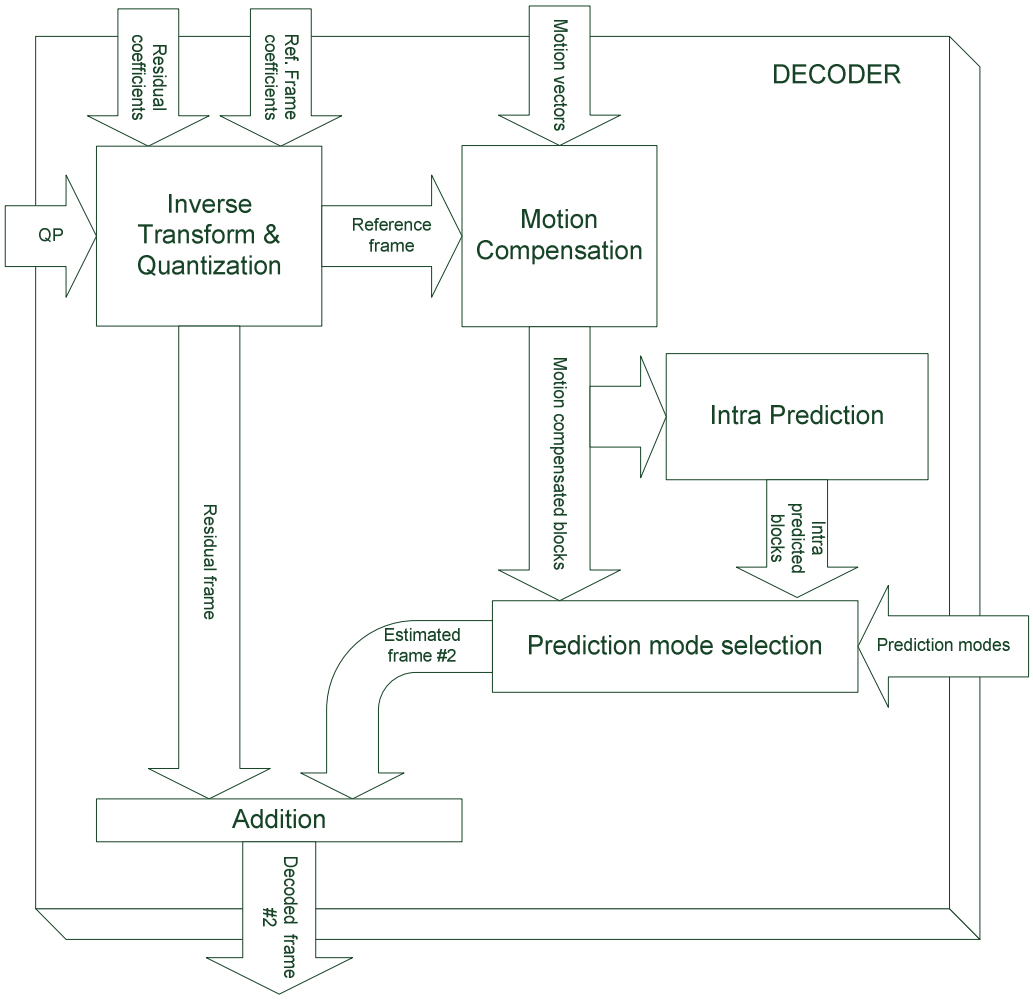

Figure 4-18. Decoder software architecture

4.4.3 Serial Data Transfer

On the PC side, parameter and pixel data for deblocking filtering is sent through serial port using serial communication software. Luma (Y) and chroma (Cb and Cr) components of image, intra/inter prediction mode flag, intra16x16 mode flag, quantized coefficients and motion vectors (in x and y directions) are sent to board

in sequential order. The software running on embedded processor inside FPGA reads the data and acknowledges for each group of data to control the flow.

As in the case for inverse transform & quantization implementation, the purpose of embedded processor implemented in FPGA is again to control the data transfer between serial communication software and designed deblocking filter hardware. The tasks of software running on embedded processor are to read frame and parameter data from serial port, write it to parameter and frame memories obeying memory organization, trigger operation of bS determination and deblocking filtering, read results of deblocking filter from frame memory and send back the results to serial port. Embedded processor software has again no task assigned related to processing for deblocking filtering.

4.4.4 Boundary Strength Determination

Boundary strength (*bS*) value of each block edge is determined by a designed hardware. "Boundary strength generator" reads prediction type, motion vectors and quantized coefficients from parameter memory for each macroblock and determines the *bS* value of each block edge by sequentially checking for the conditions described in Table 4-1. Obtained *bS* values are loaded to a "*bS* memory" which is further read by deblocking filter during filtering.

4.4.4.1 Parameter Memory Organization

Parameter memory is designed to store enough parameter data for a 48x48 image with 4:2:0 sampling rate. The purpose of 48x48 resolution selection is limiting memory resources in Spartan-3 XC3S2000 FPGA. On the other hand this resolution is enough to see deblocking filter performance and is not related with deblocking filter architecture design which operates based on a macroblock as a processed unit. Therefore 16KB resource is selected as parameter memory. Parameter memory organization is as illustrated in Table 4-12.

Table 4-12. Parameter memory organization

Data type	Memory window base address	Memory window size
Motion vector (x)	0x0	256bytes
Motion vector (y)	0x100	256bytes
Intra/Inter prediction mode flag	0x200	256bytes
Intra16x16 prediction mode flag	0x1000	4Kbytes
Transformed coefficients	0x2000	4Kbytes

Motion vectors are 12x12 matrices for a 48x48 image which correspond to motion activity of each 4x4 block. Motion vectors are separately obtained for both directions. Intra & Inter prediction mode flag is again a 12x12 matrix that represents the prediction type of each 4x4 block in a 48x48 image. Intra16x16 prediction mode flag is separate from intra & inter prediction mode flag which points out the 16x16 intra coded macroblocks. In order to ease macroblock edge detection this flag is set to be a 48x48 matrix which have all zero entries except ones for intra 16x16 prediction block edges. Intra macroblock edges are critical to be detected for boundary strength determination. Transformed and quantized coefficients also form a matrix of 48x48 for a 48x48 image. It is used to decide for lower *bS* values.

All of the mentioned parameter data is obtained by designed encoder software and sent to board by serial communication software. Embedded processor on FPGA gets the data and fills it to parameter memory with previously mentioned memory organization. Boundary strength generator is then ready for *bS* determination.

4.4.4.2 Boundary Strength Generator Hardware

Boundary strength generator is separately designed from deblocking filter. A large finite state machine is operated to check the conditions for *bS* determination and set the *bS* value for corresponding block edge. It realizes the boundary strength determination algorithm illustrated in Figure 4-19.

Operation is started by embedded processor by asserting *go_command* signal for the boundary strength generator after parameter and frame memories are filled with data. Boundary strength determination state machine processes 4 horizontal and 4 vertical edges in parallel and jump to next 4 when their *bS* values are determined. Therefore duration of *bS* determination for 4 edge groups varies depending on the *bS* value. If all edges in the group are intra block boundaries, duration is short whereas it lasts longer for inter block edges.

Determined horizontal and vertical *bS* values are loaded to separate buffers to be written to *bS* memory. Another state machine writes the content of buffers to a dual port memory called *bS* memory to be read from deblocking filter. Two state machines, which are responsible for *bS* determination and memory access, are two stages of a pipeline so that first stage is not stalled by second stage. This would not be possible if it was implemented on a general purpose processor. *bS* memory is designed to store 12x12 horizontal and vertical edges to be used for 48x48 input images. Deblocking filter is activated after *bS* of all edges of 48x48 image is determined. This would be changed when integrated to macroblock based H.264 decoder design.

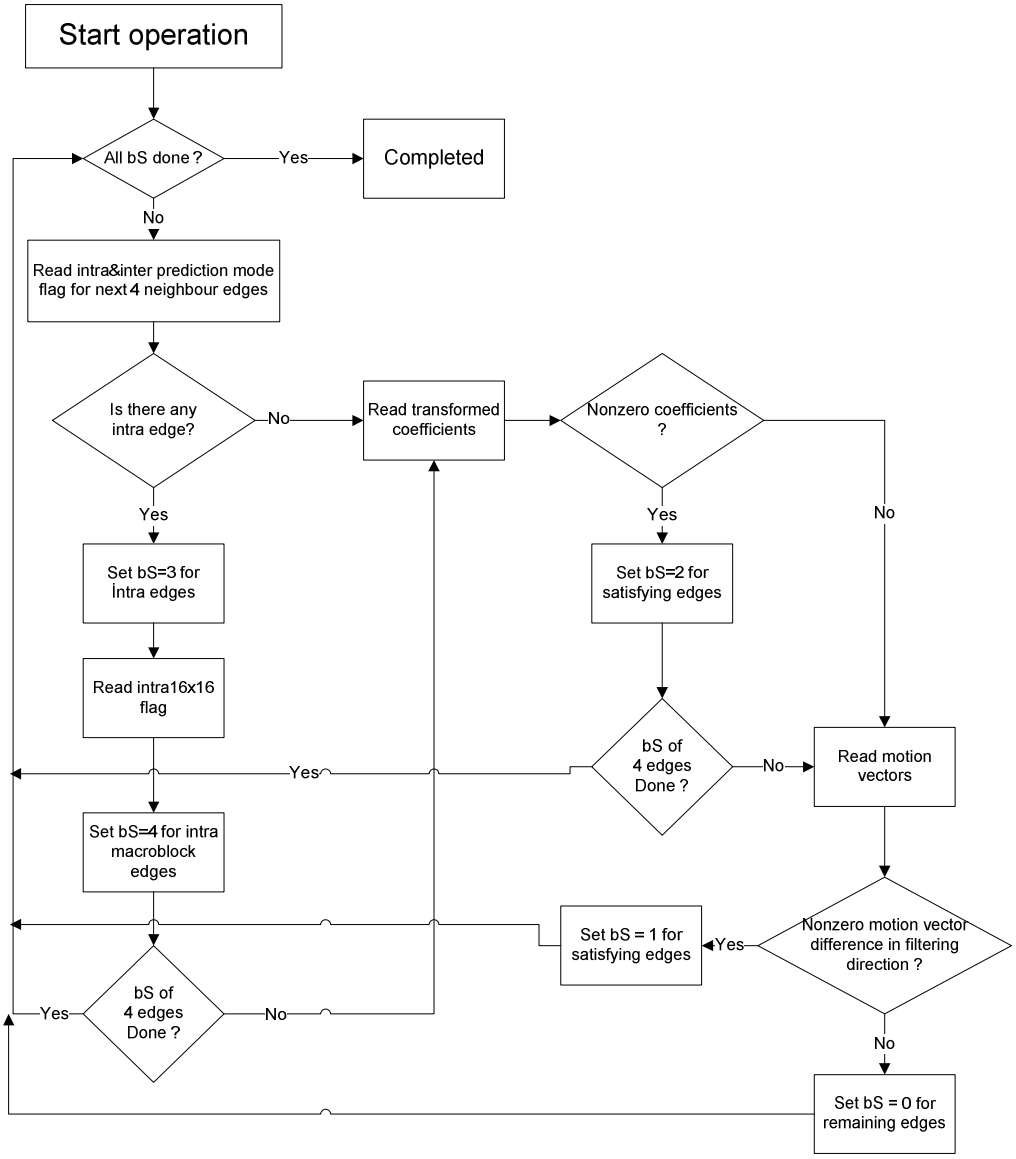

Figure 4-19. Flow chart of boundary strength determination process

4.4.5 Deblocking Filtering

Deblocking filtering is a complicated operation and therefore should be partitioned into tasks that can be handled by independent processing units. The overall architecture of designed deblocking filter hardware is illustrated in Figure 4-20. There are four main blocks in the architecture. Details of boundary strength generator are previously mentioned. The output of boundary strength generator, *bS* values, are used by deblocking filter to determine the filter type and threshold coefficients for filtering decision.

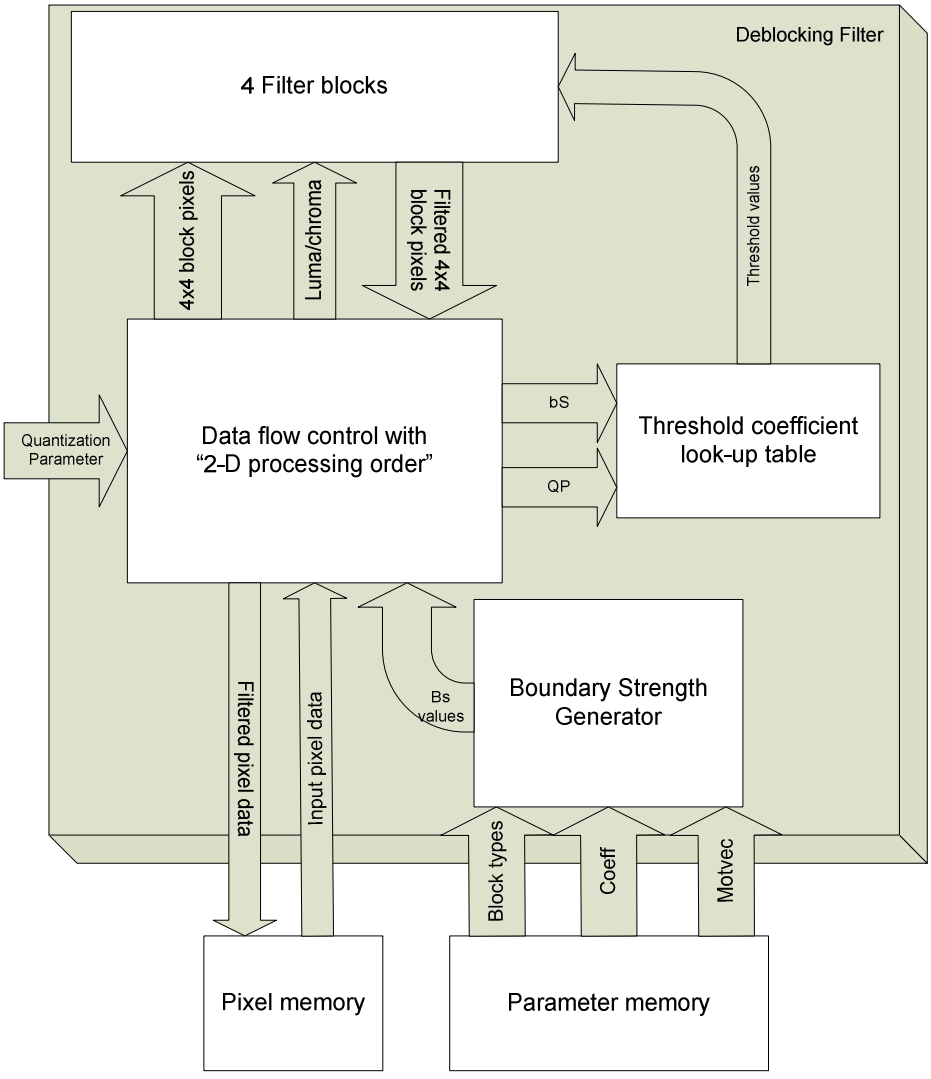

Figure 4-20. Deblocking filter hardware architecture

4.4.5.1 Threshold Coefficient Look-up Table

Threshold coefficients α, β and t_{c0} are to be determined for each edge depending on the boundary strength and quantization parameter value. In order to accomplish this, a look-up table unit is designed to output corresponding threshold values for given *bS* and *QP*. Structure of this unit is as shown in Figure 4-21. It simply includes look-up tables that have fixed table values defined in the standard [6] for α, β and t_{c0}, as illustrated before in Table 4-2 and Table 4-3. Look-up tables are combinatorial elements such that output is directly driven almost at the same time with asserted input, with a negligible propagation delay. Therefore there isn't any delay for reading threshold coefficients from look-up tables which would degrade performance.

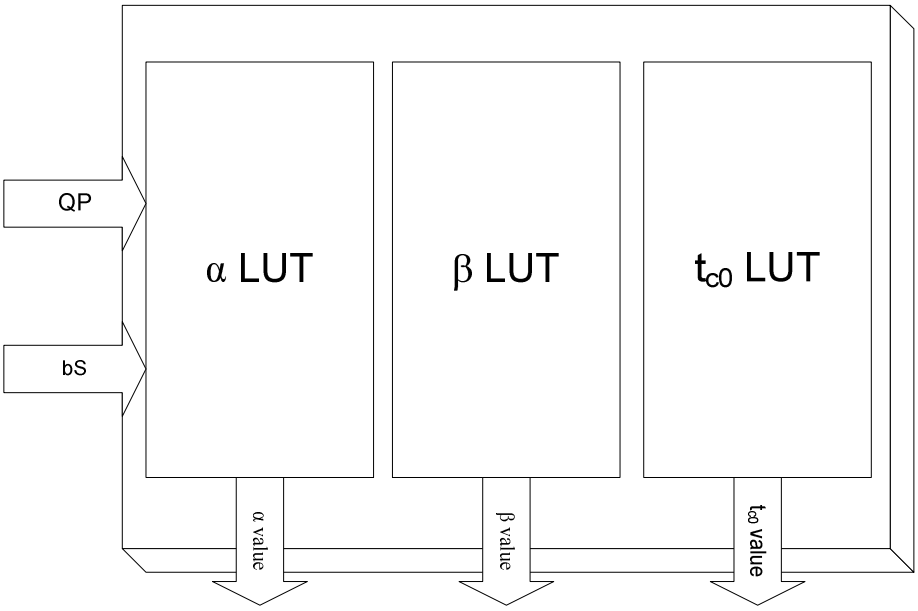

Figure 4-21. Threshold coefficient look-up table unit

4.4.5.2 Filter Blocks

Filter blocks are identical units such that each one can process an edge with 4 pixels at both sides with threshold coefficients, *bS* value and sample type (luma/chroma) are given as inputs. 4 filter blocks form a filtering unit that can process 4x4 block edge with a combinatorial logic without processing delay. A filter block is responsible to decide on filtering and its type by analyzing input pixels and boundary strength. Then the appropriate filtering is applied on input block pixels. Structure of the filter block is as illustrated in Figure 4-22.

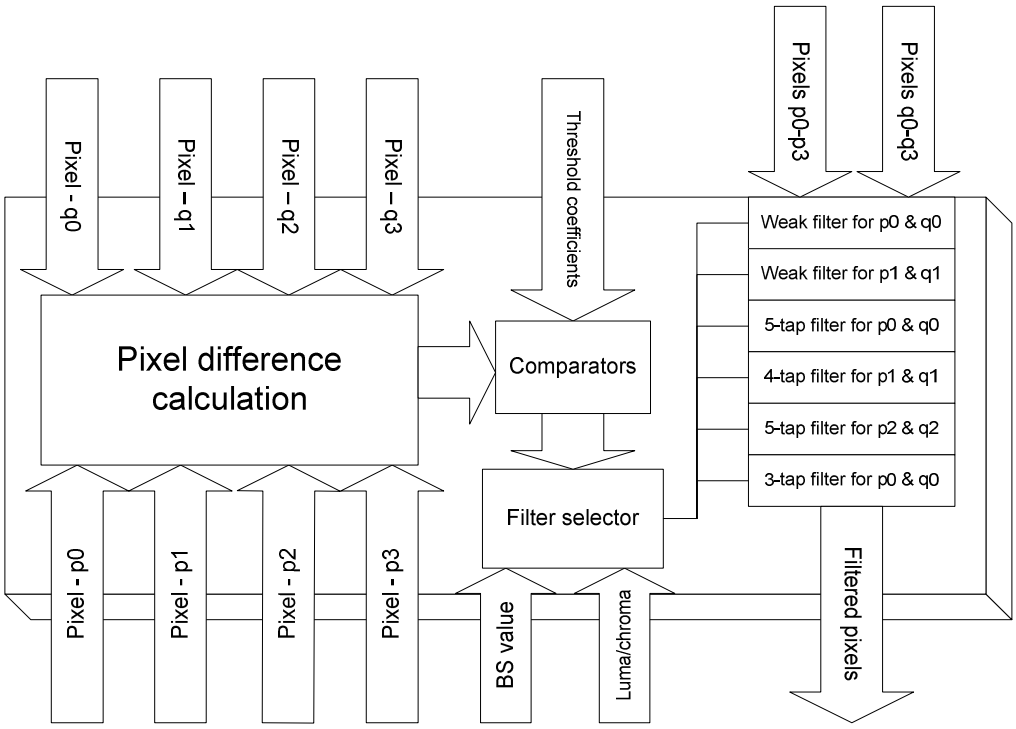

Figure 4-22. Structure of a filter block

Pixels near a block edge, shown in Figure 4-23 as p_0-p_3 and q_0-q_3, are taken as inputs and absolute values of differences related to filtering constraints are calculated. Absolute values are compared by threshold values as defined by the constraints by the logic illustrated in Figure 4-24. Comparator outputs signal are asserted when the corresponding constraint is satisfied.

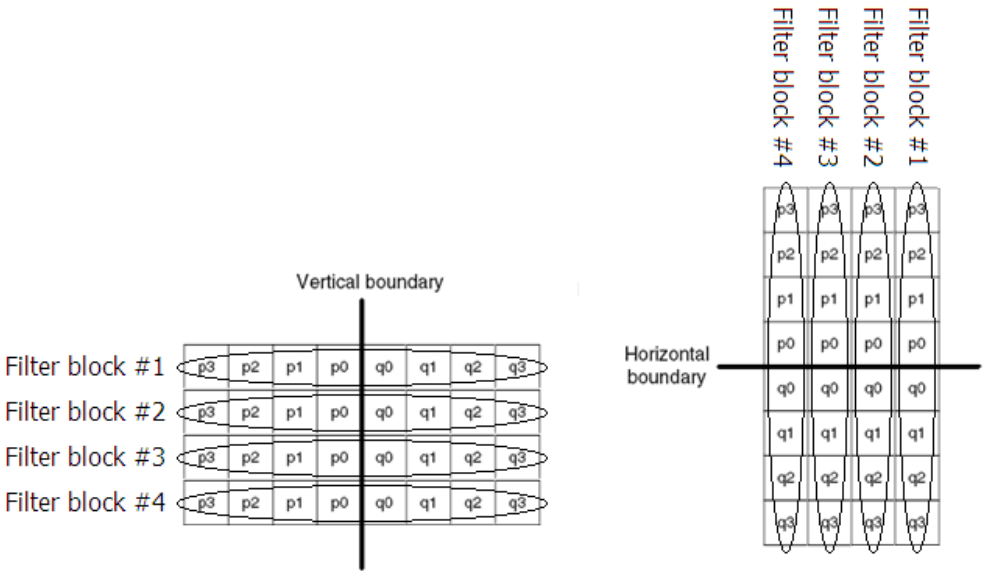

Figure 4-23. Vertical and horizontal filtering of pixels around 4x4 block edge

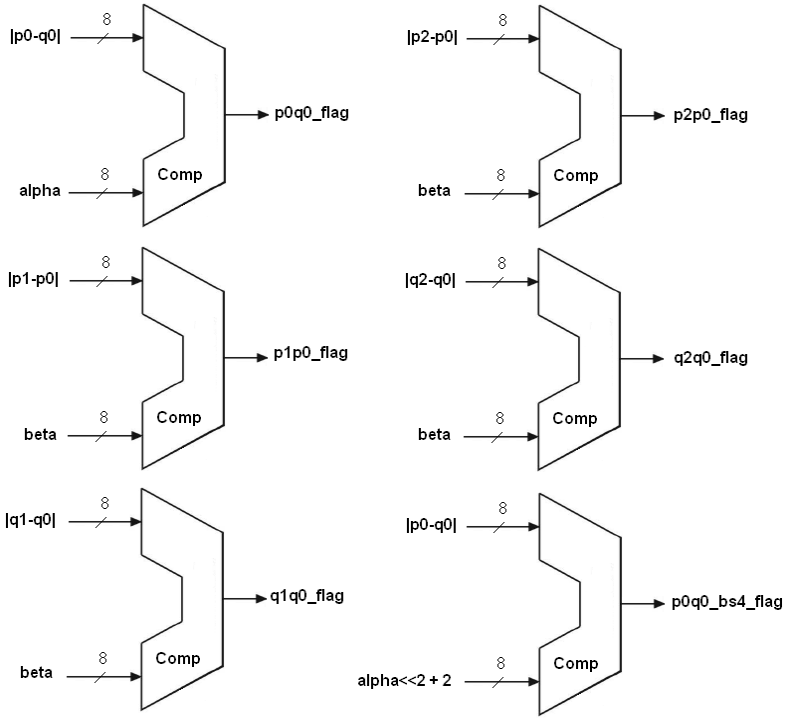

Figure 4-24. Comparison logic for filtering decision constraints

Asserted signals by the comparators are used to enable/disable all filters and some of them for special cases. When *bS* is nonzero and constraints defined in (4.2) and (4.3) are satisfied filtering is generally enabled for all types of filters, with asserting *filter_enable* signal. Filtering for luma pixels p_1, p_2 and q_1, q_2 depends on the constraints (4.2) and (4.3) respectively and a logic asserts *filter_p1p2_enable* and *filter_q1q2_enable* signals as illustrated in Figure 4-25 when constraints are satisfied. These signals select filter outputs for p_1, p_2 and q_1, q_2 respectively.

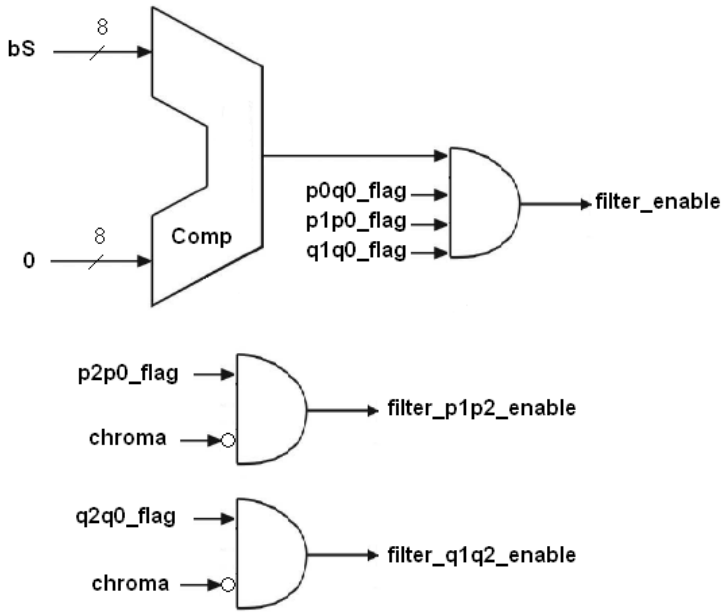

Figure 4-25. Filtering enable signals

Filtering structures are independently implemented and filter outputs are selected with a multiplexer for each pixel input. Multiplexer select signals are obtained by *bS* filter enabling signals. There are mainly 2 filter structures used in deblocking filtering. 4 copies of these structures are used to implement 4x4 block edge filtering.

For *bS < 4 case*, filtering expressions are previously mentioned by (4.12) and (4.13). The structures to implement these are as illustrated in Figure 4-26. The same structures are also used for filtering of q_0' and q_1' samples.

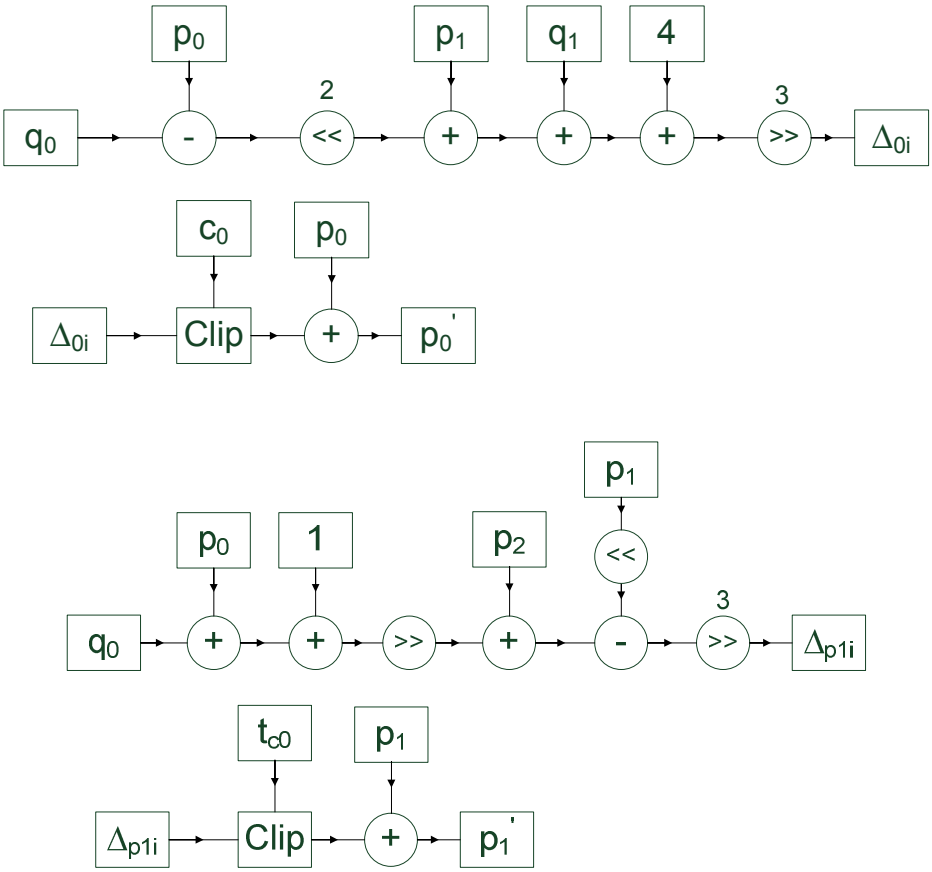

Figure 4-26. Filtering structures for p_0' and p_1' used for the case $bS < 4$

For $bS = 4$ *case*, filtering expressions are previously mentioned by (4.19), (4.20), (4.21) and (4.22). The structures used to obtain filtered samples, p_0', p_1', and p_2', are illustrated in Figure 4-27. Only additions and shift operations are used. Copies of same structures are used for filtering of q_0', q_1', and q_2'.

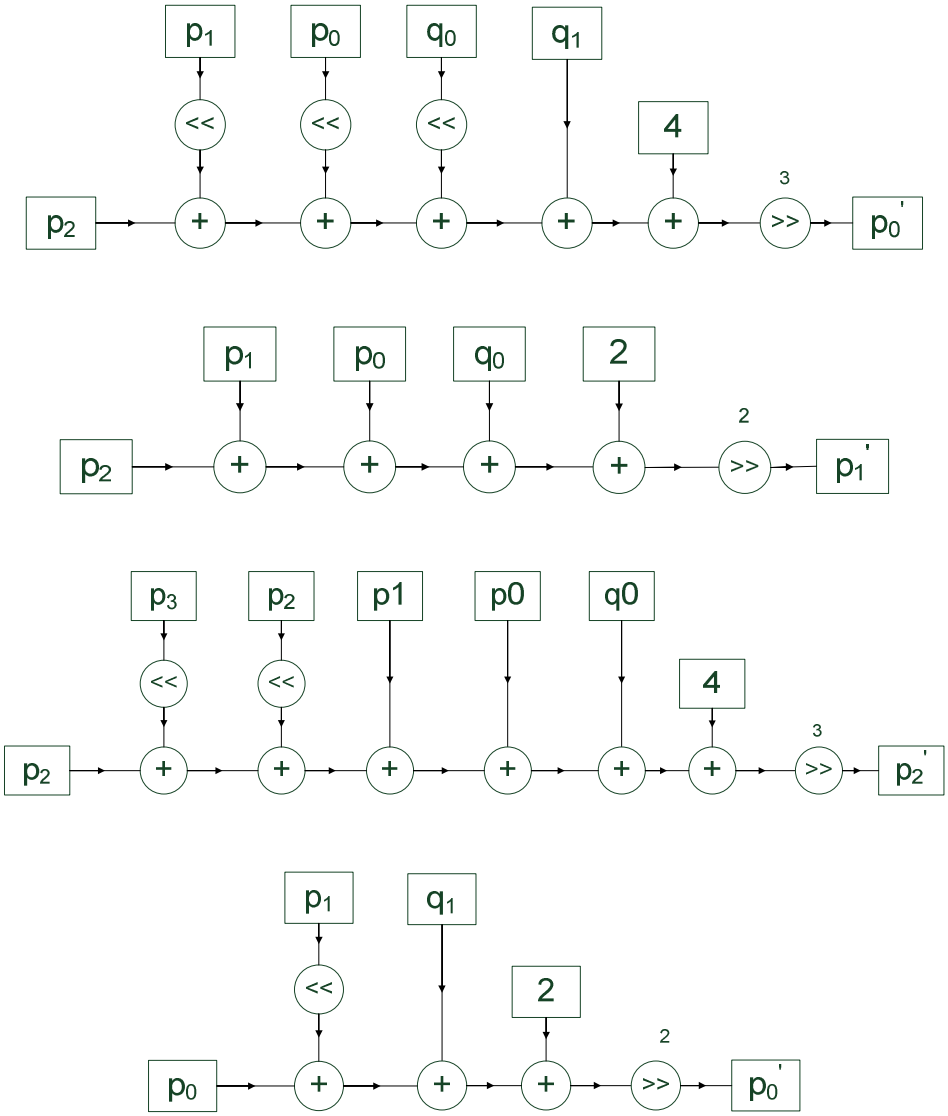

Figure 4-27. Filtering structures for p_0', p_1', and p_2' used for the case $bS = 4$

Filter blocks are initially appropriate for vertical edge filtering, for horizontal edge filtering case a *transpose_command* signal is asserted to filter blocks to transpose totally 32 input and 32 output registers (corresponding to 2 4x4 blocks) to allow filtering in other dimension.

4.4.5.3 Dataflow Control

Dataflow control is the main block of deblocking filter. Boundary strength of each edge is previously determined by *bS* generator and resides in *bS* memory. Dataflow control unit is triggered after *bS* values are completely determined. There are mainly 3 operation states in dataflow control state machine, prepare for reading input pixels from appropriate sources, read input pixels and write filtered pixels to appropriate destinations. The sources and destinations of pixels are nothing but the register set and external (frame) memory. Frame memory is called external memory since it is outside the deblocking filter hardware, which would be an external memory device to deblocking filter IC.

The operation of dataflow control unit is illustrated in Figure 4-28. Firstly *QP* is read from parameter memory, 8 *bS* values (4 for horizontal and 4 for vertical edges) are read from *bS* memory and filled to a buffer. *bS* value corresponding to currently processed edge is read from this buffer. Following the 2-D processing order, the required access to read two input blocks near the edge, to read from frame memory or register set is done. In order to read from external memory, a read command signal is asserted to be detected by memory controller state machine which reads specified address of external memory. For register access, specified register in processing order is accessed.

When external memory is read, a 4x4 block data (128-bits) is read in 4 cycles with 32-bit accesses. Therefore operation flow waits until data is ready. If two blocks are to be read from external memory, in 8 clock cycles data is ready. Else if only one block is read from memory and the other is read from register set, in 4 clock cycles data is ready. If both are read from register set, there is no wait delay.

After input pixels are read, they are sent to be filtered in parallel by 4 filter blocks. A single clock cycle is enough to obtain filtered pixels of 2 4x4 blocks near an edge. Filtered blocks are written to registers if they will further be used. Otherwise,

if all edges of a block is filtered, it is written to external memory. Before read and write accesses, memory controller state machines are checked if they are idle. During read access from external memory, *read_command* signal and read address is sent to memory controller of read port and dataflow state machine waits for data ready signal in the next state to get the read data. During write access to external memory, *write_command* signal, write address and data are sent to memory controller of write port and dataflow state machine jumps to next stage without waiting for memory controller. Since both ports of external memory are accessed from deblocking filter side, read and write accesses can be done concurrently.

After all luma edges are filtered, chroma samples are read from frame memory. Accesses required for chroma filtering, as previously shown in Table 4-11, are conducted. After all of edges are completed, data flow state machine goes back to initial state.

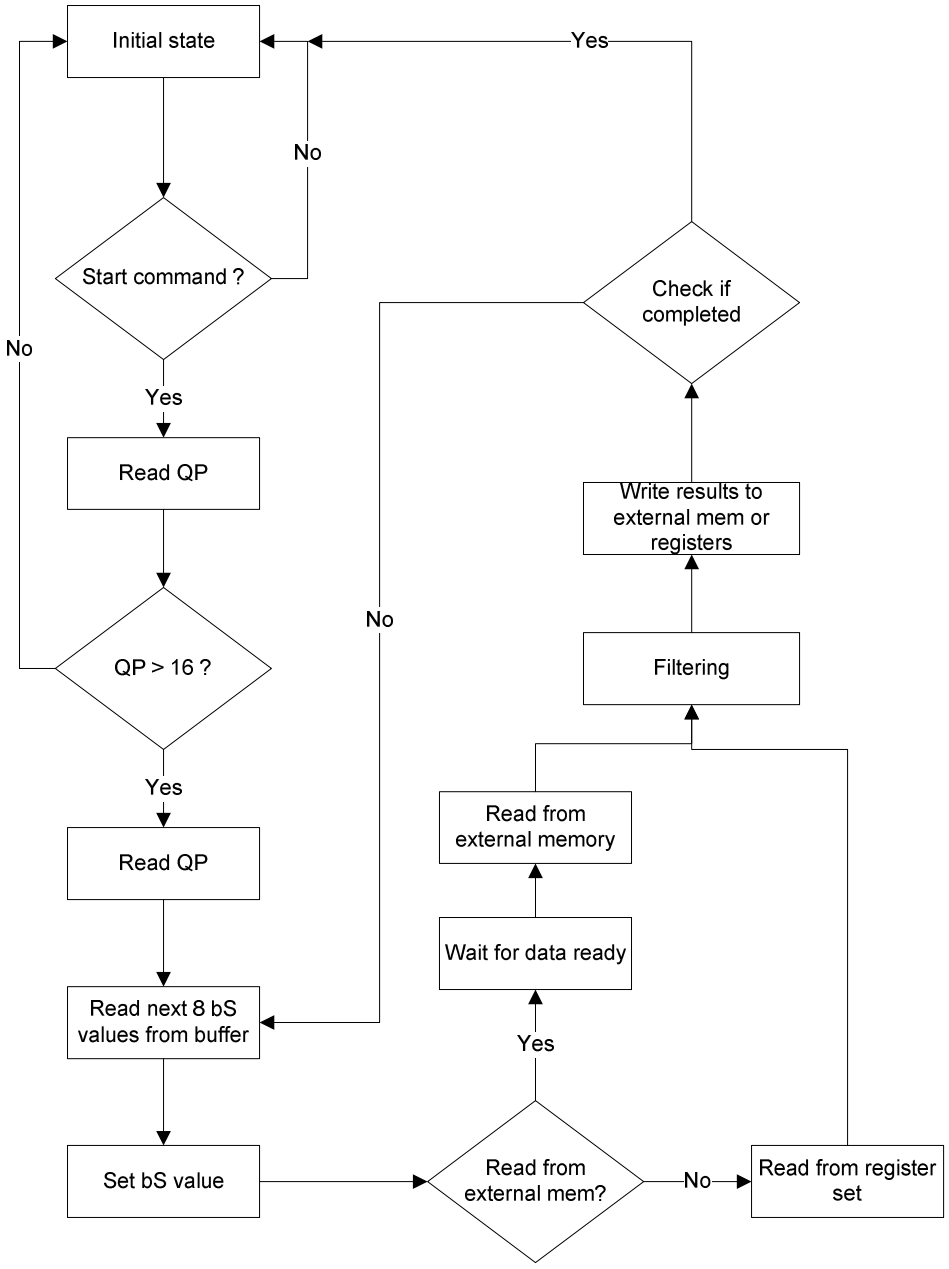

Figure 4-28. Operation of dataflow control unit

4.5 Results

Designed system is tested with a frame randomly grabbed from “foreman” sequence. Selected frame resolution is CIF (352x288) but actually sent to deblocking filter hardware as 48x48 image parts. Higher resolution images would also be tested by this method but actually designed device can support up to a limited resolution for real-time operation.

As QP gets larger filter performance is more observable, since blocking artifacts due to coarse quantization increases. Subjective quality improvement is observed by comparison of decoder output and filtered images, as illustrated in Figure 4-29 and 4-30. Objective quality improvement is measured as PSNR improvement and it is slightly increased from 31.64 dB to 31.85 dB for this frame.

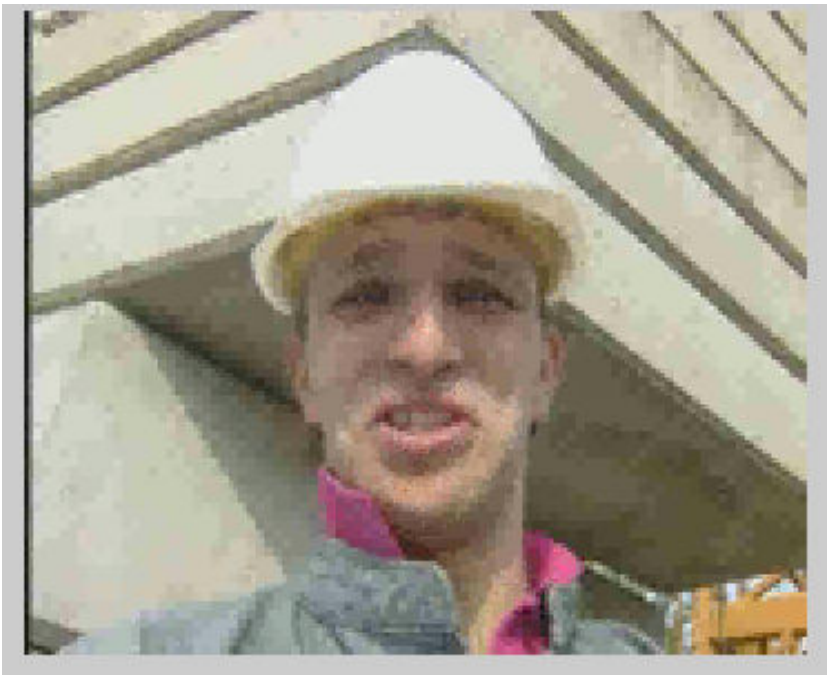

Figure 4-29. Decoder output for *QP=35*

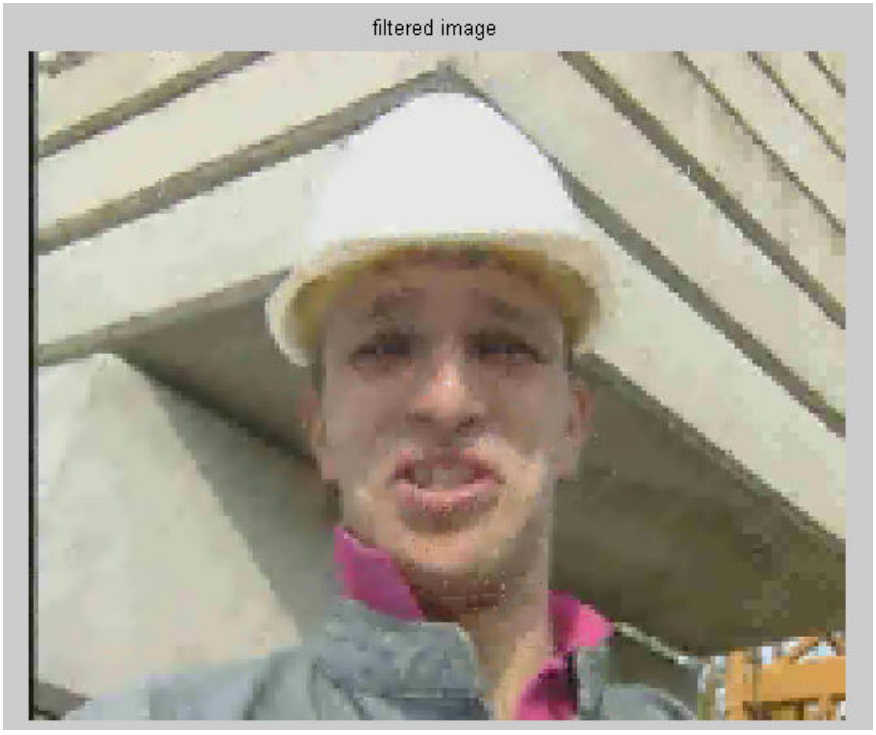

Figure 4-30. Filtered image *QP=35*

Designed system obtains the filtered output at 336 clock cycles for each macroblock (MB). Therefore hardware can process a CIF frame in 3.98 ms with 33 MHz operating clock. (396 macroblocks x 336 clock cycles/MB x 30ns clock period = 3.98 ms) So it can process 1000/3.98 = 251 CIF frames. With 33MHz operating clock it can also process 704x562 frames since it can process 62.75 fps. For CIF and QCIF resolutions operating clock frequency can be reduced to obtain lower power consumption. Some operating clock frequencies recommended to be used for different resolutions are illustrated in Table 4-13. Frequencies are given for 30fps processing rate. By doubling the clock frequency, 60fps performance can also be achieved.

Table 4-13. Recommended operating clocks

resolution	operating clock	processing rate
176x144	1 MHz	30fps
352x288	4 MHz	30fps
704x562	15 MHz	30fps
1408x1124	60 MHz	30fps

Performance of system is compared with other architectures proposed in literature. Most of them are designed to support HD resolutions with high processing clocks up to 200MHz. In terms of processing speed, system achieves a sufficient performance for a low power device, decoding much lower resolutions compared to HD supporting devices. Also designed hardware doesn't have internal memory to buffer intermediate filtering results whereas all architectures in literature have for buffering. Comparison with some of the architectures in literature is given in Table 4-14.

Table 4-14. Performance comparison

	Cycle/MB	Platform
Proposed	336	FPGA
[13]	192	VLSI
[16]	446	VLSI
[17]	614	VLSI
[18]	336	VLSI
[19]	246	FPGA
[20]	243	VLSI
[21]	300	VLSI
[22]	>600	FPGA
[23]	2268	VLSI
[25]	5600	FPGA
[34]	5600	FPGA
[35]	204	VLSI

Logic resource utilization of designed hardware in FPGA is given in Figure 4-31.

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,807	40,960	4%
Number of 4 input LUTs	6,641	40,960	16%
Logic Distribution			
Number of occupied Slices	4,027	20,480	19%
Number of Slices containing only related logic	4,027	4,027	100%
Number of Slices containing unrelated logic	0	4,027	0%
Total Number of 4 input LUTs	6,936	40,960	16%
Number used as logic	6,641		
Number used as a route-thru	295		
Number of bonded IOBs	117	489	23%
IOB Flip Flops	95		
Number of GCLKs	1	8	12%

Figure 4-31. Logic resource utilization in FPGA

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

The main objective of this study is to develop hardware architectures for power efficient H.264 decoder. Mainly two building blocks of decoder is selected and implemented with proposed architectures. In both architectures internal structures are designed to be simple enough to be used for a low complexity system.

At the beginning of study, power and complexity reduction techniques used in various parts of H.264 video coding structure are investigated. Physical methods are found to be effective but platform-dependent. Architectural methods are also effective but they actually depend on the chosen hardware algorithm which is implemented by the designed architecture. Therefore throughout this work, hardware algorithms for power efficient architectures in literature are investigated and modified to obtain solutions for selected parts of H.264 decoder.

As a starting point, inverse transform & quantization part is selected to be implemented. During implementation, it is found that by limiting *QP*, input buffering performance of the unit can be increased by reducing memory accesses and internal transform units can be modified with lower complex architectures. By this limitation it is observed that still 36.6 dB PSNR can be obtained for *QP*=22, which is satisfactory in terms of objective quality. Also obtained decoded frame is observed to have subjective quality for this *QP*.

Secondly, deblocking filter part is selected, which has significant computational complexity in H.264 decoder. Deblocking filter architectures in literature are investigated. Most of the structures in literature focus on high performance deblocking filtering for high resolutions with power-inefficient architectures. Power-efficient architectures are also investigated and data reuse is found to be the critical part of power-efficient design. Processing orders in literature that are used for edge filtering are analyzed and the best processing order in terms of data reuse is selected to be used for hardware design. Architecture is designed obeying this processing order and architecture with efficient data reuse and lowest possible external memory access is achieved. Performance degradation compared to high resolution solutions in literature is not critical since high resolutions are usually not supported by mobile decoders.

Designs are proposed to be used for a low-power baseline profile H.264 hardware decoder in a mobile device that supports low resolutions and bitstream with moderate bitrate. Operating frequency of designed hardware is chosen low to be used for a low-power which can be increased to support higher resolutions.

FPGA's are usually used in prototyping stage before ASIC design of high volume products, such as for multimedia market. The reason is the cost of FPGA for high-volume production and redundant logic in FPGA which degrades performance in terms of power and processing speed. Redundancy comes from the configurable logic blocks in FPGA which are not completely used for a design, consume power and degrade achievable clock speeds. Because of this reason, power measurement is not conducted throughout this work. Therefore architectures designed and verified throughout this work are not the final solutions for a low power H.264 decoder product, but can be migrated to an ASIC design with small modifications so that an optimum solution can be achieved. Like the examples in literature [35, 36], power consumption of an ASIC solution is more reasonable to be measured since ASIC design is optimum in terms of logic utilization.

As a future study, proposed inverse transform & quantization architecture can be modified to implement 8x8 inverse transform used in FRExt extension of H.264 [8]. Proposed deblocking filter architecture can be re-designed with a pipelined structure to support higher resolutions with same processing order and operating clock frequency . On the other hand, both architectures can be modified to support, pixel-widths up to 12-bits and other sampling ratios (4:2:2, 4:4:4).

REFERENCES

[1] Nokia H.264, ftp://standards.polycom.com/IMTC_Media_Coding_AG, updated September 25th 2005, visited June 20th 2009

[2] ISO/IEC International Standard 11172; "Coding of moving pictures and associated audio for digital storage media up to about 1.5 Mbits/s", November 1993

[3] ISO/IEC International Standard 13818, "Generic coding of moving pictures and associated audio information", November 1994

[4] ITU-T Recommendation H.261, “Video Codec for Audiovisual Services at px64 kbit/s”, 1993

[5] ITU-T Recommendation H.263, “Video Coding for very Low Bit rate Communication”, 1996

[6] Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, "Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264/ISO/IEC 14496-10 AVC)", JVT-G050, March 2003.

[7] T. Wiegand, G. J. Sullivan, G. Bjontegaard and A. Luthra, “Overview of the H.264/AVC Video Coding Standard”, Circuits and Systems for Video Technology, IEEE Transactions on Volume 13, Issue 7, July 2003, pp. 560 – 576.

[8] G. J. Sullivan, P. Topiwala, A. Luthra, “The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions”, SPIE Conference on Applications of Digital Image Processing XXVII, August 2004.

[9] Iain E.G. Richardson, H.264 and MPEG-4 Video Compression, UK: Wiley & Sons, 2003.

[10] H. S. Malvar, A. Hallapuro, M. Karczewicz, L. Kerofsky, “Low-complexity transform and quantization in H.264/AVC”, Circuits and Systems for Video Technology, IEEE Transactions on Volume 13, Issue 7, July 2003, pp. 598 – 603.

[11] T. M. Liu, T. A. Lin, S. Z. Wang, W. P. Lee, J. Y. Yang, K. C. Hou and C. Y. Lee, “A 125μW Fully Scalable MPEG-2 and H.264/AVC Video Decoder for Mobile Applications”, IEEE Journal of Solid State Circuits, Volume 42, Issue 1, January 2007, pp. 161-169.

[12] I. Amer, W. Badawy, G. Jullien “A High-Performance Hardware Implementation of the H.264 Simplified 8x8 Transformation and Quantization”, IEEE Conference on Acoustics, Speech and Signal Processing, Pennsylvania, 1137-1140 (2005).

[13] G. Khurana, A. A. Kassim, T. P. Chua, Bi Mi M., “A Pipelined Hardware Implementation of In-loop Deblocking Filter in H.264/AVC”, Consumer Electronics, IEEE Transactions on Volume 52, Issue 2, May 2006, pp. 536 – 540.

[14] T. C. Chen, Y. H. Chen, S. F. Tsai, S. Y. Chien and L. G. Chen, “Fast Algorithm and Architecture Design of Low-Power Integer Motion Estimation for H.264/AVC”, Circuits and Systems for Video Technology, IEEE Transactions on Volume 17, Issue 5, May 2007, pp. 568-577.

[15] P. List, A. Joch, J. Lainema, G. Bjontegaard, M. Karczewicz, “Adaptive Deblocking Filter”, Circuits and Systems for Video Technology, IEEE Transactions on Volume 13, Issue 7, July 2003, pp. 614-619.

[16] B. Sheng, W. Gao and D. Wu, “An Implemented Architecture of Deblocking Filter for H.264/AVC”, International Conference on Image Processing, Singapore, 665-668 (2004).

[17] Y. W. Huang, T. W. Chen, B. Y. Hsieh, T. C. Wang, T. H. Chang, L. G. Chen, “Architecture Design for Deblocking Filter in H.264/JVT/AVC” Proc. IEEE International Conference on Multimedia and Expo., Volume 1, 693-696 (2003).

[18] C. C. Cheng and T. S. Chang, “A Hardware Efficient Deblocking Filter for H.264/AVC”, IEEE Consumer Electronics, ICCE 2005 Digest of Technical Papers International Conference, 235-236 (2005).

[19] S. Y. Shih, C. R. Chang, Y. L. Lin, “A Near Optimal Deblocking Filter for H.264 Advanced Video Coding”, Proc. of the 2006 Asia and South Pacific Design Automation Conference, Yokohama, Japan, 170-175 (2006).

[20] T. M. Liu, W. P. Lee, C. Y. Lee, “An In/Post-Loop Deblocking Filter With Hybrid Filtering Schedule”, Circuits and Systems for Video Technology, IEEE Transactions on Volume 17, Issue 7, July 2007, pp. 937-943.

[21] C. C. Cheng, T. S. Chang, K. B. Lee, “An In-Place Architecture for the Deblocking Filter in H.264/AVC”, Circuits and Systems for Video Technology, IEEE Transactions on Volume 53, Issue 7, July 2006, pp. 530-534.

[22] M. Sima, Y. Zhou, W. Zhang, “An Efficient Architecture for Adaptive Deblocking Filter of H.264/AVC Video Coding”, Consumer Electronics, IEEE Transactions on Volume 50, Issue 1, February 2004, pp. 292-296.

[23] B. J. Kim, J. I. Koo, M. C. Hong, and Seongsoo Lee, “Low-Power H.264 Deblocking Filter Algorithm and Its SoC Implementation”, First Pacific Rim Symposium, Hsinchu, Taiwan, 771-779 (2006).

[24] C. J. Lian, P. C. Tseng and L. G. Chen, “Low Power and Power-Aware Video Codec Design: An Overview”, China Communications, October 2006.

[25] M. Parlak and I. Hamzaoglu, “Low Power H.264 Deblocking Filter Hardware Implementations”, Consumer Electronics, IEEE Transactions on Volume 54, Issue 2, May 2008, pp. 808-816.

[26] O. Tasdizen and I. Hamzaoglu, “A High Performance and Low Cost Hardware Architecture for H.264 Transform and Quantization Algorithms”, 13th European Signal Processing Conference, Antalya, Turkey, September 2005.

[27] T. C. Wang, Y. W. Huang, H. C. Fang and L. G. Chen, “Parallel 4x4 2D Transform and Inverse Transform Architecture for MPEG-4 AVC / H.264”, Proceedings of IEEE ISCAS 2003 – IEEE International Symposium on Circuits and Systems, Bangkok, Thailand, 800-803 (2003).

[28] T.M. Liu, etc. al., "An 865-μW H.264/AVC video decoder for mobile applications", IEEE Asian Solid-State Circuits Conference, 301 – 304 (2005).

[29] Cheng Z., C. Chen, B. Liu and L. Yang, “High Throughput 2-D Transform Architectures for H.264 Advanced Video Coders”, IEEE Asia-Pacific Conference on Circuits and Systems, Fukuoka, Japan, 1141-1144 (2004).

[30] Kordasiewicz, R. and S. Shirani, “Hardware Implementation of the Optimized Transform and Quantization Blocks of H.264”, Canadian Conference on Electrical and Computer Engineering, Ontario, Canada, 943-946 (2004).

[31] H. Y. Lin, Y. C. Chao, C. H. Chen, B. D. Liu and J. F. Yang, “Combined 2-D Transform and Quantization Architectures for H.264 Video Coders”, IEEE International Symposium on Circuits and Systems, Kobe, Japan, 1802-1805 (2005).

[32] E. P. Hong, E. G. Jung, H. Fraz, D. S. Har, “Parallel 4x4 Transform Architecture Based on Bit Extended Arithmetic for H.264/AVC”, International Symposium on Circuits & Systems, Kobe, Japan, 95-98 (2005)

[33] W. Hwangbo, J. Kim and C. M. Kyung, “A High Performance 2-D Inverse Transform Architecture for the H.264/AVC Decoder”, IEEE International Symposium on Circuits and Systems, New Orleans, USA, 1613-1616 (2007).

[34] M. Parlak, I. Hamzaoglu, “An efficient hardware architecture for H.264 adaptive deblocking filter algorithm”, Conference on Adaptive Hardware and Systems, Istanbul, Turkey, 381 – 385 (2006).

[35] K. Xu and C. S. Choy, “A 5-stage Pipeline, 204 Cycles/MB, Single-port SRAM Based Deblocking Filter for H.264/AVC”, Circuits and Systems for Video Technology, IEEE Transactions on Volume 18, Issue 3, 2008, pp. 363-374.

[36] Xilinx Corporation, http://www.xilinx.com, visited May 5th 2009.

[37] Altera Corporation, http://www.altera.com, visited May 5th 2009.

[38] Xilinx, “Xilinx Spartan-3 Development Kit Product Brief, Literature Number: ADS-AA/S3DEV/10_04”, 2004.

APPENDIX A

FPGAs AND DESIGN FLOW

In this appendix, development, structure and design flow of FPGAs will be introduced. First, brief history of FPGA development will be given. Secondly, building blocks and structure of FPGA devices will be explained. Finally design flow of an FPGA application will be discussed.

A.1 Field Programmable Gate Arrays

Programmable logic devices (PLD) are components that are used to build reconfigurable logic circuits for various applications. Field programmable gate arrays are developed based from ROM and PLDs. As the name implies, FPGAs are devices that can be programmable "in field", in other words, in design development stage, different than ROMs and PLDs which have fixed logic circuit. In 1985, Xilinx introduced the first commercial FPGA device that integrates 64 configurable logic blocks with 3-input LUTs and programmable interconnection pattern. In 1990's, production volume and gate integration of FPGA's increased enormously to 600,000 gates in single chip. Now there are a lot of vendors and FPGA devices contain over million gates. [36]

Also microprocessor cores are integrated to FPGA to enable microprocessor and reconfigurable logic implementation on single chip, so called "system on programmable chip" [36]. Additionally dedicated memory blocks, multiplier blocks, dedicated DSP slices, high speed I/O etc. are integrated to different FPGA

families to allow several high-speed signal processing and control applications possible on single chip with peripheral devices. With these additions FPGA design process became more complex then a straight forward logic design. In addition to logic design software, new software is introduced to handle complex design process.

A.2 Structure of FPGA

A.2.1 Configurable Logic Blocks (CLB)

FPGA structure consists of configurable logic blocks (CLB) with interconnections. As illustrated in Figure A-1, a standard configurable logic block contains a 4-input LUT, a multiplexer and D-flip flop. LUTs are actually programmable ROMs that can implement programmed logic functions, in sum-of-products form etc., of their inputs and give the result. Clocked D-flip flop and multiplexer allows implementation of sequential logic circuits. Some FPGA vendors introduce different CLB structures by using 6-input or 8-input LUTs and more logic components to increase performance.

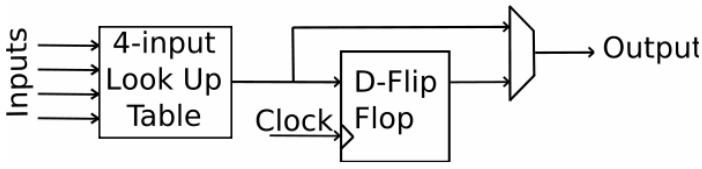

Figure A-1. Standard CLB structure

General structure of a FPGA is a matrix of CLBs connected with programmable interconnection switches, as shown in Figure A-2. Boolean logic equations is converted by software to switch positions and programmed to static RAM on FPGA that controls the switch positions and implements the desired logic circuit. I/O blocks are used to interface designed logic to device pins and external circuit.

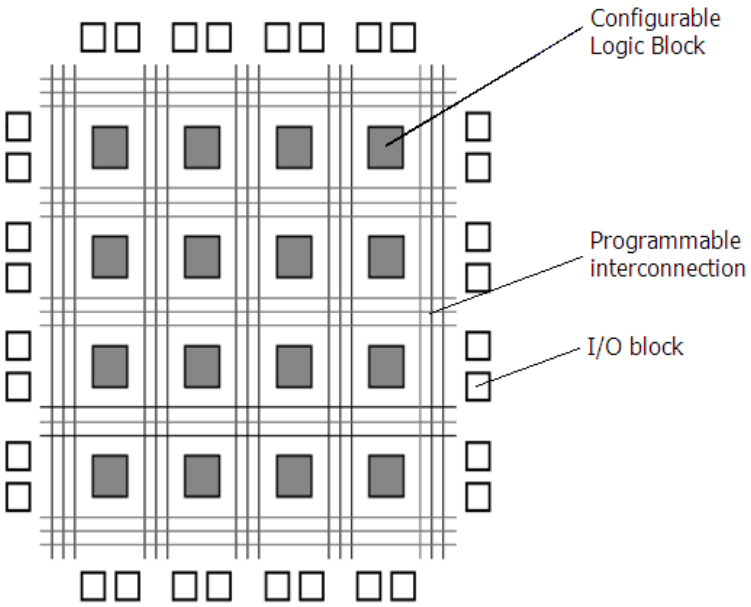

Figure A-2. General structure of FPGAs

A.2.2 Distributed Memory Blocks

New FPGA families have also dedicated blocks of static RAM distributed among and controlled by logic elements. As illustrated in Figure A-3, RAM blocks are surrounded by CLBs and can be accessed by controller logic implemented on these CLBs. The amount of distributed memory in high-end FPGA families is high to allow data buffering in several performance demanding applications such as radar signal and video processing. On the other hand, memory is costly in terms of FPGA chip area; therefore some families have limited memory resources but more CLB slices.

In design stage, memory size, data access bit width (8-bit, 16-bit, etc.), memory type (single-port, dual-port etc.) and memory access type (read-only, random-access) can be configured by user. User is responsible to design the controlling logic which should obey memory access requirements.

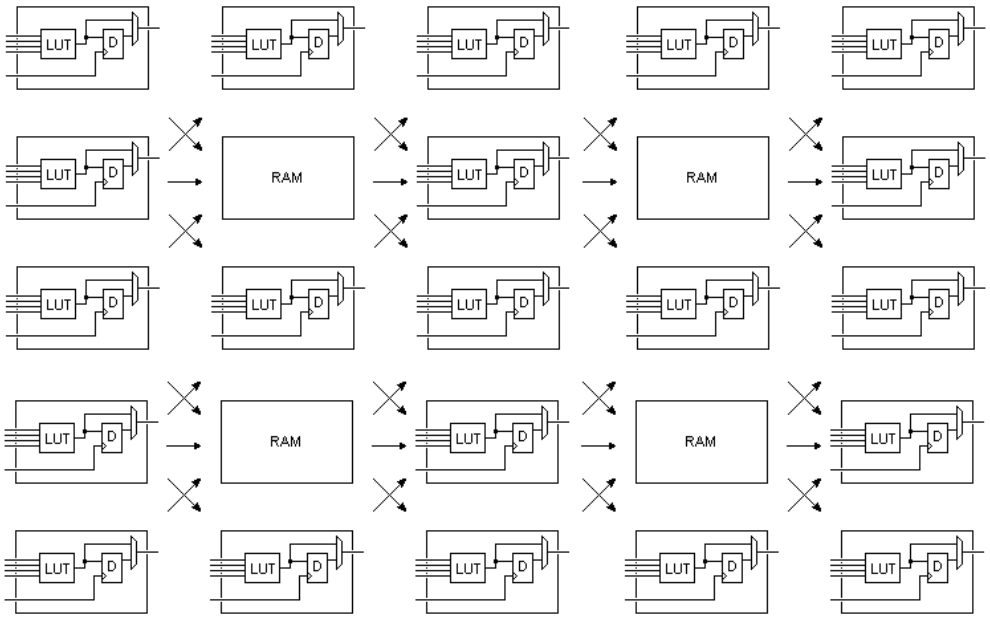

Figure A-3. Distributed memory blocks among FPGA

A.2.3 Arithmetic Processing Blocks

Hardware multipliers and DSP slices are also dedicated blocks in some FPGA families used to increase implementation speed of arithmetic operations.

A.2.4 Digital Clock Management

There are also digital clock management (DCM) units in FPGA which have dedicated digital PLL circuitry to implement clock operations such as frequency multiplication/division, phase shifting etc. The configuration of DCM is done in design stage. Generally, input oscillator clock is multiplied or divided by a scalar to obtain the desired clock frequency used in state machines or as an external clock output.

A.2.5 Embedded processors

Embedded processors, implemented inside FPGA chip, are commonly used to decrease design cycle time and assign tasks that is easy to do by software instead of hardware. In general there are two types of embedded processor implementation. Some FPGA families have embedded hard processor cores already implemented in FPGA chip and accessed and configured by user. They have high performance and even have capability to run real time operating system. For FPGA devices that don't contain hard processor core, soft processor core is implemented using logic resources. Microblaze® by Xilinx and Nios® by Altera are some of commonly used soft processor cores. [36, 37] These cores are much simpler compared to hard cores but still effective for interfacing applications, such as UART, Ethernet, PCI etc. and is available on most of FPGA families. Their disadvantage is consumption of logic resources.

A.2.6 I/O Blocks

I/O blocks are the endpoints that connect input logic circuitry to device pin and external circuit. There are I/O banks in a FPGA that can implement different I/O standards and speed. Designer is responsible to select the right pin that can satisfy required electrical characteristics. FPGA configuration and supply pins are prohibited to be used by designer.

A.3 FPGA Design Flow

A.3.1 HDL Synthesis and Simulation

The first stage of an FPGA design is description of design using a schematic editor or hardware description language (HDL). Schematic editor is an old method and hard to be used for complex designs, therefore mostly HDL is used for design description. Some of the well-known HDL are VHDL, Verilog and ABEL. VHDL is more commonly used in military applications whereas Verilog is preferred by communication market. Both languages are used for the same purpose but the syntax is different. HDL differs from software description languages such that it is not compiled and converted to instructions but it is synthesized and converted to logic equations. Since HDL describes the hardware, it doesn't "run" on device, but configures it. Therefore HDL doesn't work sequentially like instructions running on a processor but describes the hardware structure.

After synthesis is completed, design can be simulated by a simulation software to verify operation. This is done independent from FPGA device. Input signals to designed logic are simulated by a behavioral source code called test bench. Waveforms of internal signals are observed. It is a pre-check for HDL design but is not a complete verification since it doesn't include physical constraints, such as timing.

A.3.2 Constraint Determination

Physical design constraints related to timing, slice utilization, pin I/O standard and pin locations are defined in this stage. Timing constraints are related to delays between selected signals, clocks and I/O pads. These are critical and strictly defined especially for high speed applications. Logic utilization constraints are sometimes used to restrict the design to be implemented in a specific logic resource area of FPGA. It is critical for high speed memory controller applications not to

split the design among device and be close to related pins. Pin location constraints are related to I/O bank, pin and I/O standard selection for device pins. These are important to interface FPGA with other devices.

A.3.3 Place & Route

With constraints defined, the software maps the logic equations to logic resources of selected FPGA device. Additional blocks, such as memory, DCM, embedded processor etc., are also mapped. The software checks whether selected device has adequate resources (logic, memory, etc.) to implement the design. The mapped design is placed and routed. That is FPGA resources to be used are selected and their interconnection patterns are determined for routing. In this stage, defined constraints are taken into account and verified. A bit file is generated to be programmed to device to implement desired logic.

A.3.4 Embedded Processor Integration

For embedded processor integration, a tool independent from place & route tool is used to configure the processor (interfaces, operating clock, cache configuration etc.) and compile the source code running on the processor. A hex file is generated by complier which includes instructions that are defined in the instruction set of processor. The processor is added to logic design as a block with defined ports and synthesized with HDL code. In logic design, processor needs a clock and reset connection for proper operation. Processor connections to internal logic and FPGA pin are defined in HDL code. After place & route is completed, hex file is appended to bit file and a single bit file is obtained. This bit file is programmed to device by a programmer that commonly uses JTAG interface. On runtime, processor reads instructions from an instruction memory defined during configuration. Instruction memory can be distributed memory in FPGA or any external memory. The process is similar for both hard and soft processor cores.

A.3.5 On-chip Debug

After programming is completed, design is verified using on-chip debugging tool. In order to allow debugging, a sampler core should be integrated to design and connected to selected signals in place & route stage. Sampling depth, selected signals and sampling clock are configured before place & route. After bit file is generated and programmed to device user defines a triggering condition to trigger sampler to start sampling and take defined amount of samples. Samples are written to distributed memory of FPGA. Debug software reads samples from memory through JTAG interface and shows waveforms of selected signals. The overall FPGA design flow is illustrated in Figure A-4.

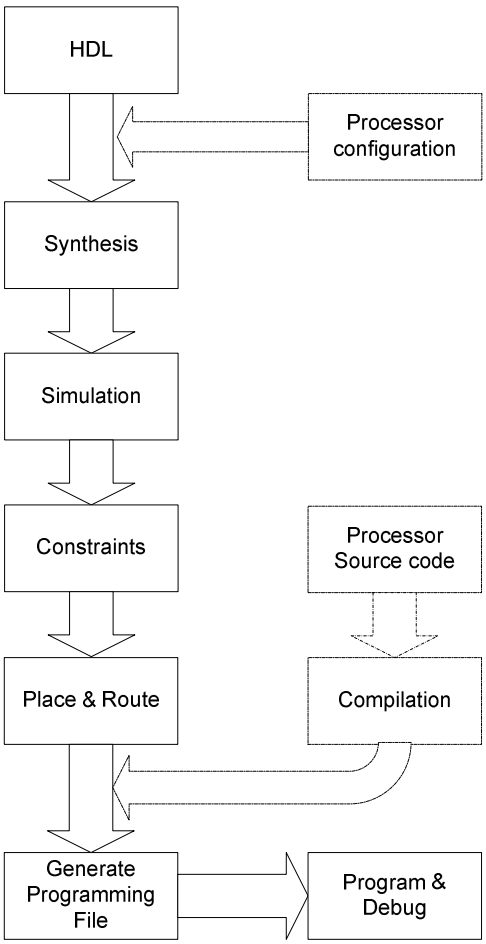

Figure A-4. FPGA design flow

APPENDIX B

HARDWARE DESIGN CONSIDERATIONS

In this appendix, hardware design considerations will be introduced.

B.1 Spartan-3 Development Board

Development board has several components used for different purposes [38]:

- Xilinx XC3S2000-FG676 Spartan-3 FPGA
- 2x16 character LCD
- 128x64 OSRAM OLED graphical display
- 32 MB DDR SDRAM
- 16 MB Flash memory
- 2 MB SRAM
- PS2 keyboard and mouse ports
- 8-position DIP switch, push-buttons and LEDs
- 140-pin I/O expansion connectors

Board has interfaces for [38]:

- RS-232
- 10/100 Ethernet
- USB 2.0
- GPIO

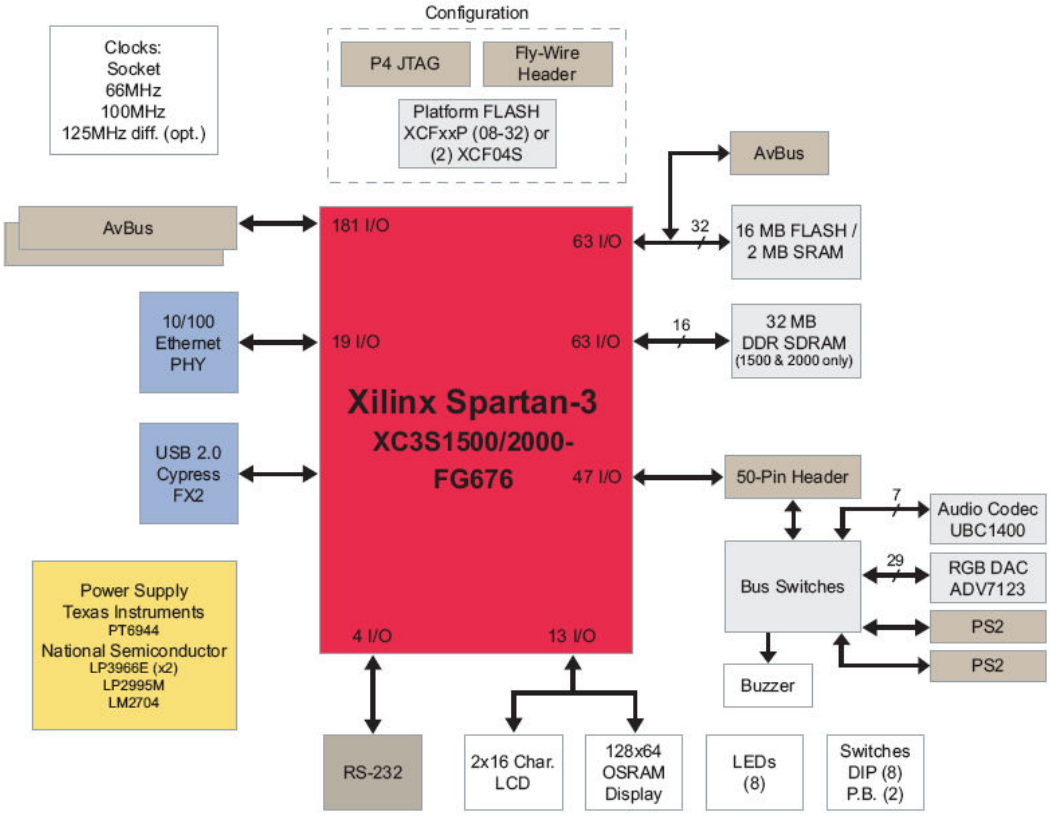

Figure B-1. Interfaces of FPGA on board [38]

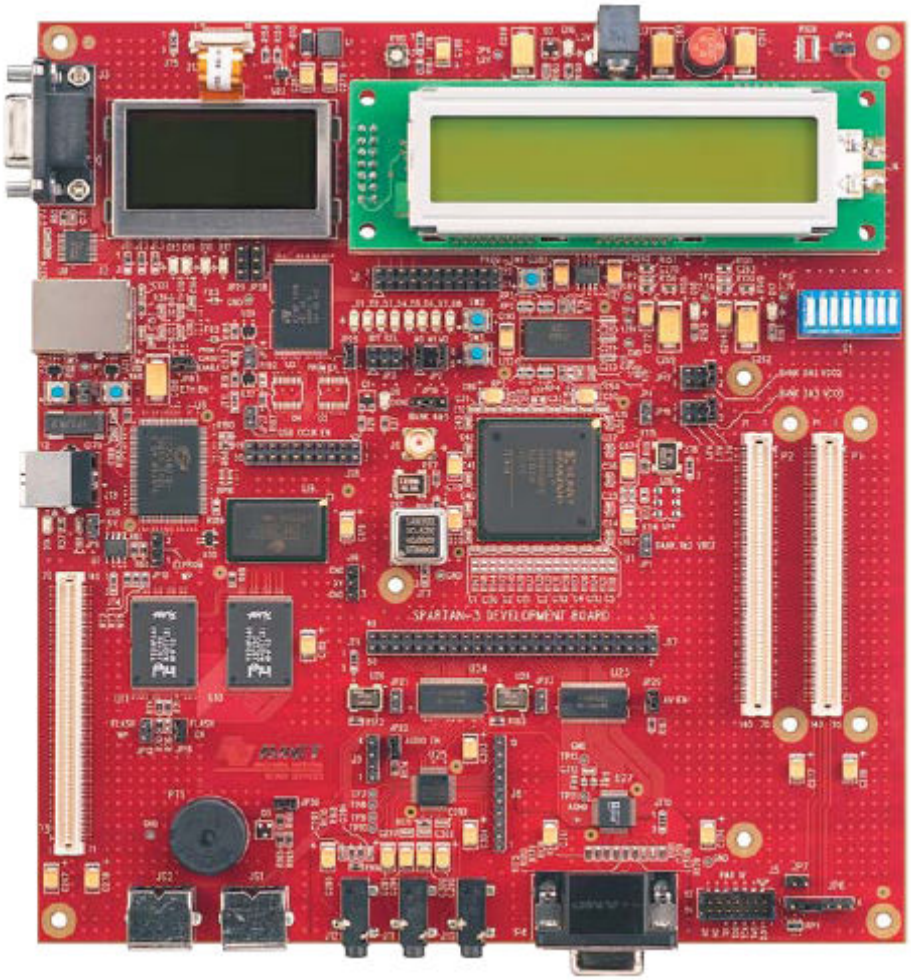

Figure B-2. XC3S2000 development board [38]

B.2 FPGA Design

Xilinx ISE® is used as the FPGA design environment including XST® for synthesis, EDK® for embedded processor development, iMPACT® for programming and ChipScope® for on-chip debugging. Building blocks of design are coded using VHDL language and combined in hierarchical structure. Memory blocks, hardware multipliers, clock management units and embedded processor are generated and inserted to FPGA design as black boxes.

Memory blocks are generated using Xilinx Core Generator® tool. The configuration of dual-port memories used for inverse transform & quantization hardware is illustrated in Figure B-3. Port data width, memory size and read/write properties are configured.

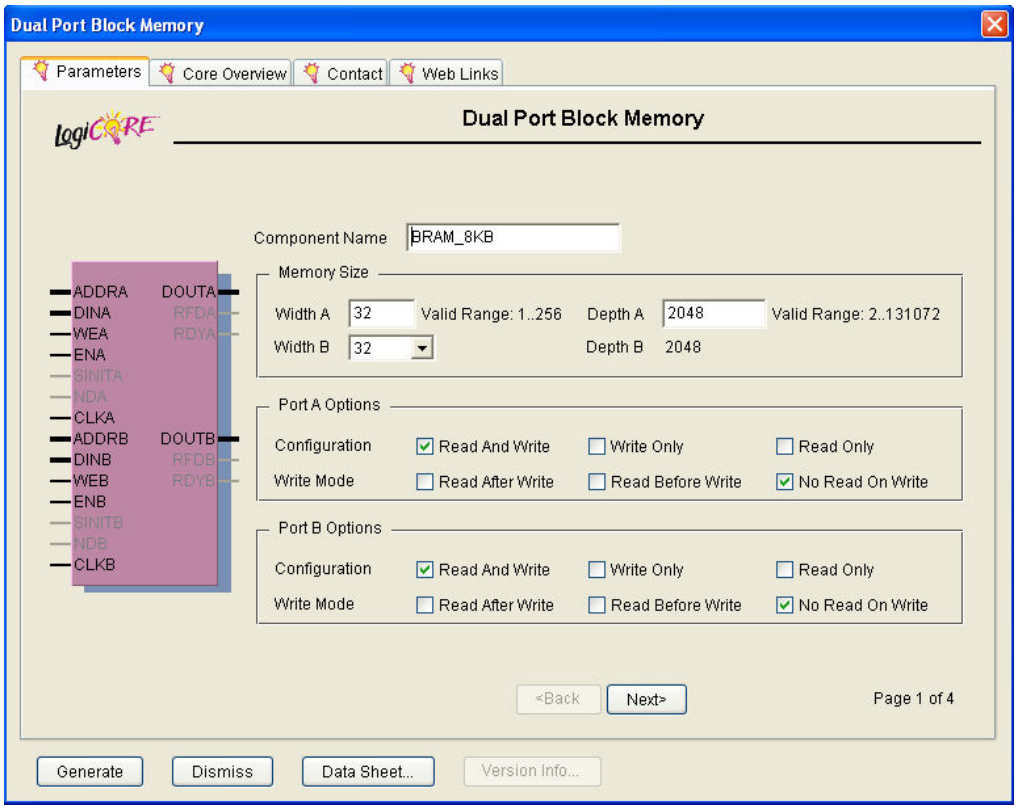

Figure B-3. Memory configuration

Hardware multipliers are limited resources inside FPGA devices. They are inserted to design by instantiating in VHDL code. Overall design is synthesized in ISE and pin location constraints are added for place & route. After place & route a BIT file is generated.

B.3 Embedded Processor Design

Embedded processor in design is constructed using MicroBlaze® processor IP for XILINX FPGAs. The processor is clocked using 40MHz oscillator on board. The processor is constructed with following features:

- Core clock frequency: 40 MHz
- Local memory bus (LMB) for instruction and data access
- Processor Local Bus (PLB) for peripheral controllers:
    - 32-bit BRAM memory controllers (x2) (*xps_bram_if_ctrl*)
    - 32-bit external memory (SRAM) controller (*xps_mch_emc*)
    - UART for serial communication (*xps_uartlite*)
    - General purpose I/O with 32 configurable pins (*xps_gpio*)

Embedded processor is implemented using Xilinx EDK® (Embedded development kit) tool. Designed processor is inserted to hardware design. The software running on embedded processor is developed in Xilinx SDK® environment. Software is coded in C language and runs in standalone mode. In other words there isn't any operating system running on processor. Obtained HEX file is added to BIT file generated by VHDL synthesis tool and a final BIT file is generated to be programmed to device. Internal structure of designed embedded processor is illustrated in Figure B-4.

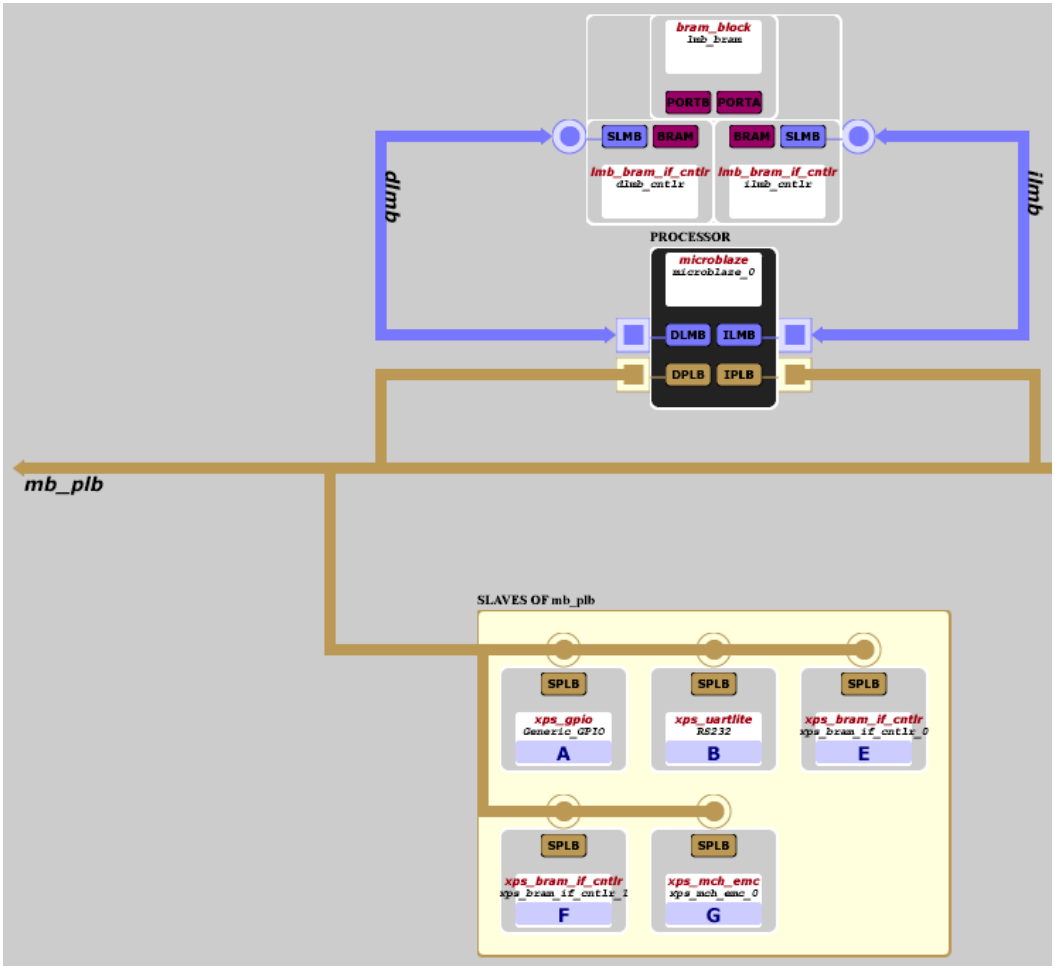

Figure B-4. Internal structure of embedded processor

B.4 Serial Communication

Serial communication between board and MATLAB® software is conducted using RS-232 port of PC and 9-pin DB9 connector on board. These are connected using a cross cable with TX (transmit), RX (receive) and GND (ground) connections. The settings of communication software developed in MATLAB® are:

- Baud rate : 115200 kbps
- Data width : 8 bits
- Parity : Odd
- Flow control : None
- Input buffer size : 8 KB
- Output buffer size : 8 KB

The software uses serial object in MATLAB® for communication. UART of embedded processor is used for serial communication. Baud rate and data parity settings of embedded processor UART are the same with MATLAB® software.

B.5 Programming Device

FPGA is programmed using the JTAG interface on board. Platform cable USB programmer is used to connect to board through USB port of PC and JTAG connector of board. Programmer is illustrated in Figure B-5. Xilinx iMPACT® tool is used to access to device and program.

Figure B-5. FPGA programmer [36]