DESIGN OF H.264/AVC COMPATIBLE INTRA-FRAME VIDEO ENCODER ON FPGA PROGRAMMABLE LOGIC DEVICES

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

BY

ÖMER GÜNAY

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

SEPTEMBER 2014

Approval of the thesis:

DESIGN OF H.264/AVC COMPATIBLE INTRA-FRAME VIDEO ENCODER ON FPGA PROGRAMMABLE LOGIC DEVICES

submitted by **ÖMER GÜNAY** 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. Gönül Turhan Sayan _______________
Head of Department, **Electrical and Electronics Engineering**

Assist. Prof. Dr. Fatih Kamışlı _______________
Supervisor, **Electrical and Electronics Eng. Dept., METU**

Examining Committee Members:

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

Assist. Prof. Dr. Fatih Kamışlı _______________
Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Dr. Cüneyt Bazlamaççı _______________
Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Dr. Çağatay Candan _______________
Electrical and Electronics Engineering Dept., METU

İsmail Özsaraç, M.Sc. _______________
Electronics Design Dept., ASELSAN

Date: _______________

ABSTRACT

DESIGN OF H.264/AVC COMPATIBLE INTRA-FRAME VIDEO ENCODER ON FPGA PROGRAMMABLE LOGIC DEVICES

Günay, Ömer

M.S., Department of Electrical and Electronics Engineering
Supervisor: Assist. Prof. Dr. Fatih Kamışlı

September 2014, 131 pages

Video compression is a technique used to reduce the amount of data in a video to limit the amount of storage space and bandwidth it requires. H.264/AVC is a widely used video compression standard developed together by the ISO (International Organization for Standardization) Moving Picture Experts Group (MPEG) and the ITU (International Telecommunication Union) Video Coding Experts Group (VCEG). H.264/AVC offers an extended range of algorithms for coding digital video to achieve superior compression efficiency with respect to previous standards, which increases computational complexity of H.264/AVC encoders and decoders.

In this thesis, an H.264/AVC compatible intra-frame video encoder is designed and implemented on FPGA devices. First, a reference encoder which includes encoding algorithms such as intra prediction, intra mode selection, transform, quantization and entropy coding, are implemented and tested in MATLAB environment. Then, the reference encoder is coded in VHDL language and tested using the Mentor Graphics Modelsim HDL simulation tool. Next, the overall FPGA implementation is tested by putting the H.264 coded bitstream into transport stream packets, streaming with UDP over Ethernet and decoding with VLC Player software on a PC. All video resolutions and frame rates defined in H.264 standard are supported by the implemented encoder.

Keywords: Video Coding, H.264, FPGA, Intra Prediction, Integer Transform, Quantization, Exponential-Golomb Coding, CAVLC

ÖZ

FPGA PROGRAMLANABİLİR ENTEGRELERİ ÜZERİNDE H.264/AVC UYUMLU INTRA-ÇERÇEVE VİDEO KODLAYICI TASARIMI

Günay, Ömer

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Yard. Doç. Dr. Fatih Kamışlı

Eylül 2014, 131 sayfa

Video sıkıştırma, bir video içerisindeki veri miktarını azaltarak o videonun saklanması için gerekli depolama alanını veya iletimi için gerekli bant genişliğini sınırlandırmak için kullanılan bir tekniktir. H.264/AVC Uluslararası Standartlar Teşkilatı (ISO) bünyesindeki Hareketli Görüntü Uzmanları Grubu (MPEG) ve Uluslararası Telekomünikasyon Birliği (ITU) bünyesindeki Video Kodlama Uzmanları Grubu tarafından ortaklaşa geliştirilmiş, geniş çapta kullanım alanına sahip bir video sıkıştırma standardıdır. H.264/AVC önceden oluşturulmuş standartlarla karşılaştırıldığında daha gelişmiş algoritmalar içerir ve bu standartlara göre videoyu daha etkin bir şekilde sıkıştırabilir. Bununla birlikte, artan işlem miktarı H.264 uyumlu kodlayıcı ve çözücü tasarımlarını daha zor bir hale getirmiştir.

Bu çalışmada, FPGA entegreleri üzerinde H.264/AVC uyumlu intra-çerçeve video kodlayıcı tasarlanmıştır. İlk etapta, intra kestirim, intra mod seçim, dönüşüm, niceleme ve entropi kodlama gibi kodlayıcı algoritmaları MATLAB ortamında oluşturulmuş ve test edilmiştir. Daha sonra MATLAB ortamında oluşturulmuş bütün kodlar, VHDL donanım tanımlama dilinde yazılmıştır ve VHDL kodları Mentor

Graphics firmasına ait Modelsim simülasyon yazılımı kullanılarak test edilmiştir. Donanım testleri esnasında kodlanmış bit katarı ilk önce TS (Transport Stream) paketi haline getirilmiş, daha sonra da UDP protokolünde Ethernet üzerinden aktarılmıştır. En son aşamada ise, H.264 kodlanmış bit katarı VLC Player yazılımı kullanılarak başarılı bir şekilde çözümlenmiştir. Tasarlanan kodlayıcı H.264 standardı tarafından desteklenen bütün seviyeleri desteklemektedir.

Anahtar Kelimeler: Video Kodlama, H.264, FPGA, Intra Kestirim, Tamsayı Dönüşüm, Niceleme, Exponential-Golomb Kodlama, CAVLC

To My Mother and Father…

ACKNOWLEDGMENTS

Firstly, I would like to thank to my supervisor, Assist. Prof. Dr. Fatih Kamışlı, for his support throughout my master study.

I would like to thank to Yüksel Serdar, Hüseyin Atik, Engin Çağlav and İsmail Özsaraç in ASELSAN for their support and help.

I would like to thank to İsmail Özsaraç for his valuable discussion, experience and help during FPGA implementation process.

I would like to thank to İsmail Özsaraç and Erkan Yavuz for his support to implement the test setup.

I would like to thank Ömer Lütfi Nuzumlalı for his assistance during thesis writing.

I also would like to thank to Aselsan Inc., Middle East Technical University and Ministry of Science, Industry and Technology for supporting the project under SANTEZ program.

I would like to thank my father Zübeyir Günay, my mother Necla Günay and my brother Bülent Günay for their infinite support, they are the people who make me as the person I currently am.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
AVC	Advanced Video Coding
BP	Baseline Profile
CABAC	Context Adaptive Binary Arithmetic Coding
CAVLC	Context Adaptive Variable Length Coding
CBP	Coded Block Pattern
CIF	Common Intermediate Format
DCT	Discrete Cosine Transform
DDR	Double Data Rate
DVD	Digital Versatile Disk
FIFO	First In First Out
FPGA	Field Programmable Gate Array
HD	High Definition
HDL	Hardware Description Language
HP	High Profile
HSMC	High Speed Mezzanine Card
IDR	Instanteneous Decoder Refresh
ISO	International Organization for Standardization
ITU	International Telecommunication Union
JM	Joint Model
JPEG	Joint Photographic Experts Group
MB	Macroblock
MP	Main Profile
MPEG	Moving Picture Experts Group
MSE	Mean Square Error
NALU	Network Abstraction Layer Unit
PC	Personal Computer

PHY	Physical
PPS	Picture Parameter Set
PSNR	Peak Signal-to-Noise Ratio
RAM	Random Access Memory
RDO	Rate Distortion Optimization
ROM	Read Only Memory
QCIF	Quarter Common Intermediate Format
QP	Quantization Parameter
SAD	Sum of Absolute Difference
SATD	Sum of Absolute Transformed Difference
SDRAM	Synchronous Dynamic Random Access Memory
SSD	Sum of Squared Difference
VCEG	Video Coding Experts Group
VCL	Video Coding Layer
VHDL	Very High Speed Integrated Circuits Hardware Description Language
VLC	VideoLAN Client
TS	Transport Stream
UDP	User Datagram Protocol

CHAPTER 1

INTRODUCTION

H.264, also known as MPEG-4 part 10 or Advanced Video Coding (AVC), is a video compression standard, which is a successor to previous H.263 and MPEG-2 standards [1]. It is currently one of the most commonly used formats for the recording, compression, and distribution of video content. H.264/AVC is used in a wide range of applications including internet video streaming, digital cinema applications, Blu-ray and HD DVD, and television broadcasting [2]. H.264/AVC was developed by the ITU-T Video Coding Experts Group (VCEG) together with ISO/IEC Moving Picture Experts Group (MPEG) and ratified as an international standard in 2003. H.264 aims an average bit rate reduction of 50% given fixed fidelity compared to previous standards.

In H.264/AVC, more complicated algorithms are used to achieve superior compression with respect to previous standards. Power consumption, logic usage, design cost and flexibility are some of the design criterions that should be considered while designing an H.264/AVC codec. Depending on the performance requirements of the applications, various platforms have been used, such as general purpose (GP) processors, multimedia co-processors, ASICs and FPGAs. Encoders on general purpose processors have been developed for comparison and development. General purpose processors however are not able to meet the constraints of real-time video encoding. Multimedia co-processors have focused on smaller frame sizes, generally CIF and below with high power consumptions. So they are not a suitable solution for portable applications. Due to the parallel processing architecture of ASICs and

FPGAs, they are more advantageous in terms of video encoding applications. However, the high cost of custom silicon makes ASIC solutions economically feasible only in high volume applications. Therefore, in lower-volume type of applications, FPGAs are typically preferred.

In this thesis, an H.264/AVC compatible intra-frame video encoder is designed and implemented on FPGA programmable devices. First, a reference encoder which includes encoding algorithms such as intra prediction, intra mode selection, transform, quantization and entropy coding, are implemented and tested in MATLAB environment. Then, the reference encoder is coded in VHDL language and tested using the Mentor Graphics Modelsim HDL simulation tool. Next, the overall FPGA implementation is tested by putting the H.264 coded bitstream into transport stream packets, streaming with UDP over Ethernet and decoding with VLC Player software on a PC.

The remainder of this thesis is organized as follows: Chapter 2 provides a basic overview of video encoding and H.264/AVC video coding standard. In Chapter 3, an overview of designed H.264 encoder and some design criteria about encoder design are given. Intra prediction and its FPGA implementation are studied in Chapter 4. Chapter 5 explains the implementation and theory of residual coding processes, i.e. transform and quantization. Entropy coding tools, including Exp-Golomb (Exponential-Golomb) coding and CAVLC (Context Adaptive Variable Length Coding) are studied in Chapter 6. The results are given in Chapter 7. Finally, Chapter 8 concludes the paper.

CHAPTER 2

AN OVERVIEW OF H.264/AVC

As shown in Figure 2.1, an H.264/AVC codec is composed of two main parts: video encoder and video decoder. The encoder part performs prediction, transforming and encoding operations to generate a compressed bitstream. In H.264, while predicting the current blocks, previously coded pixels are used. For this reason, in an encoder inverse quantization, inverse transformation and reconstruction operations are also applied to quantized transform coefficients in order to produce the previously coded pixels. The video decoder part performs the complementary operations of decoding, inverse transforming and reconstruction to generate a decoded video sequence. The compressed H.264/AVC bitstream can be transmitted or stored in different mediums such as the internet or DVD disks. Because H.264/AVC is a lossy compression standard, in general, there will be some differences between the original video and the decoded one.

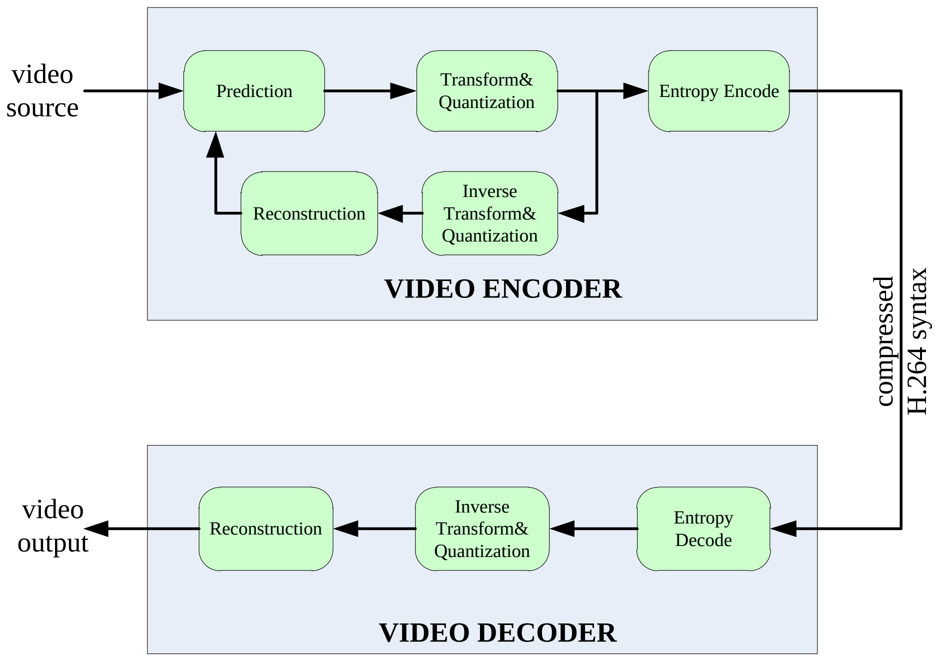


Figure 2.1 Main Parts of an H.264/AVC Codec

In this chapter, after defining some basic concepts about digital video and video coding, H.264/AVC encoder data flow, H.264/AVC decoder data flow and H.264/AVC syntax will be explained.

2.1 Digital Video Basics

2.1.1 Digital Video

Video is the combination of still pictures that are displayed at a high rate to give the impression that objects in pictures are moving. Frame rate is the number of still pictures per unit of time. Presently, the movie industry uses 24 frame per second (fps), while the TV industry 25 (in PAL and SECAM systems) and 30 fps (in NTSC system). Frame resolution specifies the number of pixels used to represent each frame (1920x1080, 640x512 etc.). Pixel depth specifies the number of bits that are used to represent each pixel (8 bits per pixel, 14 bits per pixel).

2.1.2 Progressive and Interlaced Scan

As shown in Figure 2.2, in progressive scan the horizontal lines are scanned successively. In the interlaced scan, each frame is scanned in two fields and each field contains half the number of lines in a frame.

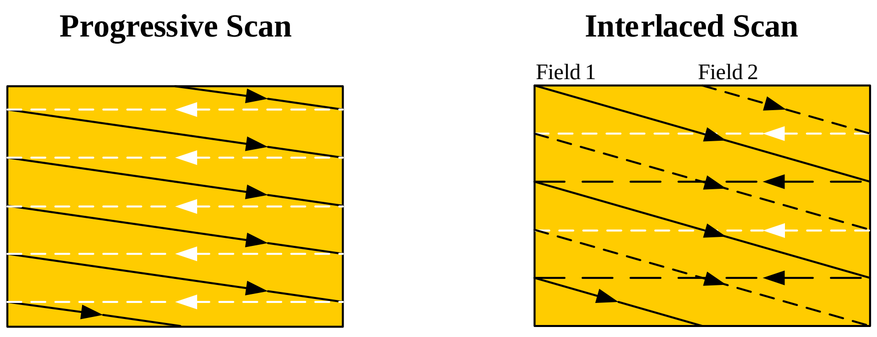


Figure 2.2 Progressive and Interlaced Scan

Interlaced scan increases temporal resolution. However, it also decreases the vertical resolution. So it comes up with a trade-off between temporal and vertical resolution. It may allow us more detailed images to be created than would otherwise be possible within a given amount of bandwidth. But it may also lead to interlacing artifacts especially in high frequency regions.

2.1.3 Video Format

In Table 2.1, commonly used video formats are listed. The choice of frame resolution depends on the application. For example, SQCIF or QCIF are appropriate for mobile multimedia applications; CIF and QCIF are popular for videoconferencing applications; 4CIF is appropriate for standard-definition television and DVD-video; HD is appropriate for high-definition television.

Table 2.1 Various Video Formats

Format	Resolution
Sub-QCIF	128x96
Quarter CIF	176x144
CIF	352x288
4CIF	704x576
720p	1280x720
HD	1920x1080
UHD	3840x2160

2.1.4 Color Spaces

Color is described by the luminance and chrominance attributes of light. Luminance refers to the perceived brightness of the light, which is proportional to the total energy in the visible band. Chrominance describes the perceived color tone of a light, which depends on the wavelength composition of the light. Human visual system is more sensitive to luminance changes than chrominance. For this reason, representing luminance and chrominance components separately is often more efficient. RGB and YCbCr are commonly used color spaces.

In RGB representation, red, green, and blue light are added together in various ways to reproduce a broad array of colors. All three components are equally important. RGB format is commonly used when displaying and storing an image.

In the YCbCr representation, Y is the luminance component and Cb, Cr are the chroma components. Y, Cb and Cr components of the image shown in Figure 2.3 are represented in Figure 2.4, Figure 2.5 and Figure 2.6, respectively. As shown from these figures, luminance component contains more information than chroma components.

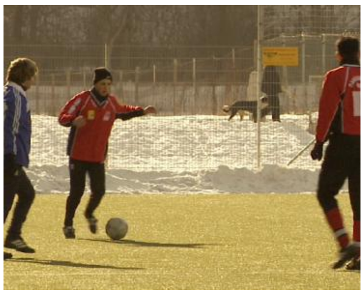


Figure 2.3 Original Frame

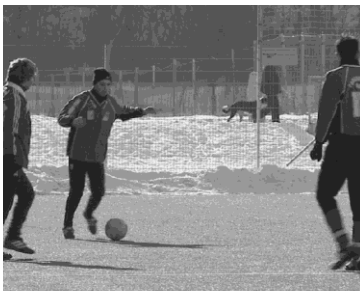


Figure 2.4 Y Component of Picture in Figure 2.3

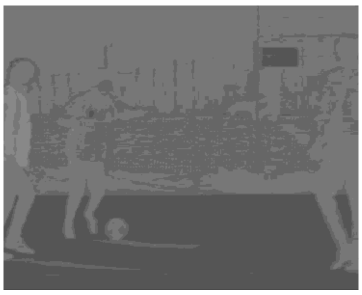


Figure 2.5 Cb Component of Picture in Figure 2.3

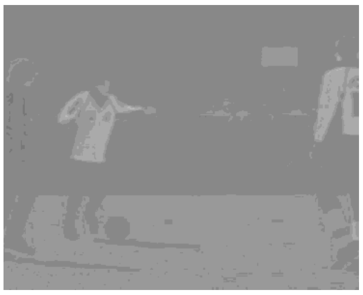


Figure 2.6 Cr Component of Picture in Figure 2.3

2.1.5 Chroma Sampling Formats

As mentioned in section 2.1.4, luma component of a frame carries more information than chroma components. For this reason, representing chroma components with less data usually may give better results in some video compression applications. This is achieved as follows: Instead of using one Cb and one Cr pair for each Y component, same Cb and Cr pairs are used for more than one Y component.

In Figure 2.7, three sampling formats that are supported by H.264/AVC are shown. In 4:4:4 sampling format, one Cb and one Cr pair is used for each luma component. In 4:2:2 sampling format, chroma components are sampled by two in the horizontal axis which means the same Cb and Cr components are used for each two horizontally neighbor luma components. In 4:2:0 sampling format, which is commonly used format, chroma components are sampled by two both in the horizontal and vertical directions.

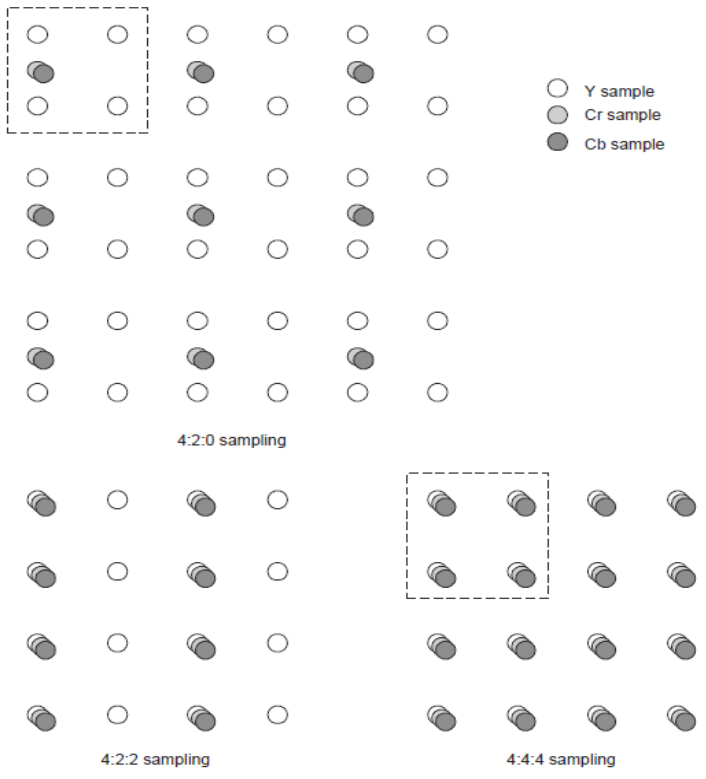


Figure 2.7 Chroma Sampling Formats

2.1.6 Image Buffering

There are two common memory structures for an image: Planar and Interleaved. In planar images, all of the samples are stored consecutively, and they are consecutive in memory as well (YYY…CbCbCb…CrCrCr…). In interleaved images, the samples are interleaved with each other in memory (YCbCrYCbCrYCbCr …). In Figure 2.8 and Figure 2.9, interleaved and planar images are shown respectively.

Y	Cb	Cr	Y	Cb	Cr	Y	Cb	Cr	
Y	Cb	Cr	Y	Cb	Cr	Y	Cb	Cr	
Y	Cb	Cr	Y	Cb	Cr	Y	Cb	Cr	

Figure 2.8 Interleaved Memory Structure

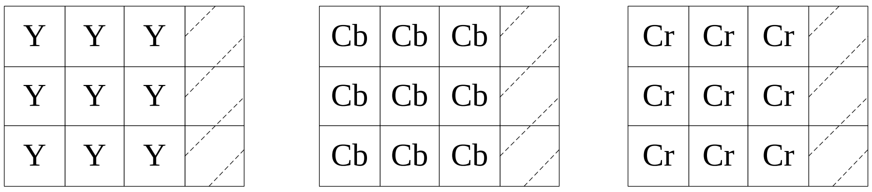


Figure 2.9 Planar Memory Structure

2.2 H.264/AVC VIDEO CODING

2.2.1 H264/AVC Profiles and Levels

The H.264/AVC standard document does not specify how to encode a digital video. It only defines syntax for compressed video and a method for decoding this syntax. An encoder may choose any tools defined in the standard. However, a decoder must implement a given set of tools and be able to process a given amount of data. These are defined as profile and level of a decoder in H.264/AVC standard.

Each H.264 profile defines a subset of tools and targets specific classes of applications. It places limits on the algorithmic capabilities required of an H.264/AVC decoder. The standard defines 21 profiles. But these 21 profiles are extended or reduced versions of the baseline profile (BP), main profile (MP) and high profile (HP). In general, BP is used for low-cost applications, MP is used for standard-definition digital TV broadcasts and HP is used for broadcast and disc storage. In Table 2.2, several H.264 profiles are compared (see [2] for more information).

Table 2.2 Some Tools Used In Several H.264 Profiles

Feature	CBP	BP	XP	MP	ProHiP	HiP	Hi10P	Hi422P	Hi444PP
Bit depth (per sample)	8	8	8	8	8	8	8 to 10	8 to 10	8 to 14
Chroma formats	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0/4:2:2	4:2:0/4:2:2/4:4:4
Interlaced coding (PicAFF, MBAFF)	No	No	Yes	Yes	No	Yes	Yes	Yes	Yes
B slices	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CABAC entropy coding	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
4:0:0 (Monochrome)	No	No	No	No	Yes	Yes	Yes	Yes	Yes
8×8 vs. 4×4 transform adaptivity	No	No	No	No	Yes	Yes	Yes	Yes	Yes
Separate C_b and C_r QP control	No	No	No	No	Yes	Yes	Yes	Yes	Yes
Separate color plane coding	No	No	No	No	No	No	No	No	Yes
Predictive lossless coding	No	No	No	No	No	No	No	No	Yes

H.264/AVC levels define the maximum data processing rate of a decoder. It puts constraints on some video parameters such as the maximum frame rate and the maximum frame size of a video. In Table 2.3, the minimum required decoder levels are listed for some different video formats at various frame rates.

Profile and level parameters are sufficient to define all the capabilities of a decoder. They also give information about the decoder complexity. From BP (Baseline Profile) to HP (High Profile) and from level 1 to 5, a decoder complexity and capability increase.

Table 2.3 Minimum Decoder Levels for Some Video Formats

Format (luma resolution)	Max frames per second	Level
QCIF (176x144)	15	1, 1b
	30	1.1
CIF (352x288)	15	1.2
	30	1.3, 2
525 SD (720x480)	30	3
625 SD (720x576)	25	3
720p HD (1280x720)	30	3.1
1080p HD (1920x1080)	30	4, 4.1
	60	4.2
4Kx2K (4096x2048)	30	5.1

2.2.2 H.264/AVC Encoder Path

H.264/AVC is a block based video compression standard. Video is processed block by block. The main block size in H.264/AVC is 16x16 which is also called as macroblock (MB). After encoding one macroblock, the next macroblock is processed until all the MBs in a frame are processed in that manner.

In Figure 2.10, the detailed block diagram of an H.264/AVC encoder is given. An H.264/AVC encoder performs prediction, motion estimation and compensation, transformation, quantization, entropy encoding operations to compress video blocks. It also generates H.264/AVC compatible bitstream from the entropy coded video blocks and entropy coded video control parameters.

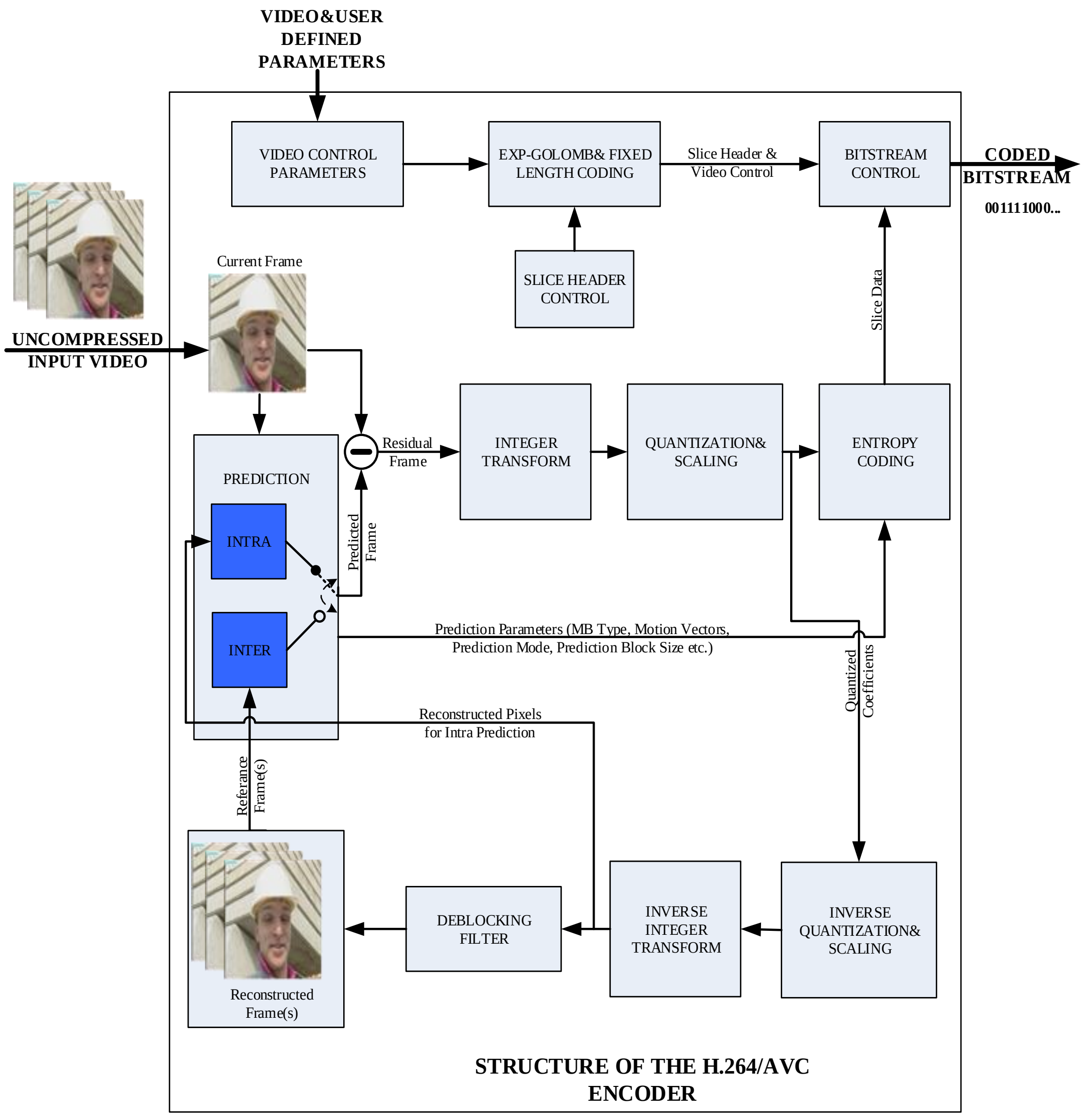


Figure 2.10 H.264 Encoder Structure

2.2.2.1 Prediction

In H.264/AVC, a prediction is created for every macroblock by using the previously coded pixels. The aim of this operation is to construct a prediction block as close as possible to the original block and send the difference (error or residual) between these blocks instead of the original block. Error between the original block and the prediction block directly affects the compression performance. If the error is small, that means residual block contains less information, the bitrate to transmit the error will be less. So the compression efficiency increases.

H.264/AVC supports two types of prediction methods: Intra prediction and Inter prediction. Intra prediction method uses the previously coded data in the current frame (Figure 2.11); on the other hand, inter prediction method uses the previously coded data in other frame(s) (Figure 2.12).

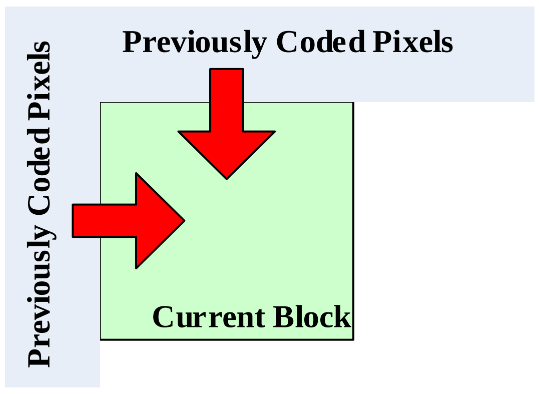


Figure 2.11 Intra Prediction

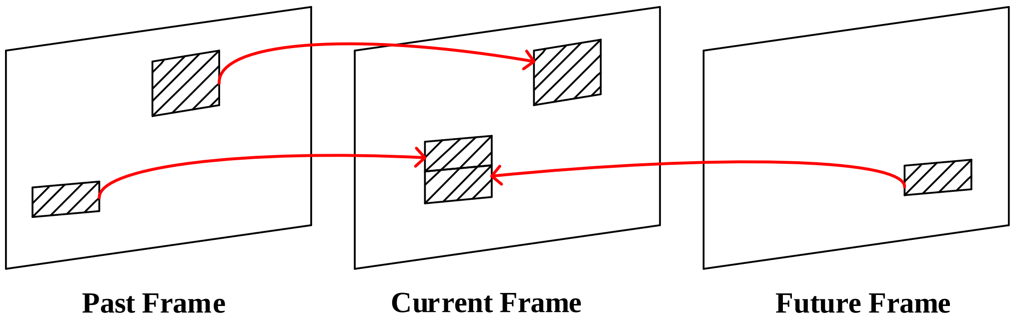


Figure 2.12 Inter Prediction

In intra prediction method, there are three choices of block sizes for luma components and there is single choice of block size for chroma components. Block sizes for luma components are 16x16, 8x8 and 4x4 (8x8 block size is only used in high profiles) and for chroma 8x8. There are 9 different prediction modes for each 4x4 and 8x8 luma blocks; 4 for a 16x16 luma block and 4 for chroma blocks.

An efficient encoder should try to decide the optimum prediction size and prediction mode before constructing the bitstream. Smaller block sizes commonly provide better prediction, but require more bits for signaling the prediction modes. The details of intra prediction modes and an efficient mode selection algorithm are represented in Chapter 4.

As mentioned earlier, in contrast to intra prediction, inter prediction uses blocks from different frame(s) other than the current frame while constructing the prediction block. In H.264/AVC, several inter prediction block sizes are allowed from 16x16 to 4x4 (Figure 2.13). A macroblock can be divided into two 16x8 blocks or two 8x16 or four 8x8 blocks which are called as macroblock partitions. Further, an 8x8 macroblock partition can be divided into two 4x8 blocks or two 8x4 blocks or four 4x4 blocks which are called as sub-macroblock partitions. A macroblock can be predicted using macroblock partitions from different frames, however, sub-macroblock partitions of a macroblock partition must be in the same frame.

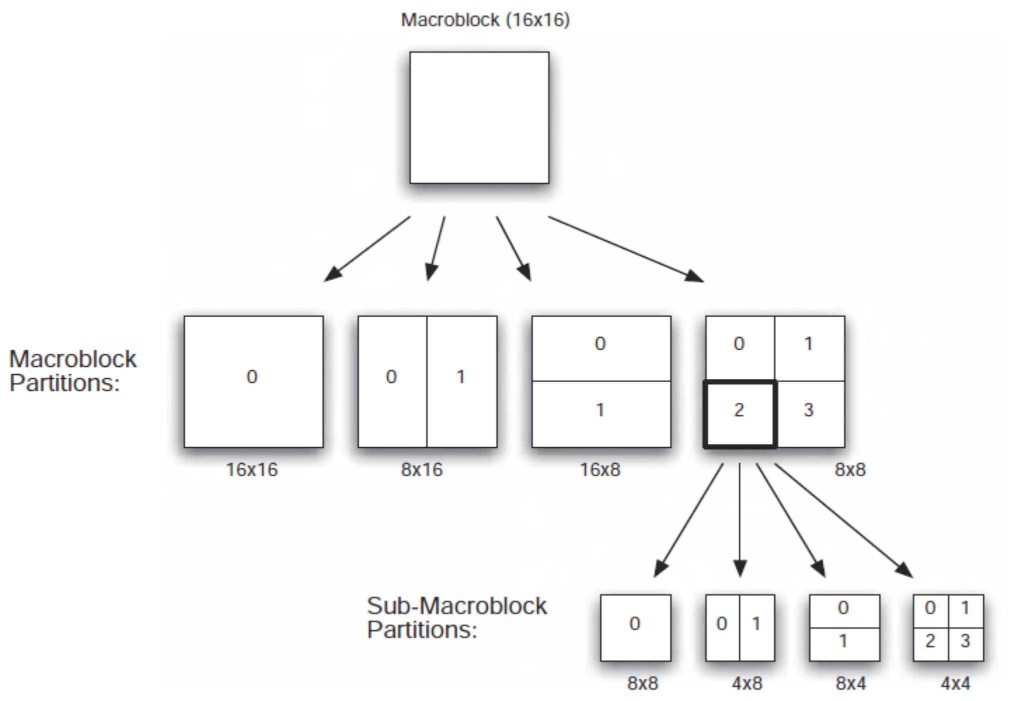


Figure 2.13 Inter Prediction Block Sizes

If a macroblock is inter predicted, the reference frame index or indexes and motion vector or vectors must be signaled to the decoder side to properly construct the decoded picture.

Some operations, such as reference picture interpolation and loop filtering, may be applied before inter prediction to increase the prediction performance. By doing the interpolation of the reference pictures, extra pixels are added into the reference frame and motion estimation can be achieved at the 1/4 pixel resolution for luma components and 1/8 pixel resolution for chroma components. Also an in-loop deblocking filter [3] reduces the blocking artifacts.

2.2.2.2 Transform & Inverse Transform

As done in the former standards, transform operation is applied after the prediction operation to code the prediction error signal. Because of the characteristics of an image, the correlation between the pixels is commonly high in the horizontal and vertical directions. There is typically also correlation left in the prediction error signal. Transform coding reduces the spatial redundancy of the prediction error signal. Former standards such as JPEG, MPEG-2 Video and MPEG-4 Visual applied a two dimensional Discrete Cosine Transform (DCT) [4] of size 8x8.

In H.264/AVC, different types of integer transforms are used to minimize the computational complexity and to avoid encoder/decoder mismatch. Equation 2.1 shows the general transformation equation, where **A** is the transform matrix, **X** is the residual block and **Y** is the transformation result. The core transform in H.264/AVC is 4x4 or 8x8 integer transform. 8x8 transform is only used in High profiles.

$$\mathbf{Y} = \mathbf{A}\mathbf{X}\mathbf{A}^{\mathbf{T}} \qquad (2.1)$$

As mentioned earlier, the size and type of the transform matrix **A** varies. In Figure 2.14, transform matrixes used in H.264/AVC are shown. A_1 and A_4 are the 4x4 and 8x8 core transforms matrices, respectively. When the luma prediction type is Intra 16x16, a second transform (also called Hadamard transform) is applied to the DC coefficients of each 4x4 blocks after the core transform by using the A_2 matrix. A_3 matrix is used for the similar purpose as A_2, but this matrix is used for chroma components. After the core transformation of chroma blocks, all DC coefficients are collected and a second transform is applied to these DC coefficients using the A_3 matrix.

$$A_1 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \quad A_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \quad A_3 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad A_4 = \begin{bmatrix} 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 12 & 10 & 6 & 3 & -3 & -6 & -10 & -12 \\ 8 & 4 & -4 & -8 & -8 & -4 & 4 & 8 \\ 10 & -3 & -12 & -6 & 6 & 12 & 3 & -10 \\ 8 & -8 & -8 & 8 & 8 & -8 & -8 & 8 \\ 6 & -12 & 3 & 10 & -10 & -3 & 12 & -6 \\ 4 & -8 & 8 & -4 & -4 & 8 & -8 & 4 \\ 3 & -6 & 10 & -12 & 12 & -10 & 6 & -3 \end{bmatrix}$$

Figure 2.14 Transform Matrixes Used in H.264/AVC

The inverse integer transform operation shown in Equation 2.2 is similar to the integer transform operation. Here **W** is the inverse transform matrix, **X** is the matrix obtained after the inverse quantization process and **Z** is the result of the inverse transformation.

$$\mathbf{Z} = \mathbf{W}^{\mathbf{T}}\mathbf{X}\mathbf{W} \qquad (2.2)$$

In Chapter 5, more details about the transform & inverse transform operations are given.

2.2.2.3 Quantization & Inverse Quantization

After transform operation, transform coefficients are quantized in order to reduce the precision of the transform coefficients according to a quantization parameter. The basic forward quantizer operation is shown in equation (2.3), where X_{ij} is a transform coefficient, Y_{ij} is the quantized output and Q_{step} is the quantization step size.

$$Y_{ij} = \text{round}(X_{ij}/Q_{step}) \qquad (2.3)$$

The quantization is the only part of the H.264/AVC that adds intentional errors into coding systems. The reason to do is to increase the compression performance with a reasonable distortion. If the quantization step size increases, more quantized coefficients will be zero which means less data to represent. This leads to keeping only a few coefficients for efficient representation and results in more distortion. It is important to note that quantization process directly controls the quality and compression ratio for applications. In H.264/AVC 52 different QP values, from 0 to 51, are supported. An encoder can control the QP parameter to control the trade-off between compression ratio and distortion.

The inverse quantization does the inverse operation of the quantization process. The basic inverse quantizer operation (or rescale) is shown in equation (2.4), where Y_{ij} is a quantized coefficient, Z_{ij} is the inverse quantization output and Q_{step} is the quantization step size.

$$Z_{ij} = Y_{ij}.Q_{step} \tag{2.4}$$

In Chapter 5, more details about the quantization and inverse quantization operations are given.

2.2.2.4 Entropy Coding

In Figure 2.15, two 4x4 residual blocks, their transforms and quantization results are given (used H.264/AVC JM reference software of version 18.4 and QP is set to 10). As seen from these results, after the quantization operation most of the quantized coefficients (especially high frequency components) become zero and occurrence of some values are more probable than others. Actually we do not need to send all these zero-valued coefficients one by one and represent all the symbols with the same length of bits.

$$\begin{matrix} 0 & 0 & 0 & 0 \\ -1 & -3 & -2 & -1 \\ -2 & -2 & -3 & -2 \\ -1 & 0 & 0 & -2 \end{matrix} \xrightarrow{\textbf{Transform}} \begin{matrix} -19 & 2 & 1 & 1 \\ 8 & -6 & 8 & 2 \\ 13 & 2 & -7 & 1 \\ -1 & 2 & -1 & -9 \end{matrix} \xrightarrow{\textbf{Quantization}} \begin{matrix} -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

$$\begin{matrix} -1 & -1 & 0 & -1 \\ 1 & -3 & -5 & -1 \\ -1 & -4 & -7 & -3 \\ -2 & -2 & -3 & -1 \end{matrix} \xrightarrow{\text{Transform}} \begin{matrix} -34 & 11 & 16 & -7 \\ 17 & -1 & -5 & 12 \\ 12 & -15 & -14 & 5 \\ -9 & 2 & -5 & 1 \end{matrix} \xrightarrow{\textbf{Quantization}} \begin{matrix} -4 & 1 & 2 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & -1 & -2 & 0 \\ -1 & 0 & 0 & 0 \end{matrix}$$

Figure 2.15 Transformation and Quantization Results of Two Residual Blocks

Entropy coding is a lossless data compression technique. Entropy coding algorithms try to assign shortest codes to the most commonly occurred symbols at the input in order to produce smaller bitstream.

H.264 specifies several methods for coding of symbols. These are fixed length code, Exponential-Golomb (Exp-Golomb) variable length code, CAVLC (Context-Adaptive Variable Length Coding) and CABAC (Context-Adaptive Binary Arithmetic Coding).

While entropy coding quantized transform coefficients, CABAC or CAVLC techniques are used. They have major improvements in terms of coding efficiency compared to the techniques used in prior video coding standards. Both methods of H.264/AVC offer a high degree of adaptation to the underlying source, even though at a different complexity-compression trade-off.

CAVLC is commonly used in baseline profile and simpler than CABAC. However, CABAC algorithm can represent the same video data with about 10% fewer bits when compared to CAVLC [5].

In Chapter 6, more details about the entropy coding techniques including Exp-Golomb and CAVLC are given.

2.2.2.5 Bitstream Control

Bitstream Control block is responsible from the construction of H.264/AVC coded bitstream properly. It manages the data flow, takes coded symbols from different blocks and combines them with a correct H.264/AVC syntax as will be described in section 2.3.

2.2.3 H.264/AVC Decoder Path

An H.264/AVC decoder constructs the video from the H.264/AVC coded bitstream. As shown in Figure 2.16, firstly the coded bitstream is entropy decoded, then inverse quantized and inverse transformed, and finally reconstruction process is applied by using prediction parameters and reference pictures or pixels. The inverse quantization, inverse transform and reconstruction operations are the same as in the encoder side.

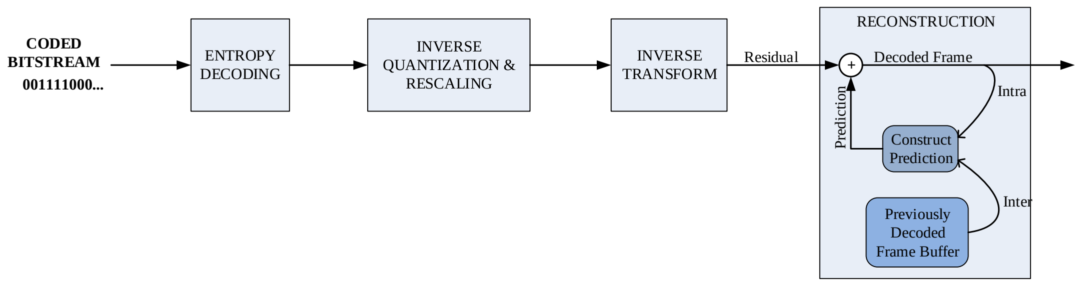


Figure 2.16 Decoder Flow

Although an H.264/AVC encoder must generate a standard compliance bitstream, a decoder must decode all the encoded bitstreams for a specific profile and level pair. So we can say that, the computational complexity of an encoder is more than a decoder but a decoder must cover all the tools defined in a specific profile and level.

2.3 H.264/AVC Syntax

H.264 syntax is defined in the H.264 standard and specifies the exact structure of an H.264-compliant video sequence. It defines the syntax elements and the construction of the coded bitstreams from these syntax elements.

Figure 2.17 shows the H.264/AVC syntax hierarchy [6]. H.264/AVC syntax consists of different Network Abstraction Layer Unit (NALU). Sequence Parameter Sets (SPS) and Picture Parameter Sets (PPS) are NAL units that signal the common control parameters about the video and video data is represented by different type of slices (IDR slice, I slice, P slice, B slice etc.).

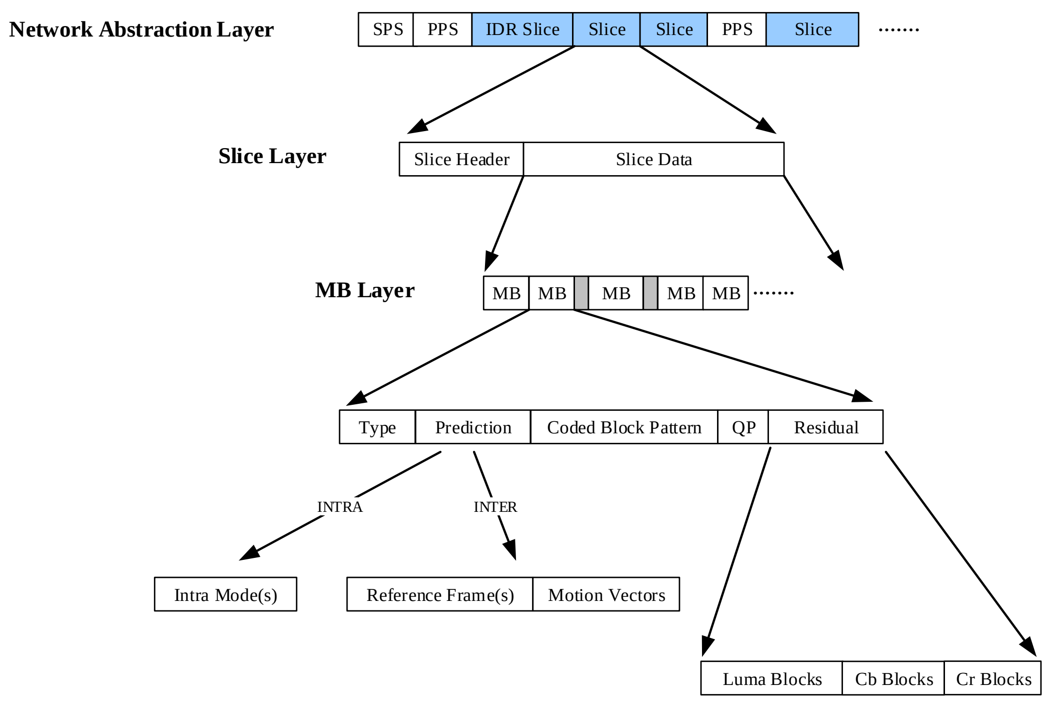


Figure 2.17 H.264/AVC Syntax

2.3.1 Video Control NAL Units

As mentioned above, SPS and PPS contain the common control parameters about the video. SPS signals the control parameters about the coded video sequences such as profile, level, video resolution and maximum frame number. In Table 2.4, one example of SPS is shown (JM 18.4 is used in trace mode). 'profile_idc' and 'level_idc' parameters shown in this table signal the profile and level values of the coded video sequence, respectively. More details about the meaning and coding of each syntax element shown in Table 2.4 can be found in the H.264 standard [1].

Table 2.4 An Example of SPS Syntax Elements

Bit Position	NALU Type	Parameter	Binary Code	Symbol
@0	SPS:	profile_idc	1000010	(66)
@8	SPS:	constrained_set0_flag	0	(0)
@9	SPS:	constrained_set1_flag	0	(0)
@10	SPS:	constrained_set2_flag	0	(0)
@11	SPS:	constrained_set3_flag	0	(0)
@12	SPS:	reserved_zero_4bits	0	(0)
@16	SPS:	level_idc	11111	(31)
@24	SPS:	seq_parameter_set_id	1	(0)
@25	SPS:	log2_max_frame_num_minus4	1	(0)
@26	SPS:	pic_order_cnt_type	1	(0)
@27	SPS:	log2_max_pic_order_cnt_lsb_minus4	1	(0)
@28	SPS:	num_ref_frames	10	(1)
@31	SPS:	gaps_in_frame_num_value_allowed_flag	0	(0)
@32	SPS:	pic_width_in_mbs_minus1	1011	(10)
@39	SPS:	pic_height_in_map_units_minus1	1001	(8)
@46	SPS:	frame_mbs_only_flag	1	(1)
@47	SPS:	direct_8x8_inference_flag	1	(1)
@48	SPS:	frame_cropping_flag	0	(0)
@49	SPS:	vui_parameters_present_flag	0	(0)

PPS signals the control parameters about the coded pictures such as entropy coding technique (CAVLC or CABAC) and the initial value of QP. In Table 2.5, one example of PPS is shown (JM 18.4 is used in trace mode). 'entropy_coding_mode_flag' set to 0 means CAVLC technique is used; otherwise, it means CABAC technique is used. ‘pic_init_qp_minus26’ syntax element signals the initial QP for luma components. In this example, setting this value to 0 means the initial QP value is 26 for luma coding. More details about the meaning and coding of each syntax element shown in Table 2.5 can be found in the H.264 standard [1].

Table 2.5 An Example of PPS Syntax Elements

Bit Position	NALU Type	Parameter	Binary Code	Symbol
@50	PPS:	pic_parameter_set_id	1	(0)
@51	PPS:	seq_parameter_set_id	1	(0)
@52	PPS:	entropy_coding_mode_flag	0	(0)
@53	PPS:	bottom_field_pic_order_in_frame_present_flag	0	(0)
@54	PPS:	num_slice_groups_minus1	1	(0)
@55	PPS:	num_ref_idx_l0_default_active_minus1	1	(0)
@56	PPS:	num_ref_idx_l1_default_active_minus1	1	(0)
@57	PPS:	weighted_pred_flag	0	(0)
@58	PPS:	weighted_bipred_idc	0	(0)
@60	PPS:	pic_init_qp_minus26	1	(0)
@61	PPS:	pic_init_qs_minus26	1	(0)
@62	PPS:	chroma_qp_index_offset	1	(0)
@63	PPS:	deblocking_filter_control_present_flag	0	(0)
@64	PPS:	constrained_intra_pred_flag	0	(0)
@65	PPS:	redundant_pic_cnt_present_flag	0	(0)

2.3.2 Video Coding Layer NAL Unit

Coded video data is represented with different type of slices (Figure 2.17). Each slice consists of a slice header and a slice data. Slice data is a series of coded macroblocks and skip macroblock indicators (gaps between macroblocks at macroblock layer shown in Figure 2.17) which signal that a macroblock contains no data. A macroblock contains syntax elements that represent the macroblock type, prediction type, coded block pattern (CBP), quantization parameter offset for a macroblock and residual data. Macroblock type shows the macroblock prediction type. When the macroblock type is I/Intra, this macroblock is predicted using only intra prediction method. If the macroblock type is P, intra prediction or inter prediction methods using one reference frame may be used. However, if the macroblock type is B, intra prediction or inter prediction methods using one or more reference frames may be used.

If the prediction type is intra, intra prediction modes are specified in prediction section (Figure 2.17); otherwise, reference pictures and motion vectors are specified in this region. CBP indicates which luma and chroma blocks contain non-zero residual coefficients, QP shows the quantization parameter offset for MBs and residual data gives the information about the quantized transform coefficients in the residual block.

CHAPTER 3

H.264/AVC ENCODER HARDWARE MODEL AND ENCODER DESIGN CRITERIA

In this chapter, general information about the designed H.264 encoder are given and some important encoder design criteria which should be exactly specified before starting an encoder designing are discussed.

3.1 H.264/AVC Encoder Hardware Model

In this thesis work, we have designed and implemented a baseline profile intra-frame H.264/AVC compliant video encoder on FPGA programmable devices. The block diagram of our hardware design is shown in Figure 3.1.

In this design, only intra prediction technique is used while predicting the macroblocks of an image. Low complexity mode selection algorihm in JM reference software of version 18.4 is implemented in order to select the prediction modes efficiently. Because this is a baseline profile encoder, 4x4 core transform and CAVLC entropy coding technique are used.

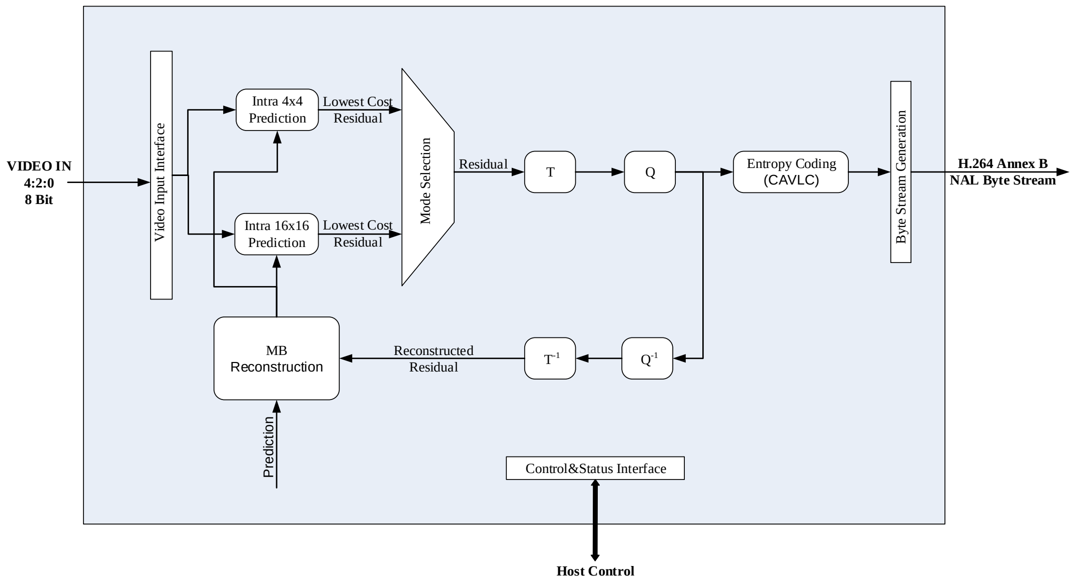


Figure 3.1 Encoder FPGA Design Top Block

3.1.1 Symbol

Figure 3.2 represents the top level signals of the designed H.264 encoder. It takes video signals (active_frame, active_line, pixel_valid and pixel_data) as inputs and outputs the H.264/AVC coded stream.

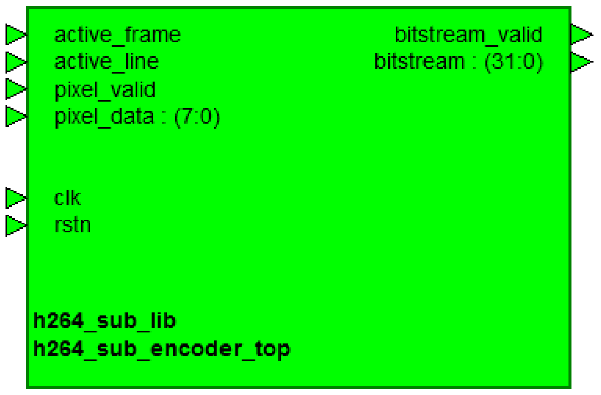


Figure 3.2 Top Level Representation of the Designed H.264 Encoder

Designed encoder supports only 4:2:0 chroma sampling format, and expects one chroma component (Cb or Cr) after two luminance components while the pixel_valid signal is high. Figure 3.3 shows encoder input signals and their orientation according to each other. As shown from this figure, after two Y components, one Cb component should come and after the next two Y components, one Cr component should come (YYCbYYCrYYCb…).

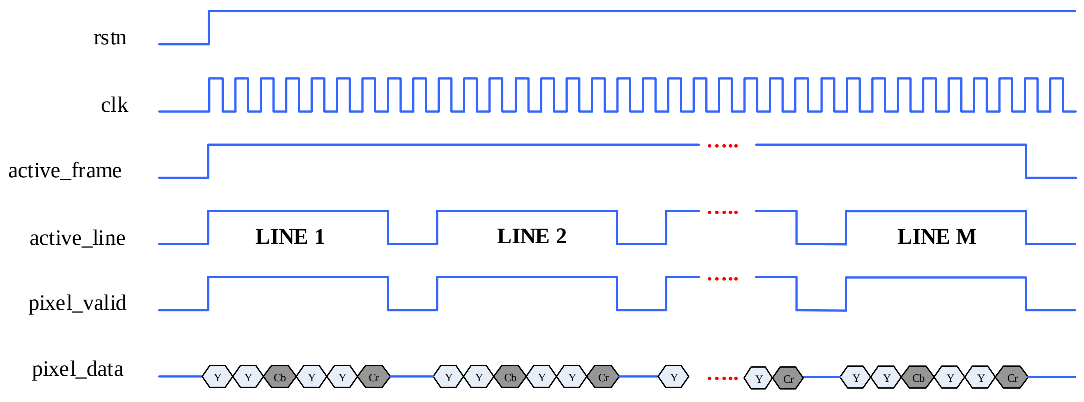


Figure 3.3 Timing Representations of the Input Signals

3.1.2 Pins Description

Implemented H.264/AVC encoder core uses only unidirectional pins. With the exception of the rstn input port signal, which is an asynchronous reset, all ports operate synchronously to the clock input clk.

Table 3.1 shows the top level signals and gives the definitions of these signals.

Table 3.1 H.264 Encoder Pins Description

Port Name	Direction	Polarity/ Bus Size	Description
System Interface			
rstn	IN	LOW	Optional global asynchronous active low reset
clk	IN	RISING	Encoder clock input
Video Data Interface			
active_frame	IN	HIGH	Video frame valid
active_line	IN	HIGH	Video line valid
pixel_valid	IN	HIGH	Video pixel data valid
pixel_data	IN	[7..0]	Video pixel data
NAL Data Interface			
bitstream_valid	OUT	HIGH	Byte stream data valid
bitstream	OUT	[31..0]	Annex B NAL byte stream data

3.2 Encoder Design Criteria

While designing an encoder, there are some parameters that should be exactly specified before starting an encoder design. Some of them are listed below.

- Compression Performance
- Encoder Delay
- Logic Usage
- Target Video Resolution and Frame Rate
- Power Consumption
- Maximum Frequency Used in FPGA Fabric

The importance of these parameters changes from one application to another. For example, if you store a movie, most probably the coding performance will be more important. Or in military applications, if you desire to transmit video from an aircraft to ground station, not only the coding performance but also the video delay may be very important. Designers should define all these requirements with respect to their applications before starting their designs.

In our encoder design, encoder delay, target video resolution and frame rate and maximum frequency used in FPGA fabric are the key design criteria. Our encoder needs one clock signal and this signal must be equal to the component frequency of the input video data. For this reason, in order to code videos at different resolution and frame rate, the only thing that must be applied is changing the input clock frequency with respect to the input video.

CHAPTER 4

H.264/AVC INTRA PREDICTION AND FPGA IMPLEMENTATION

As mentioned in Chapter 2, intra prediction uses previously coded pixels in the same frame while constructing the prediction information of the current macroblock. However, inter prediction uses previously coded MBs which are in different frame or frames and typically provides better compression. The first frame must be coded using intra frame because there is no reference frame before the first frame. However, intra frame is not only used for the first frame. There are two main reasons for this:

1) A decoder must know the reference frame(s) while constructing the decoded picture from the coded bitstream. For example, if the only first frame is intra coded (IPPPPP…), a decoder that misses the first frame would never decode the remaining part of the coded video.
2) I frames prevent error propagation. The same reference frame(s) must be used both in the encoder and decoder sides. Otherwise, the prediction blocks at the encoder and decoder will be different which may result with an unexpected error while decoding the video. In real life, the bitstream at the encoder output will be not always the same as the bitstream at the decoder input due to some noise (Figure 4.1). If an inter-coded frame uses an erroneously constructed reference frame, this previously occurred error will affect the current frame and error will be propagated with an increasing manner. In

order to prevent this error propagation, intra coded frames are commonly inserted between inter coded frames.

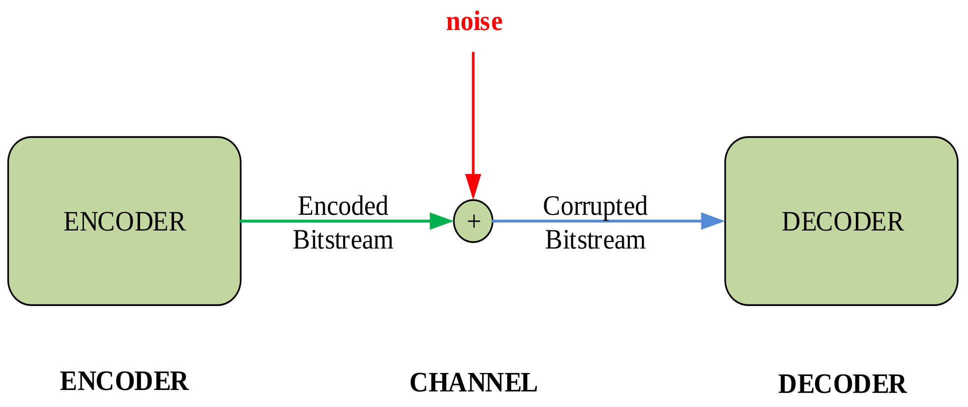


Figure 4.1 Encoder/Channel/Decoder Block Representation

An overview of the remainder of this chapter is as follows. In section 4.1, H.264/AVC intra prediction for luma and chroma components is explained. In section 4.2, an intra mode selection algorithm is presented. All intra prediction and mode selection algorithms represented in section 4.1 and 4.2 are fully implemented in MATLAB environment. In section 4.3, FPGA implementation of intra 4x4 predictions is given. In section 4.4, FPGA implementation of intra mode selection algorithm is shown. Finally, FPGA resource usage summary of intra prediction method and intra mode selection algorithm is presented in section 4.5.

4.1 H.264/AVC Intra Prediction

4.1.1 Luma Prediction

There are two types of prediction size while predicting the luma components of an image: 4x4 and 16x16. In 4x4 intra prediction, first a MB is divided into four 8x8 blocks and then each 8x8 block is divided into four 4x4 blocks as shown in Figure 4.2. The 4x4 prediction is processed block by block in the order shown in Figure 4.2. On the other hand, when 16x16 block size is used, a MB is directly predicted from its neighboring pixels without partitioning it.

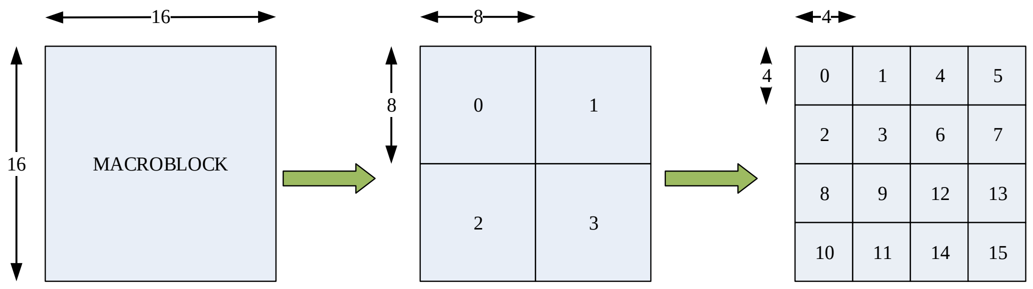


Figure 4.2 8x8 and 4x4 Scanning of a MB

There are nine intra 4x4 luma prediction modes and four intra 16x16 luma prediction modes. The 16x16 modes are generally used for homogeneous areas where there is relatively little difference, such as background, and 4x4 prediction block sizes are commonly used in areas of greater detail. Most of the time, using 4x4 prediction sizes gives better results than using 16x16 block sizes. However, signaling the prediction modes of each 4x4 block requires more bits than signaling the 16x16 prediction mode. So, in order to code an intra-frame efficiently, encoders should make a good decision while selecting the prediction mode and block size.

4.1.1.1 Luma 4x4 Intra Prediction

Figure 4.3 shows a 4x4 block (a, b, c, …, p) and neighboring pixels (A, B, C, …, M) that are used while forming the prediction of a 4x4 block.

Nine available prediction modes are shown in Figure 4.4. The arrows indicate the direction of each prediction mode.

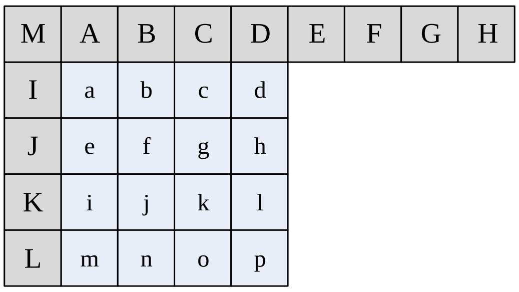


Figure 4.3 A 4x4 Block and Neighboring Pixels

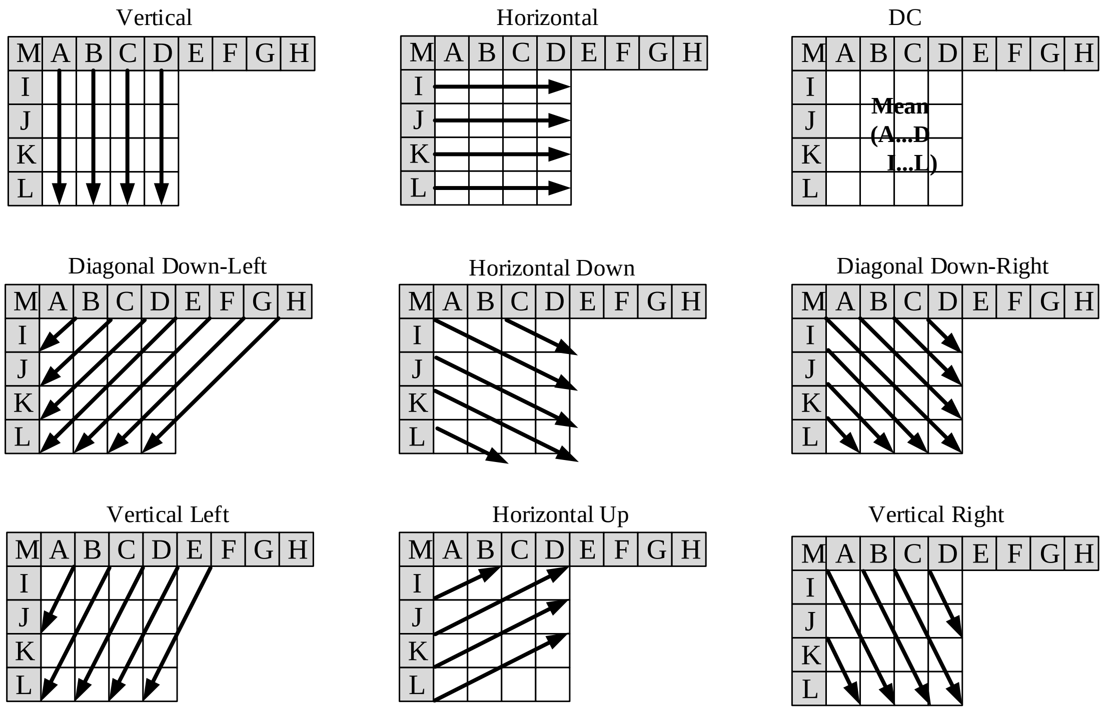


Figure 4.4 Intra 4x4 Prediction Modes

Figure 4.5 shows pixel by pixel equations for each luma 4x4 prediction modes. In this figure, pred(y,x) with 0 ≤ x, y ≤ 3 represents the prediction result. The prediction equations implement the directional copying operations shown in Figure 4.4 (>> indicates bitwise right shift). The top left, top right, bottom left and bottom right positions of a 4x4 block are denoted as pred(0,0), pred(0,3), pred(3,0) and pred(3,3), respectively.

pred(0,0) = pred(1,0) = pred(2,0) = pred(3,0) = A
pred(0,1) = pred(1,1) = pred(2,1) = pred(3,1) = B
pred(0,2) = pred(1,2) = pred(2,2) = pred(3,3) = C
pred(0,3) = pred(1,3) = pred(2,3) = pred(3,3) = D

(a) Vertical

pred(0,0) = pred(0,1) = pred(0,2) = pred(0,3) = I
pred(1,0) = pred(1,1) = pred(1,2) = pred(1,3) = J
pred(2,0) = pred(2,1) = pred(2,2) = pred(2,3) = K
pred(3,0) = pred(3,1) = pred(3,2) = pred(3,3) = L

(b) Horizontal

pred(y,x) = (A + B + C + D + I + J + K + L + 4) >> 3
(Left and upper pixels are both available)

pred(y,x) = (I + J + K + L + 2) >> 2
(Only left pixels are available)

pred(y,x) = (A + B + C + D + 2) >> 2
(Only left pixels are available)

pred(y,x) = 128
(Only left pixels are available)

(c) DC

pred(0, 0) = A + 2B + C + 2 >> 2
pred(0, 1) = B + 2C + D + 2 >> 2
pred(0, 2) = C + 2D + E + 2 >> 2
pred(0, 3) = D + 2E + F + 2 >> 2
pred(1, 0) = B + 2C + D + 2 >> 2
pred(1, 1) = C + 2D + E + 2 >> 2
pred(1, 2) = D + 2E + F + 2 >> 2
pred(1, 3) = E + 2F + G + 2 >> 2
pred(2, 0) = C + 2D + E + 2 >> 2
pred(2, 1) = D + 2E + F + 2 >> 2
pred(2, 2) = E + 2F + G + 2 >> 2
pred(2, 3) = F + 2G + H + 2 >> 2
pred(3, 0) = D + 2E + F + 2 >> 2
pred(3, 1) = E + 2F + G + 2 >> 2
pred(3, 2) = F + 2G + H + 2 >> 2
pred(3, 3) = G + 3H + 2 >> 2

(d) Diagonal Down Left

pred(0, 0) = A + 2M + I + 2 >> 2
pred(0, 1) = M + 2A + B + 2 >> 2
pred(0, 2) = A + 2B + C + 2 >> 2
pred(0, 3) = B + 2C + D + 2 >> 2
pred(1, 0) = M + 2I + J + 2 >> 2
pred(1, 1) = A + 2M + I + 2 >> 2
pred(1, 2) = M + 2A + B + 2 >> 2
pred(1, 3) = A + 2B + C + 2 >> 2
pred(2, 0) = I + 2J + K + 2 >> 2
pred(2, 1) = M + 2I + J + 2 >> 2
pred(2, 2) = A + 2M + I + 2 >> 2
pred(2, 3) = M + 2A + B + 2 >> 2
pred(3, 0) = J + 2K + L + 2 >> 2
pred(3, 1) = I + 2J + K + 2 >> 2
pred(3, 2) = M + 2I + J + 2 >> 2
pred(3, 3) = A + 2M + I + 2 >> 2

(e) Diagonal Down Right

pred(0, 0) = M + A + 1 >> 1
pred(0, 1) = A + B + 1 >> 1
pred(0, 2) = B + C + 1 >> 1
pred(0, 3) = C + D + 1 >> 1
pred(1, 0) = I + 2M + A + 2 >> 2
pred(1, 1) = M + 2A + B + 2 >> 2
pred(1, 2) = A + 2B + C + 2 >> 2
pred(1, 3) = B + 2C + D + 2 >> 2
pred(2, 0) = M + 2I + J + 2 >> 2
pred(2, 1) = M + A + 1 >> 1
pred(2, 2) = A + B + 1 >> 1
pred(2, 3) = B + C + 1 >> 1
pred(3, 0) = I + 2J + K + 2 >> 2
pred(3, 1) = I + 2M + A + 2 >> 2
pred(3, 2) = M + 2A + B + 2 >> 2
pred(3, 3) = A + 2B + C + 2 >> 2

(f) Vertical Right

pred(0, 0) = M + I + 1 >> 1
pred(0, 1) = I + 2M + A + 2 >> 2
pred(0, 2) = B + 2A + M + 2 >> 2
pred(0, 3) = C + 2B + A + 2 >> 2
pred(1, 0) = I + J + 1 >> 1
pred(1, 1) = M + 2I + J + 2 >> 2
pred(1, 2) = M + I + 1 >> 1
pred(1, 3) = I + 2M + A + 2 >> 2
pred(2, 0) = J + K + 1 >> 1
pred(2, 1) = I + 2J + K + 2 >> 2
pred(2, 2) = I + J + 1 >> 1
pred(2, 3) = M + 2I + J + 2 >> 2
pred(3, 0) = K + L + 1 >> 1
pred(3, 1) = J + 2K + L + 2 >> 2
pred(3, 2) = J + K + 1 >> 1
pred(3, 3) = I + 2J + K + 2 >> 2

(g) Horizontal Down

pred(0, 0) = A + B + 1 >> 1
pred(0, 1) = B + C + 1 >> 1
pred(0, 2) = C + D + 1 >> 1
pred(0, 3) = D + E + 1 >> 1
pred(1, 0) = A + 2B + C + 2 >> 2
pred(1, 1) = B + 2C + D + 2 >> 2
pred(1, 2) = C + 2D + E + 2 >> 2
pred(1, 3) = D + 2E + F + 2 >> 2
pred(2, 0) = B + C + 1 >> 1
pred(2, 1) = C + D + 1 >> 1
pred(2, 2) = D + E + 1 >> 1
pred(2, 3) = E + F + 1 >> 1
pred(3, 0) = B + 2C + D + 2 >> 2
pred(3, 1) = C + 2D + E + 2 >> 2
pred(3, 2) = D + 2E + F + 2 >> 2
pred(3, 3) = E + 2F + G + 2 >> 2

(h) Vertical Left

pred(0, 0) = I + J + 1 >> 1
pred(0, 1) = I + 2J + K + 2 >> 2
pred(0, 2) = J + K+ 1 >> 1
pred(0, 3) = J + 2K + L + 2 >> 2
pred(1, 0) = J + K+ 1 >> 1
pred(1, 1) = J + 2K + L + 2 >> 2
pred(1, 2) = K + L + 1 >> 1
pred(1, 3) = K + 3L + 2 >> 2
pred(2, 0) = K + L + 1 >> 1
pred(2, 1) = K + 3L + 2 >> 2
pred(2, 2) = L
pred(2, 3) = L
pred(3, 0) = L
pred(3, 1) = L
pred(3, 2) = L
pred(3, 3) = L

(i) Horizontal Up

Figure 4.5 4x4 Luma Prediction Equations

The number of possible intra 4x4 prediction modes for a block varies with respect to the block positions. For example, upper reference pixels (A to H) are not available at the upper border of an image, so one cannot use intra 4x4 prediction modes of 0, 3, 4, 5, 6 and 7 while predicting these blocks. We can divide an image into four regions with respect to the possible intra 4x4 prediction modes that can be applied during the prediction of these blocks (Figure 4.6). As seen from Figure 4.6, the DC mode (mode 2) is the only mode which is always used.

As shown from Figure 4.4, Diagonal Down-Left and Vertical Left prediction modes needs upper eight pixels (A to H) in order to form the prediction. However, while predicting the fourth 4x4 blocks of an 8x8 block (blocks 3,7,11 and 15 in Figure 4.2), the reconstructed pixels E, F, G and H are not available. This problem is solved by the H.264/AVC standard as follows: If the top right pixels (E, F, G and H) are not available, use the nearest available pixel value (pixel D) instead of these pixels.

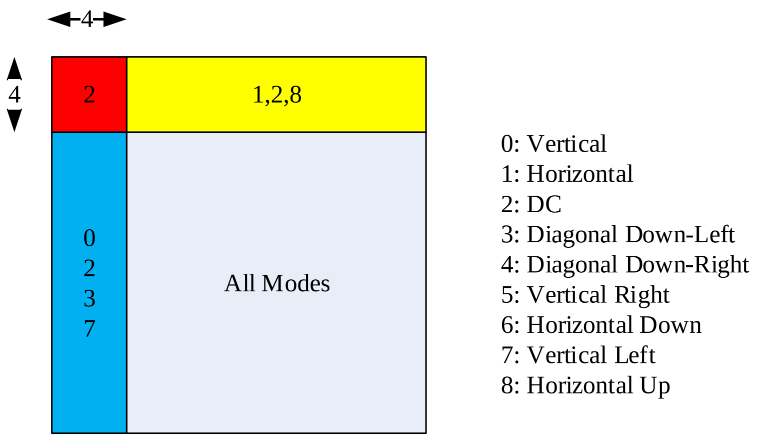


Figure 4.6 Possible 4x4 Prediction Modes at Various Block Locations inside an Image

4.1.1.2 Luma 16x16 Intra Prediction

Four intra 16x16 prediction modes are represented in Figure 4.7. The arrows indicate the direction of each prediction mode. The prediction equations that implement these intra 16x16 luma prediction modes are shown in equations 3.1 to 3.4 where (y, x) denotes the position of pixels, p represents the neighboring pixel values and $\mathrm{Clip1}_Y$ function in equation 3.4 clips the result into [0 - 255] range (actually the range is [0 - $(2^{\text{pixel depth}}-1)$]).

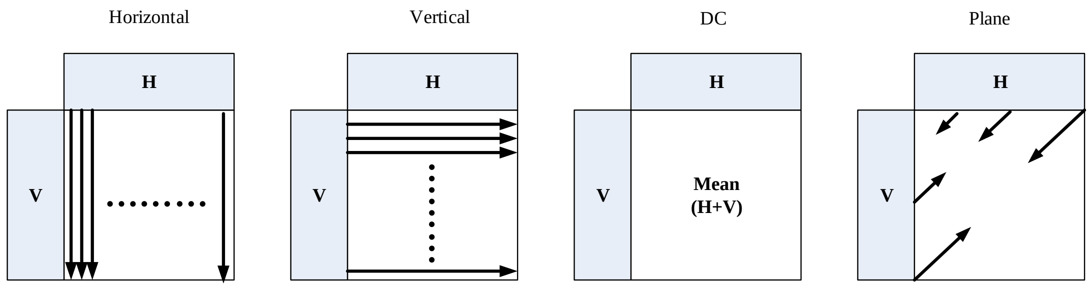


Figure 4.7 16x16 Prediction Modes

$\mathrm{pred}_{ver}(y,x) = p(-1,x)$ with $x, y = 0..15$ (3.1)

$\mathrm{pred}_{hor}(y,x) = p(y,-1)$ with $x, y = 0..15$ (3.2)

$\mathrm{pred}_{dc}(y,x) = ({}^{15}_{x'=0}\, p\ \ x', -1\ \ + \ {}^{15}_{y'=0}\, p\ \ -1, y'\ \ + 16) \gg 5$ $x, y = 0 .. 15$ (3.3a)

(If both upper and left neighbor pixels are available)

$\mathrm{pred}_{dc}(y,x) = ({}^{15}_{y'=0}\, p\ \ y', -1\ \ + 8) \gg 4$ with $x, y = 0 .. 15$ (3.3b)

(If left neighbor pixels are not available)

$\mathrm{pred}_{dc}(y,x) = ({}^{15}_{x'=0}\, p\ \ -1, x'\ \ + 8) \gg 4$ with $x, y = 0 .. 15$ (3.3c)

(If upper neighbor pixels are not available)

$\mathrm{pred}_{dc}(y,x) = 128$ with $x, y = 0 .. 15$ (3.3d)

(If both upper and left neighbor pixels are not available)

$\mathrm{pred}_{pl}(y,x) = \mathrm{Clip1}_Y((a + b * (x - 7) + c * (y - 7) + 16) >> 5)$ $x, y = 0..15$ (3.4)

where,

$a = 16 * (p(15,-1) + p(-1,15))$ (3.4a)

$$b = (5 * H + 32) >> 6 \qquad (3.4b)$$

$$c = (5 * V + 32) >> 6 \qquad (3.4c)$$

where,

$$H = (\ {}_{x'=0}^{7}(x' + 1) * (p \ \ -1, 8 + x' \ \ - p(-1, 6 - x')) \qquad (3.4d)$$

$$V = (\ {}_{y'=0}^{7}(y' + 1) * (p \ \ 8 + y', -1 \ \ - p(6 - y', -1)) \qquad (3.4e)$$

Figure 4.8 shows the possible intra 16x16 prediction modes of an image with respect to MB positions. The DC mode (mode 2) is the only mode which is always used. Horizontal and DC modes are used at the upper image boundaries, vertical and DC modes are used at the left image boundaries.

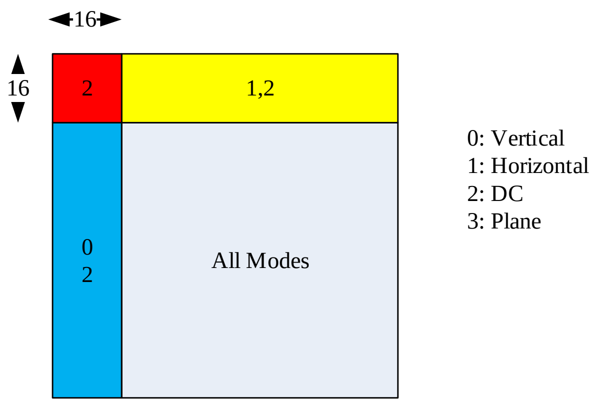


Figure 4.8 Possible 16x16 Prediction Modes at Various Block Locations inside an Image

4.1.2 Chroma Prediction

Each chroma component of a macroblock is predicted from the previously encoded chroma samples above and/or to the left of the macroblock. One prediction block is generated for each chroma component. Both chroma components must use the same prediction mode in H.264/AVC. There are four possible intra prediction modes and they are very similar to the intra 16x16 prediction modes described in section 4.1.1.2, except that the numbering of the modes are different.

Figure 4.9 shows chroma 8x8 prediction mode equations where (y, x) denotes the position of pixels, p represents the neighboring pixel values and Clip1 function clips the result into [0-255] range (actually the range is [0 - ($2^{\text{pixel depth}}$-1)]). Figure 4.10 shows possible chroma 8x8 prediction modes of an image with respect to 8x8 block positions. The DC mode is the only mode used for the prediction of all chroma blocks. Horizontal and DC modes are used at the upper image boundaries and vertical and DC modes are used at the left image boundaries.

(a) Vertical

$$
\begin{aligned}
\text{pred}(y, 0) &= p(-1, 0)\\
\text{pred}(y, 1) &= p(-1, 1)\\
\text{pred}(y, 2) &= p(-1, 2)\\
\text{pred}(y, 3) &= p(-1, 3)\\
\text{pred}(y, 4) &= p(-1, 4)\\
\text{pred}(y, 5) &= p(-1, 5)\\
\text{pred}(y, 6) &= p(-1, 6)\\
\text{pred}(y, 7) &= p(-1, 7)
\end{aligned}
$$

$$0 \le y \le 7$$

(b) Horizontal

$$
\begin{aligned}
\text{pred}(0, x) &= \text{p}(0, -1)\\
\text{pred}(1, x) &= \text{p}(1, -1)\\
\text{pred}(2, x) &= \text{p}(2, -1)\\
\text{pred}(3, x) &= \text{p}(3, -1)\\
\text{pred}(4, x) &= \text{p}(4, -1)\\
\text{pred}(5, x) &= \text{p}(5, -1)\\
\text{pred}(6, x) &= \text{p}(6, -1)\\
\text{pred}(7, x) &= \text{p}(7, -1)
\end{aligned}
$$

$$0 \le x \le 7$$

(c-1) DC
$0 \le x \le 3$ and $0 \le y \le 3$

pred(y, x) = (Σ *p(*x', -1) + Σ *p(*-1, y') + 4) >> 3
(If p(x', –1) with x' = 0..3, and p(–1, y') and y' = 0..3 are available)

pred(y, x) = (Σ *p(*x', -1) + 2) >> 2
(Else If p(x', –1) with x' = 0..3 are available and p(–1, y') and y' = 0..3 are not available)

pred(y, x) = (Σ *p(*-1, y') + 2) >> 2
(Else If p(–1, y') and y' = 0..3 are available and p(x', –1) with x' = 0..3 are not available)

pred(y, x) = 128
(Else If p(x', –1) with x' = 0..3, and p(–1, y') and y' = 0..3 are not available

(c-2) DC
$4 \le x \le 7$ and $0 \le y \le 3$

pred(y, x) = (Σ *p(*x', -1) + 2) >> 2
(If p(x', –1) with x' = 4..7 are available)

pred(y, x) = (Σ *p(*-1, y') + 2) >> 2
(Else If p(–1, y') and y' = 0..3 are available)
pred(y, x) = 128
(Else)

(c-3) DC
$0 \le x \le 3$ and $4 \le y \le 7$
pred(y, x) = (Σ *p(*-1, y') + 2) >> 2
(If p(–1, y') and y' = 4..7 are available)

pred(y, x) = (Σ *p(*x', -1) + 2) >> 2
(Else If p(x', –1) with x' = 0..3 are available)
pred(y, x) = 128
(Else)

(c-4) DC
$4 \leq x \leq 7$ and $4 \leq y \leq 7$

$$pred(y, x) = (\Sigma\, p(x', -1) + \Sigma\, p(-1, y') + 4) >> 3$$
(If p(x', –1) with x' = 4..7, and p(–1, y') and y' = 4..7 are available)

$$pred(y, x) = (\Sigma\, p(x', -1) + 2) >> 2$$
(Else If p(x', –1) with x' = 4..7 are available and p(–1, y') and y' = 4..7 are not available)

$$pred(y, x) = (\Sigma\, p(-1, y') + 2) >> 2$$
(Else If p(–1, y') and y' = 4..7 are available and p(x', –1) with x' = 4..7 are not available)

$$pred(y, x) = 128$$
(Else If p(x', –1) with x' = 4..7, and p(–1, y') and y' = 4..7 are not available)

(d) Plane

$$pred(y,x) = Clip1\,(((a + b * (x - 3) + c * (y - 3) + 16) >> 5)$$
$$a = 16 *(p(-1,7) + p(7,-1))$$
$$b = (5 * H + 32) >> 6$$
$$c = (5 * V + 32) >> 6$$

$$H = \Sigma\, (x'+1)*(\, p(-1,4 + x') + p(-1, 2- x'))$$
$$(x' = 0, 1, 2, 3)$$

$$V = \Sigma\, (y'+1)*(\, p(4 + y',-1) + p(\, 2- y',-1))$$
$$(y' = 0, 1, 2, 3)$$

Figure 4.9 8x8 Chroma Prediction Equations

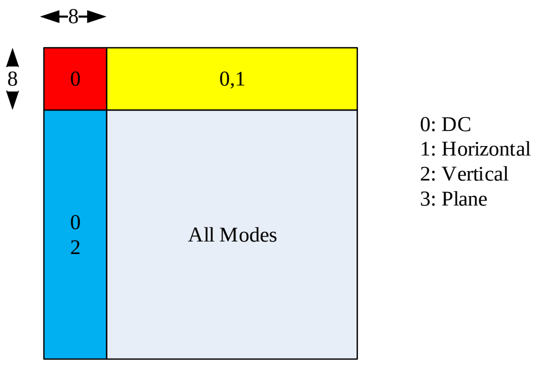


Figure 4.10 Possible Chroma Prediction Modes at Various Block Locations inside an Image

4.2 Intra Mode Selection

A video encoder aims to minimize coded bitrate and maximize decoded video quality. One way to achieve this aim is deciding the best prediction mode. Because there are many possible combinations of encoding parameters, deciding the best tradeoff between minimizing bitrate and minimizing distortion is a challenging task.

An intra-frame H.264/AVC video encoder may choose many different modes. These include:

- Luma Macroblock Intra Mode: 16x16 intra vs. 4x4 intra
- For 16x16 Intra: one of four 16x16 intra modes
- For 4x4 Intra: one of nine 4x4 intra modes for each 4x4 block in macroblock
- For Chroma: one of four intra chroma modes

The best choice of mode depends on the particular characteristics of the MB and the chosen weighting between distortion and rate. In our encoder design, a rate distortion optimization (RDO) based mode selection algorithm ([7], [8]) is implemented.

4.2.1 Rate Distortion Optimized (RDO) Mode Selection

This algorithm attempts to find a mode that minimizes the joint cost J shown in equation 3.5. The joint cost J depends on the bitrate cost R and distortion cost D. The tradeoff between distortion (D) and rate (R) is controlled by the Lagrange multiplier λ. If λ is small, encoder will give more emphasis to minimizing the distortion which results with a higher bitrate. However, a larger λ will tend to minimize R at the expense of higher distortion. Usually, these two cases are both not desired. Desired is a working point at which both the distortion and the rate are minimized together.

$$J = D + \lambda R \tag{3.5}$$

The general formula for the Lagrangian parameter λ is shown in equation 3.6. As seen from this equation, λ depends on the quantization parameter QP and selected scaling factor, LambdaWeight. Setting the LambdaWeight to 0.85 usually gives good results. However, different LambdaWeight values, such as 0.57, 0.65, 0.68 and 0.85, are used in JM reference software while coding different type of slices.

$$\lambda = \text{LambdaWeight} * 2^{(QP-12)/3} \tag{3.6}$$

Different metrics can be used to calculate the distortion. Sum of Absolute Transformed Differences (SATD), Sum of Absolute Differences (SAD) and Sum of Squared Differences (SSD) are the commonly used distortion functions. SAD, shown in equation 3.7, calculates sum of the absolute difference between pairs of samples b and b’. SSD, shown in equation 3.8, calculates the sum of the squared difference between pairs of samples b and b’ and more computationally intensive when compared to SAD.

$$D_{SAD} = \ _{x,y} |b(x, y) – b'(x, y)| \tag{3.7}$$

$$D_{SSD} = \ _{x,y} (b(x, y) – b'(x, y))^2 \tag{3.8}$$

The residual block, difference between the original and the prediction block, is usually highly correlated in spatial domain. After calculating the residual, transform operation is applied to further eliminate this redundancy. If the residual block contains more spatial redundancy, transformed and quantized block contains less number of nonzero coefficients which results with a better compression. SATD distortion metric (eq. 3.9) is developed for this purpose. The residual block is Hadamard transformed to count the spatial redundancy while calculating the mode error. T operation in equation 3.9 takes the Hadamard transform of the input block.

$$D_{SATD} = \ _{x,y} |T(b(x, y) – b'(x, y))| \tag{3.9}$$

4.2.2 Luma Mode Selection Algorithm

In this work, an RDO based mode decision algorithm is used. This algorithm is also used in H.264/AVC reference model which is also called as low complexity mode decision algorithm.

In our implementation, SATD metric is used for the calculation of distortion and LambdaWeight parameter is set to 0.85.

4.2.2.1 Intra 4x4 Mode Selection

The following 8-step algorithm is used to determine the best 4x4 intra mode for a 4x4 block.

1. For each 4x4 prediction modes, find the 4x4 residual matrices by using equation 3.10.

$$RES_{4x4} = \mathrm{orj}\ \ i,j\ \ - \mathrm{pred}(i,j)\ \ \text{, where } 0 \leq i, j \leq 3 \tag{3.10}$$

2. Find the Hadamard transform of matrix RES_{4x4}.

$$SATD = H * RES_{4x4} * H^{T} \tag{3.11}$$

$$\text{where,} \quad H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

3. Calculate distortion D by using equation 3.12.

$$D = ((\ {}_{i=1}^{4}\ \ {}_{j=1}^{4}|\mathrm{SATD}\ \ i,j\ \ |) + 1) \gg 1 \tag{3.12}$$

4. Find the most probable mode. If both upper and left blocks of the current block are available, the most probable mode is the smaller one of these blocks; otherwise the most probable mode is 2.

5. Calculate R.

```
If (prediction mode is equal to the most probable mode)
  R=1
Else
  R=4
End
```

6. Calculate λ by using equation 3.13.

$$\lambda = [(0.85 * 2^{(QP-12)/3})^{1/2} + 1] >> 1 \tag{3.13}$$

7. Use equation 3.14 to find the cost of each mode

$$J = D + \lambda R \tag{3.14}$$

8. Choose the mode with the minimum cost.

4.2.2.2 Intra 16x16 Mode Selection

The following 7-step algorithm is used to determine the best 16x16 intra mode for a 16x16 block.

1. Divide a MB into 16 4x4 blocks as shown in Figure 4.11.

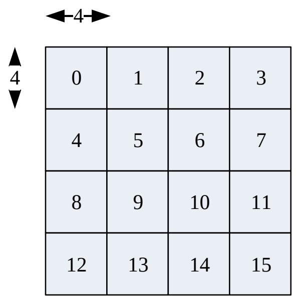


Figure 4.11 4x4 Representation of a MB

2. Find residual for each 4x4 block (RES_{16x16}) by using the equation 3.15.

$$RES_{16x16} = orj(i, j) - pred(i, j) \quad \text{, where } 0 \leq i, j \leq 3 \tag{3.15}$$

3. Find the Hadamard transform of matrix $RES_{16x16.}$

$$SATD = (H * RES_{16x16} * H^T) >> 1 \tag{3.16}$$

4. Take the DC coefficients of the transformed matrixes in step 3 and shift them logically one bit right.
5. Hadamard transform the DC coefficients one more time by applying the equation 3.16.
6. Take the absolute values of AC coefficients obtained after step 3 and DC coefficients obtained after step 5 and sum all of them. The summation result is equal to the cost.
7. Choose the mode with the minimum cost.

4.2.2.3 Mode Selection between Intra 4x4 and Intra 16x16

After finding the best 4x4 modes and the best 16x16 mode of a MB, encoder must select one of the prediction sizes. The steps for this are as follows:

1. Sum all the costs of selected intra 4x4 modes. These costs are calculated in section 4.2.2.1.
2. Add an offset of 6λ to the summation obtained in step 1. This value is the total cost of intra 4x4 prediction, J_{4x4}.
3. Total cost of intra 16x16 prediction, J_{16x16}, is the minimum cost found in section 4.2.2.2 at step 7.
4. Compare J_{4x4} and J_{16x16}. If $J_{4x4} < J_{16x16}$, chose 4x4 block size; otherwise chose 16x16 block size.

4.2.3 Chroma Mode Selection Algorithm

The following steps are applied to find the prediction mode of the chroma blocks.

1. Divide each Cb and Cr MBs into 4x4 blocks as shown in Figure 4.12.

2. For each 4x4 prediction modes, find the 4x4 residual matrixes by using equation 3.10.
3. Apply equation 3.11 to all residual matrixes in order to find the Hadamard transform of these matrixes.
4. Calculate the distortion for each 4x4 block by applying the equation 3.12.
5. Sum all the distortion results for each mode. The summation result is the total cost of the corresponding prediction mode.
6. Choose the prediction mode which results with the minimum cost after step 5.

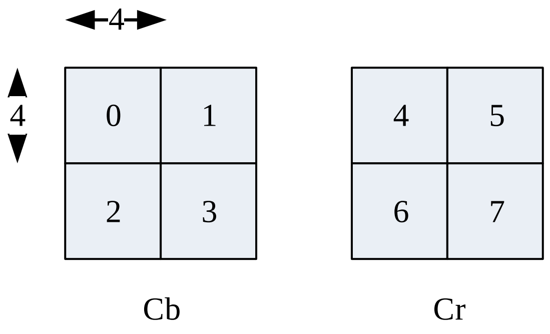


Figure 4.12 4x4 Representation of Cb and Cr Components

4.3 FPGA Implementation of 4x4 Intra Prediction

The necessary FPGA blocks and signals for the 4x4 Intra prediction method are given and explained in this section. VHDL (Very High Speed Integrated Circuit Hardware Description Language) language is used to code this block. The 4x4 intra prediction and residual calculation FPGA block diagram is represented in Figure 4.13. This diagram is composed of two main parts: “Video Flow Control” and “4x4 Residual Calculation”.

“Video Flow Control” block is responsible from the macroblock by macroblock processing of the video data. It takes video data and stores it in two different Line

RAMs. Each Line RAM (Random Access Memory) can store 16 lines of video data. In H.264/AVC, MB is the main processing unit. In order to start the coding operation, one MB data must be captured. For this reason, 16 lines of video data are stored in FPGA internal block RAMs. When 16 lines are captured from the sensor, video coding operation starts. While video data in one Line RAM is processed, the next video lines are stored in the other Line RAM. Video Flow Control block reads one MB data from the Line RAMs and passes the data to 4x4 Residual Calculation block. It also gives information if the MB is at the upper border or left border of the video frame. As explained in section 4.1.1.1, border information of a MB is necessary while predicting the MB.

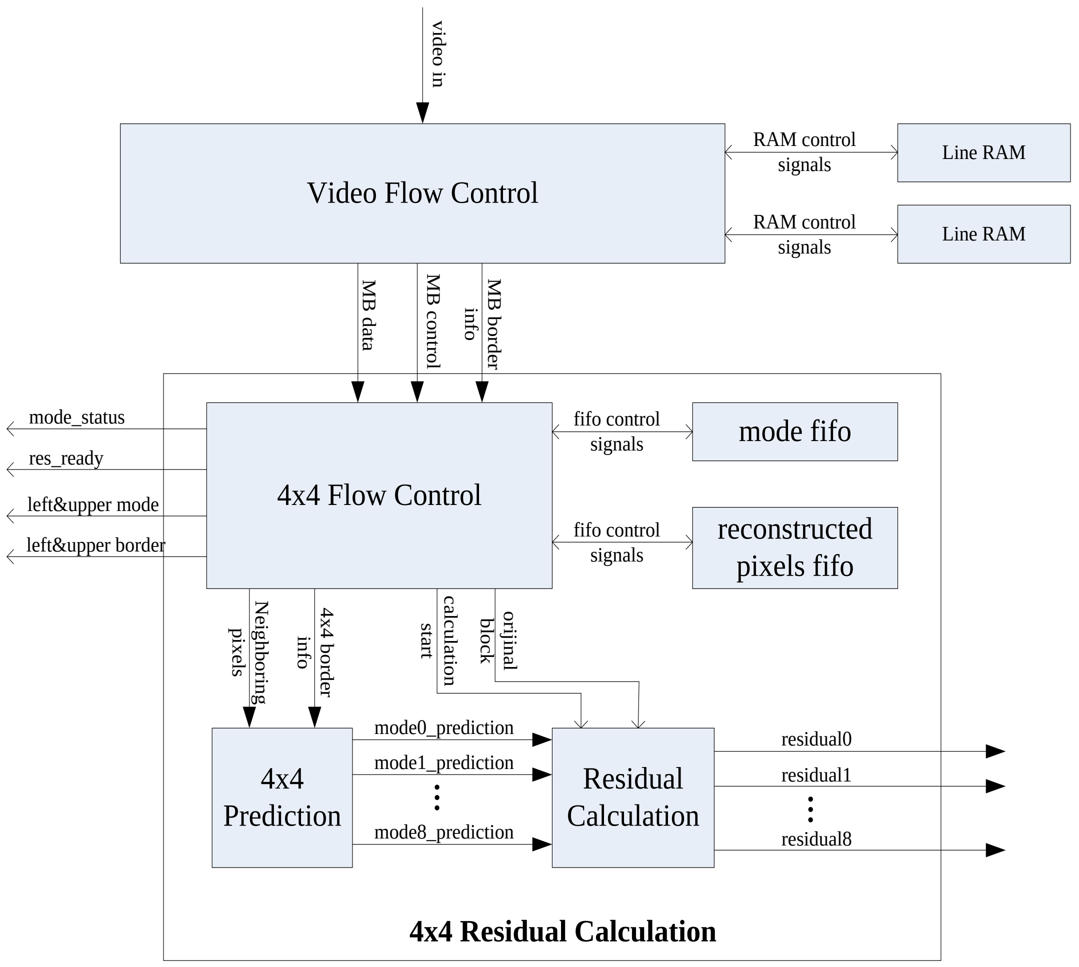


Figure 4.13 4x4 Intra Prediction and Residual Calculation

The "4x4 Residual Calculation" block calculates the residual data for each prediction mode. As shown in Figure 4.13, it is composed of five sub-blocks: "4x4 Flow Control", "4x4 Prediction", "Residual Calculation", "Mode FIFO (First In First Out)" and "Reconstructed Pixels FIFO".

The "4x4 Flow Control" block manages the whole 4x4 intra prediction system. It takes the MB data and processes it as 4x4 blocks as shown in Figure 4.2. This block sends the neighboring pixels and the border information of the current 4x4 block to the "4x4 Prediction" block in order to construct the prediction data for each mode. It also controls “Residual Calculation” block to construct the residual data by sending a start flag and original pixels of the current 4x4 block.

As mentioned earlier, because intra prediction uses neighboring pixels which belong to the previously coded left and upper blocks, the previously coded 4x4 blocks must be reconstructed in order to start the prediction operation of the current block. The previously reconstructed pixels are kept in the “Reconstructed Pixels FIFO”. The “4x4 Flow Control” block reads the reconstructed pixel values from this FIFO when they are necessary.

The “4x4 Flow Control” block drives res_ready signal to high when the residuals for the current 4x4 block is ready. It also sends mode_status, left_mode, upper_mode, left_border and upper_border signals for different purposes.

The mode_status is a 9 bit signal (Figure 4.14) and used during the mode selection operation to decide the valid prediction modes of the current block. Each bit in this signal corresponds to a specific mode. A value of logical ‘1’ states that the corresponding mode is applicable for the current block.

The left_mode and upper_mode signals are used by the mode selection block while calculating the rate information (R) which is shown in equation 3.5. They are also used together with left_border and upper_border signals while signalling the prediction mode of the current block.

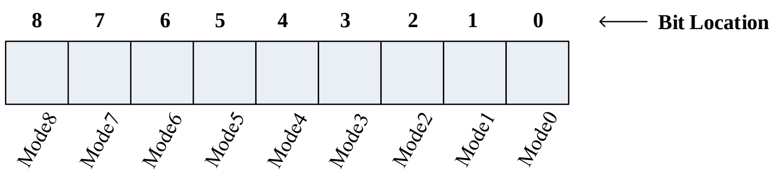


Figure 4.14 “mode_status” Signal Bits and Corresponding Modes

The “4x4 Prediction block” in Figure 4.13 calculates the prediction information of the nine different modes for the current block by using the neighboring pixels. It performs this operation in a parallel manner. All the prediction results are valid at the same clock cycle.

In intra prediction equations shown in Figure 4.5, there are some common parts and some of equations are the same. This information can be used in order to reduce the computation complexity, computation time and to save the hardware resources. Intra prediction equations are reorganized as shown in Figure 4.15 to exploit these redundancies.

While calculating the prediction information, firstly the equations shown in Figure 4.15 are performed. At the next clock cycle, the concatenation and bit shifting operations are performed. For this reason, it takes 2 clock cycles to construct the prediction results from the neighboring pixels.

```
mode0_P1_P5_P9_P13        =  A
mode0_P2_P6_P10_P14       =  B
mode0_P3_P7_P11_P15       =  C
mode0_P4_P8_P12_P16       =  D
```

(a) Vertical Prediction

```
mode1_P1_P2_P3_P4         =  I
mode1_P5_P6_P7_P8         =  J
mode1_P9_P10_P11_P12      =  K
mode1_P13_P14_P15_P16     =  L
```

(b) Horizontal Prediction

(If upper and left neighboring pixels are available)

mode2_P_ALL	=	A + B + C + D + I + J + K + L + 4

(If only upper neighboring pixels are available)

mode2_P_ALL	=	A + B + C + D + 2

(If only left neighboring pixels are available)

mode2_P_ALL	=	I + J + K + L + 2

(If neither upper and left neighboring pixels are available)

mode2_P_ALL	=	128

(c) DC Prediction

mode3_P1	=	A + 2*B + C + 2
mode3_P2_P5	=	B + 2*C + D + 2
mode3_P3_P6_P9	=	C + 2*D + E + 2
mode3_P4_P7_P10_P13	=	D + 2*E + F + 2
mode3_P8_P11_P14	=	E + 2*F + G + 2
mode3_P12_P15	=	F + 2*G + H + 2
mode3_P16	=	F + 2*H + H + 2

(d) Diagonal Down-Left

mode4_P1_P6_P11_P16	=	I + 2*M + A + 2
mode4_P2_P7_P12	=	M + 2*A + B + 2
mode4_P5_P10_P15	=	J + 2*I + M + 2
mode4_P9_P14	=	K + 2*J + I + 2
mode4_P13	=	L + 2*K + J + 2
mode4_P3_P8	=	**Same as mode3_P1**
mode4_P4	=	**Same as mode3_P2_P5**

(e) Diagonal Down-Right

mode5_P1_P10	=	M + A + 1
mode5_P2_P11	=	A + B + 1
mode5_P3_P12	=	B + C + 1
mode5_P4	=	C + D + 1
mode5_P5_P14	**=**	**Same as mode4_P1_P6_P11_P16**
mode5_P6_P15	**=**	**Same as mode4_P2_P7_P12**
mode5_P7_P16	**=**	**Same as mode3_P1**
mode5_P8	**=**	**Same as mode3_P2_P5**
mode5_P9	**=**	**Same as mode4_P5_P10_P15**
mode5_P13	**=**	**Same as mode4_P9_P14**

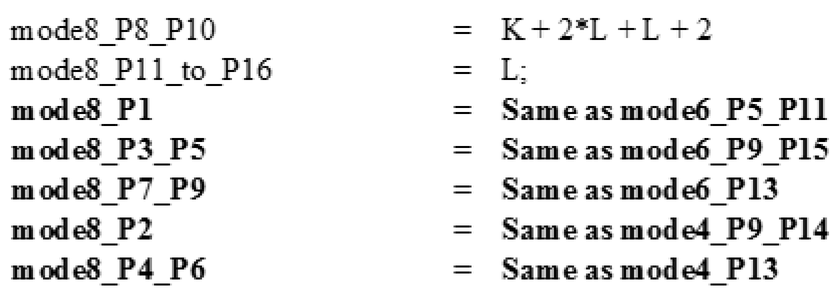
(f) Vertical Right

mode6_P1_P7	=	M + I + 1
mode6_P5_P11	=	I + J + 1
mode6_P9_P15	=	J + K + 1
mode6_P13	=	K + L + 1
mode6_P2_P8	**=**	**Same as mode4_P1_P6_P11_P16**
mode6_P6_P12	**=**	**Same as mode4_P5_P10_P15**
mode6_P10_P16	**=**	**Same as mode4_P9_P14**
mode6_P3	**=**	**Same as mode4_P2_P7_P12**
mode6_P4	**=**	**Same as mode3_P1**
mode6_P14	**=**	**Same as mode4_P13**

(g) Horizontal Down

mode7_P4_P11	=	D + E + 1
mode7_P12	=	E + F + 1
mode7_P1	**=**	**Same as mode5_P2_P11**
mode7_P2_P9	**=**	**Same as mode5_P3_P12**
mode7_P3_P10	**=**	**Same as mode5_P4**
mode7_P5	**=**	**Same as mode3_P1**
mode7_P6_P13	**=**	**Same as mode3_P2_P5**
mode7_P7_P14	**=**	**Same as mode3_P3_P6_P9**
mode7_P8_P15	**=**	**Same as mode3_P4_P7_P10_P13**
mode7_P16	**=**	**Same as mode3_P8_P11_P14**

(h) Vertical Left



mode8_P8_P10	=	K + 2*L + L + 2
mode8_P11_to_P16	=	L;
mode8_P1	=	**Same as mode6_P5_P11**
mode8_P3_P5	=	**Same as mode6_P9_P15**
mode8_P7_P9	=	**Same as mode6_P13**
mode8_P2	=	**Same as mode4_P9_P14**
mode8_P4_P6	=	**Same as mode4_P13**

(i) Horizontal Up

Figure 4.15 Similarities between Prediction Equations

The "Residual Calculation" block calculates the residual data of the nine prediction modes by using the original and predicted pixels. The detailed block diagram of this block is shown in Figure 4.16. There is a residual calculation sub-block for each mode. In each of this sub-block, there are sixteen subtraction blocks which are used to subtract the predicted pixels from the original ones. After subtractions, the pixels are concatenated and stored in a FIFO. The subtraction operation is synchronous and takes one clock cycle. The residual calculation needs 2 clock cycles: One for subtraction and one for concatenation and FIFO write operations.

The "Mode FIFO" and the "Reconstructed Pixels FIFO" are used to store the mode and reconstructed pixels information which will be used while coding the next blocks. These FIFOs are controlled by the "4x4 Flow Control" block.

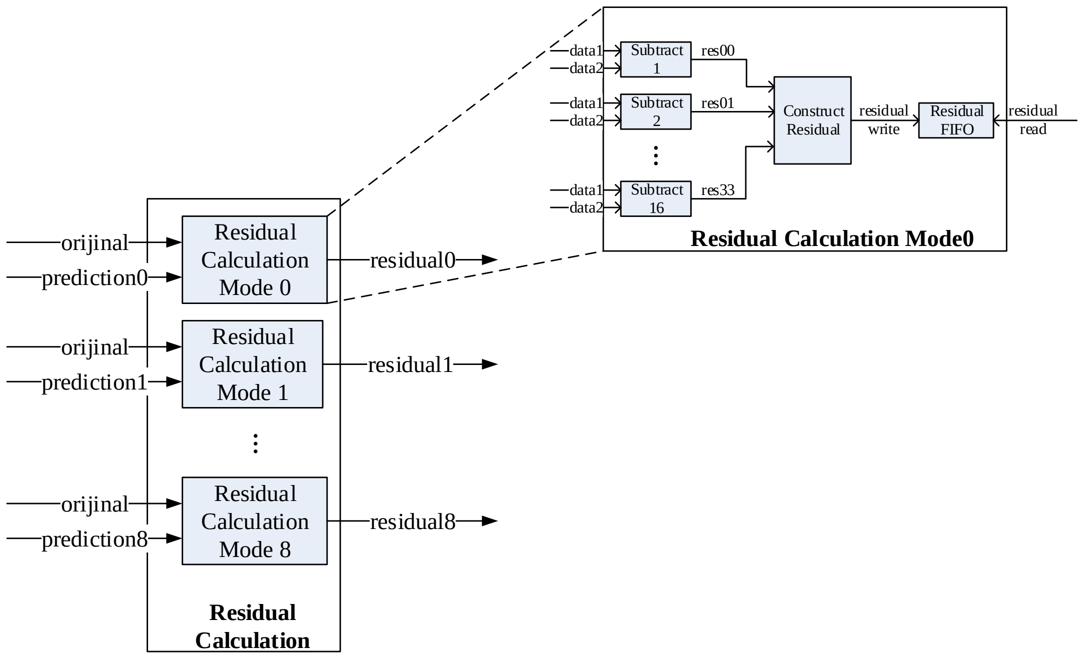


Figure 4.16 Residual Calculation Block Diagram

4.4 FPGA Implementation of 4x4 Intra Mode Selection Algorithm

FPGA implementation of intra 4x4 mode selection algorithm is explained in this section. Like intra prediction, this block is also coded by VHDL. FPGA sub-blocks of mode selection algorithm are represented in Figure 4.17.

In this design model, all of the rate, distortion and cost values (equation 3.5) of each mode are calculated in parallel. After finding the costs of each mode, nine cost values are divided into three groups to find the minimum cost. Each group takes three cost values and finds the minimum of these. Finally the three minimum costs are sorted again and the minimum of these are found. At this stage, the mode that corresponds to the minimum cost is selected as the best prediction mode. The residual and mode information associated with this mode is sent to the other blocks.

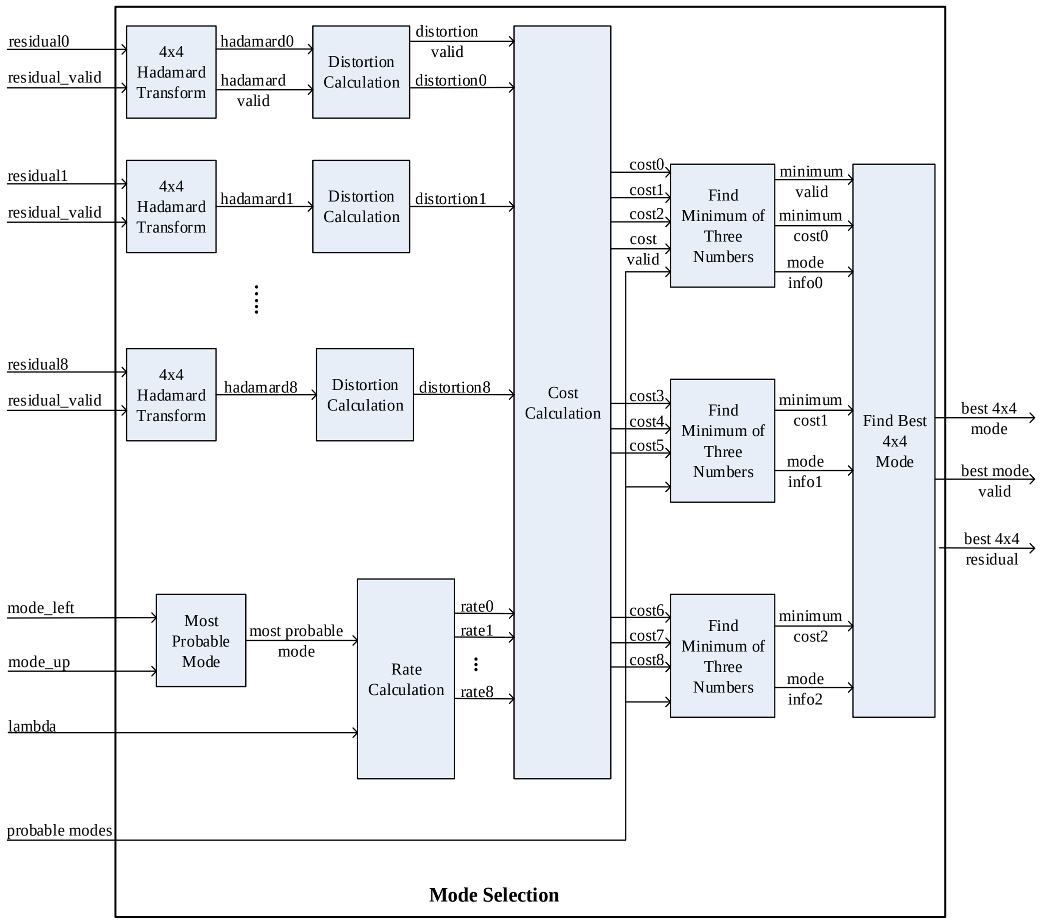


Figure 4.17 FPGA Design Architecture of 4x4 Intra Mode Selection Algorithm

The "4x4 Hadamard Transform" block in Figure 4.17 calculates the Hadamard transform of the residual block as shown in equation 3.11. It applies a two dimensional Hadamard transform, firstly in the horizontal axis and secondly in the vertical axis. The Modelsim simulation results of this block are shown in Figure 4.18. As shown from this figure, when pred_4x4_res_valid flag is asserted, Hadamard transformation operation starts and after two clock cycles the transformed results are driven by asserting the hadamard_valid signal.

The "Distortion Calculation" block calculates the distortion by using the equation 3.12. It takes the Hadamard transformed residual block and calculates the sum of transformed coefficients. The distortion calculation process is started when the Hadamard transformation results are valid (hadamard_valid is high) and the end of the process is declared by asserting the distortion_valid signal. The Modelsim simulation results of this block are shown in Figure 4.19.

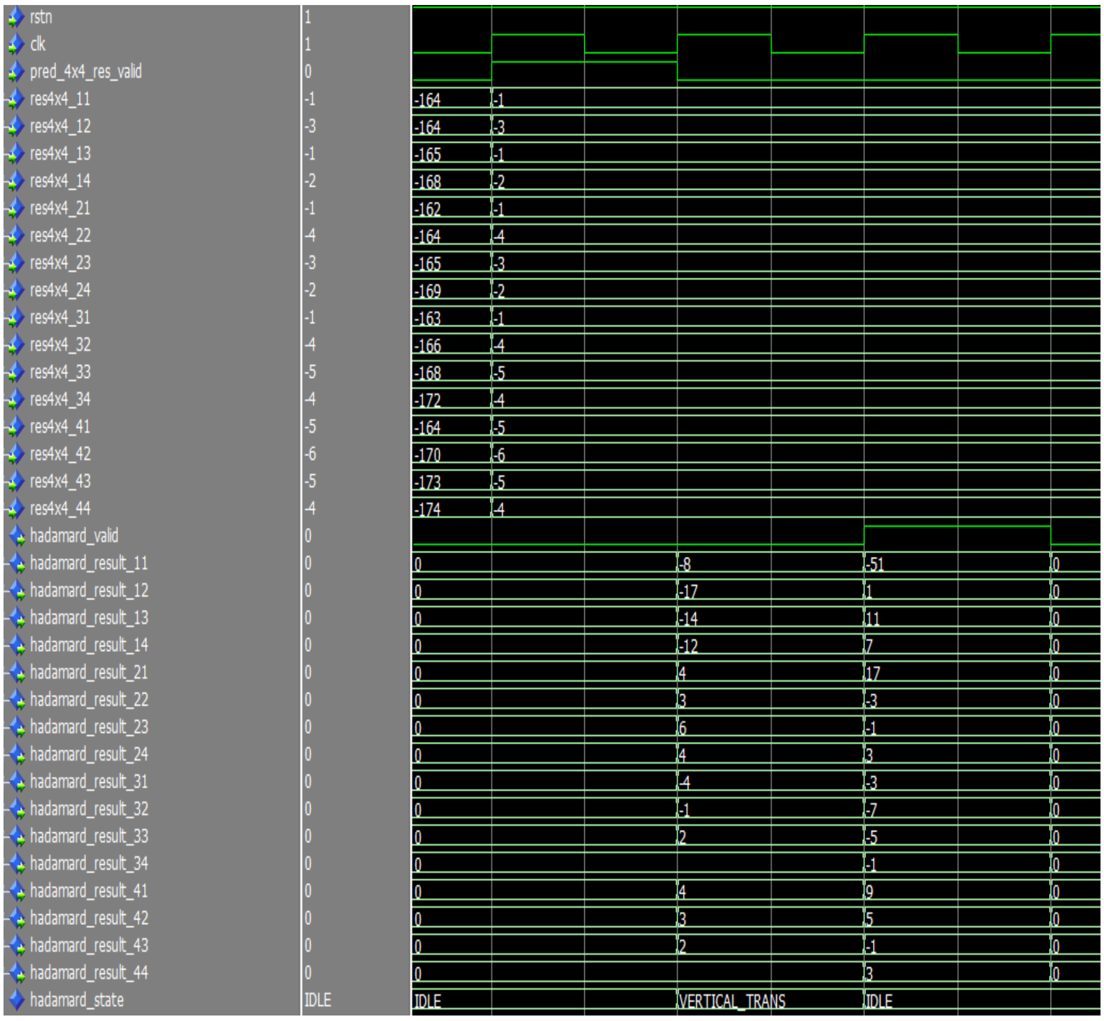

rstn 1
clk 1
pred_4x4_res_valid 0
res4x4_11 -1 -164 -1
res4x4_12 -3 -164 -3
res4x4_13 -1 -165 -1
res4x4_14 -2 -168 -2
res4x4_21 -1 -162 -1
res4x4_22 -4 -164 -4
res4x4_23 -3 -165 -3
res4x4_24 -2 -169 -2
res4x4_31 -1 -163 -1
res4x4_32 -4 -166 -4
res4x4_33 -5 -168 -5
res4x4_34 -4 -172 -4
res4x4_41 -5 -164 -5
res4x4_42 -6 -170 -6
res4x4_43 -5 -173 -5
res4x4_44 -4 -174 -4
hadamard_valid 0
hadamard_result_11 0 0 -8 -51 0
hadamard_result_12 0 0 -17 1 0
hadamard_result_13 0 0 -14 11 0
hadamard_result_14 0 0 -12 7 0
hadamard_result_21 0 0 4 17 0
hadamard_result_22 0 0 3 -3 0
hadamard_result_23 0 0 6 -1 0
hadamard_result_24 0 0 4 3 0
hadamard_result_31 0 0 -4 -3 0
hadamard_result_32 0 0 -1 -7 0
hadamard_result_33 0 0 2 -5 0
hadamard_result_34 0 0 -1 0
hadamard_result_41 0 0 4 9 0
hadamard_result_42 0 0 3 5 0
hadamard_result_43 0 0 2 -1 0
hadamard_result_44 0 0 3 0
hadamard_state IDLE IDLE VERTICAL_TRANS IDLE

Figure 4.18 The 4x4 Hadamard Transform Block Modelsim Simulation Results

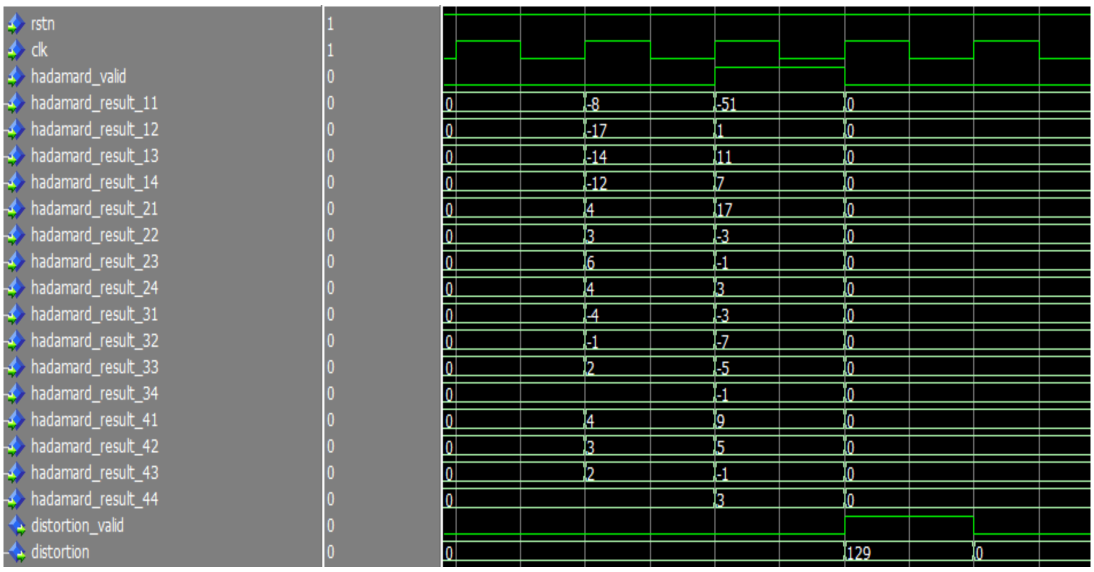

rstn 1
clk 1
hadamard_valid 0
hadamard_result_11 0 0 -8 -51 0
hadamard_result_12 0 0 -17 1 0
hadamard_result_13 0 0 -14 11 0
hadamard_result_14 0 0 -12 7 0
hadamard_result_21 0 0 4 17 0
hadamard_result_22 0 0 3 -3 0
hadamard_result_23 0 0 6 -1 0
hadamard_result_24 0 0 4 3 0
hadamard_result_31 0 0 -4 -3 0
hadamard_result_32 0 0 -1 -7 0
hadamard_result_33 0 0 2 -5 0
hadamard_result_34 0 0 -1 0
hadamard_result_41 0 0 4 9 0
hadamard_result_42 0 0 3 5 0
hadamard_result_43 0 0 2 -1 0
hadamard_result_44 0 0 3 0
distortion_valid 0
distortion 0 0 129 0

Figure 4.19 The Distortion Calculation Block Modelsim Simulation Results

"The Rate Calculation" block and "Most Probable Mode" block calculates the rate values of each mode. "Most Probable Mode" block finds the most probable mode and "Rate Calculation" block calculates the rates of each mode as shown in section 4.2.2.1 at steps 4 and 5, respectively.

"Cost Calculation" block calculates the cost of each mode by using the rate and distortion values by applying equation 3.14. The cost calculation process is started when the distortion_valid signal is high and the end of the process is declared by asserting the cost_valid signal. The Modelsim simulation results of this block are shown in Figure 4.20.

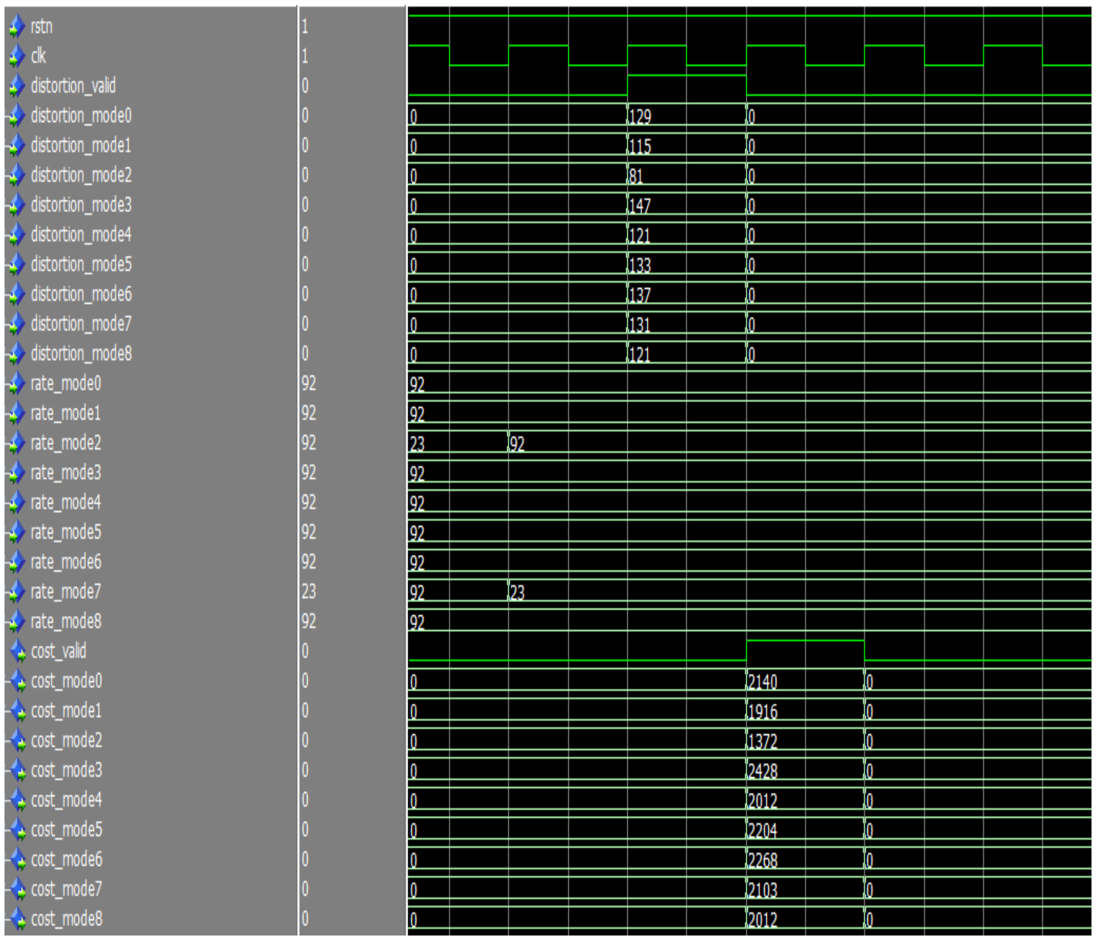


Figure 4.20 The Cost Calculation Block Modelsim Simulation Results

"Find Minimum of Three Numbers" blocks find the smallest one of the three natural numbers (numbers are the cost values in this case). This block uses probable_modes signal to know which prediction modes are valid. If the corresponding bit of a mode in this signal is '0', the cost value of this mode is discarded and this mode is never selected. The Modelsim simulation results of this block are shown in Figure 4.21.

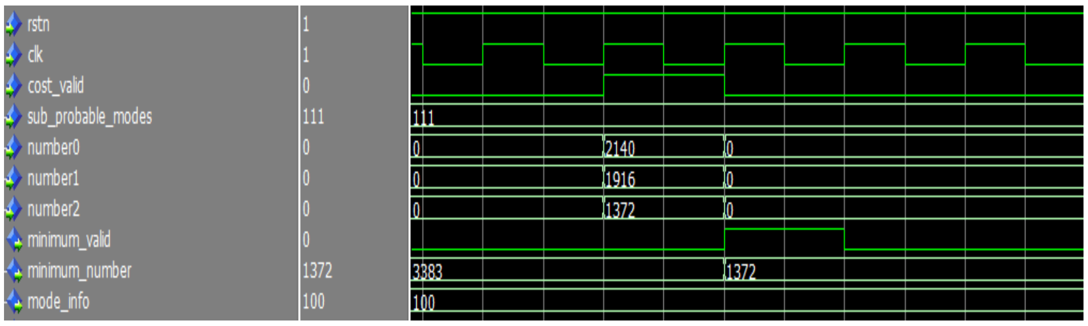
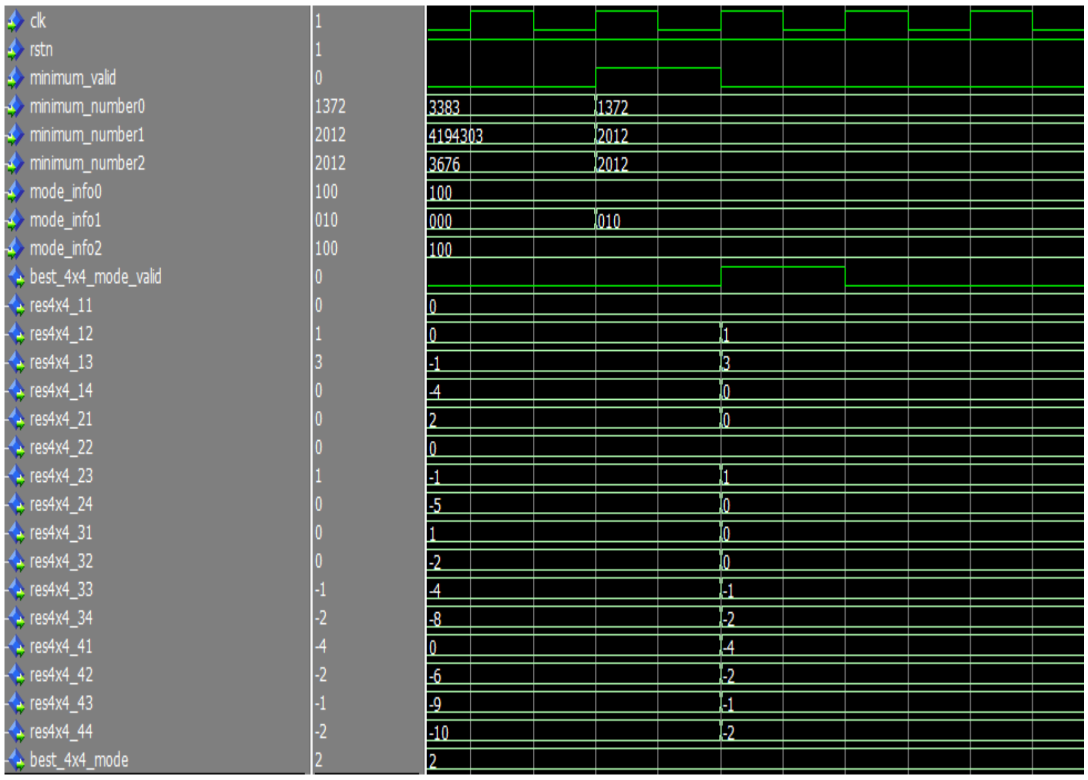


Figure 4.21 Find Minimum of Three Numbers Block Modelsim Simulation Results

.

“Find Best 4x4 Mode” block finds the mode that results with the minimum cost and outputs the related residual block and the prediction mode of this block. The Modelsim simulation results of this block are shown in Figure 4.22. As shown from this figure, residual data and prediction mode information are valid when the best_4x4_mode_valid signal is high.



```
clk                1
rstn               1
minimum_valid      0
minimum_number0    1372    3383      1372
minimum_number1    2012    4194303   2012
minimum_number2    2012    3676      2012
mode_info0         100     100
mode_info1         010     000       010
mode_info2         100     100
best_4x4_mode_valid 0
res4x4_11          0       0
res4x4_12          1       0         1
res4x4_13          3       -1        3
res4x4_14          0       -4        0
res4x4_21          0       2         0
res4x4_22          0       0
res4x4_23          1       -1        1
res4x4_24          0       -5        0
res4x4_31          0       1         0
res4x4_32          0       -2        0
res4x4_33          -1      -4        -1
res4x4_34          -2      -8        -2
res4x4_41          -4      0         -4
res4x4_42          -2      -6        -2
res4x4_43          -1      -9        -1
res4x4_44          -2      -10       -2
best_4x4_mode      2       2
```

Figure 4.22 Find Best 4x4 Mode Block Modelsim Simulation Results

4.5 Resource Usage Summary of 4x4 Intra Prediction and 4x4 Intra Mode Selection Blocks

In Table 4.1, resource usage summary of “4x4 Flow Control”, “4x4 Prediction”, “Residual Calculation” and “Mode Selection” are given. “Mode Selection” block consumes more FPGA logic than others; because cost calculation of each intra prediction mode requires many arithmetic and logical operations (especially “Hadamard Transform” and “Distortion Calculation” operations) and these costs of each nine prediction modes are calculated in parallel.

Table 4.1 Resource Usage Summary of Intra Prediction and Mode Selection

Block Name	ALMs	Dedicated Logic Registers	Block Memory Bits	M20Ks	DSP Blocks
4x4 Flow Control	647/234720	1256/469440	0/52428800	0/2560	0/256
4x4 Prediction	289/234720	265/469440	0/52428800	0/2560	0/256
Residual Calculation	1002/234720	2162/469440	258048/52428800	18/2560	0/256
Mode Selection	9436/234720	8812/469440	0/52428800	0/2560	0/256

CHAPTER 5

H.264/AVC 4x4 TRANSFORM AND QUANTIZATION AND FPGA IMPLEMENTATIONS

After prediction, the transform coding operation is applied to reduce the spatial redundancy of the prediction error signal. All former standards such as MPEG-1 and MPEG-2 used a 2-D DCT of the size 8x8 with floating point arithmetic. Instead, different integer transforms of 4x4 or 8x8 size and corresponding quantization processes are used to approximate the orthonormal 2-D DCT transform.

After transform coding, dynamic range of the transform coefficients is high. For this reason, transform coefficients are quantized. Quantization reduces the precision of the transform coefficients according to a quantization parameter (QP). Typically, the result of quantization is a block in which most or all of the coefficients are zero, with a few non-zero coefficients. In Figure 5.1, the forward transform and quantization processes block diagram is shown.

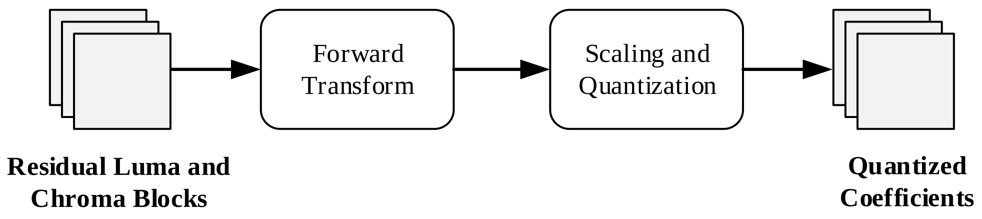


Figure 5.1 Forward Transform and Quantization

Because an encoder uses the previously coded pixels for intra prediction, the quantized video data must be reconstructed not only at the decoder but also at the encoder side. For this reason, inverse operations of transform and quantization are implemented also in an encoder. These inverse operations are exactly defined in H.264/AVC standard documentation. The inverse transform and inverse quantization processes are shown in Figure 5.2.

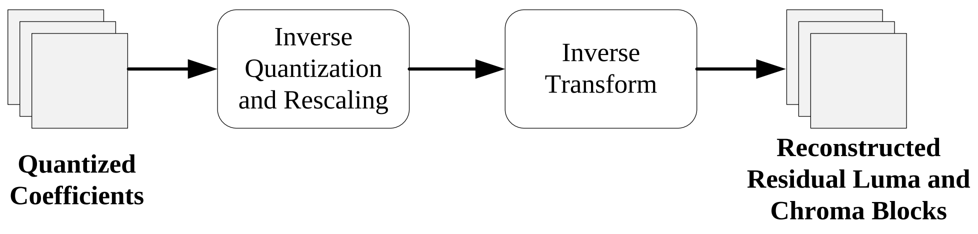

Inverse Quantization and Rescaling
Inverse Transform
Quantized Coefficients
Reconstructed Residual Luma and Chroma Blocks

Figure 5.2 Inverse Transform and Quantization

In this section, transform, inverse transform, quantization and inverse quantization methods used in H.264/AVC and proposed FPGA implementations of these methods are explained.

5.1 Transform Coding

In H.264/AVC, different types of integer transforms are used to minimize computational complexity and to avoid encoder/decoder mismatch. Equation 4.1 shows the general transformation equation, where **A** is the transform matrix, **X** is the residual block and **Y** is the transformation result. The core transform in H.264/AVC is 4x4 or 8x8 integer transform. 8x8 transform is only used in High profiles. 4x4 and 8x8 core transform matrices are shown in Figure 5.3. As shown from these matrices, transform operation can be applied without using any multiplication. It can be achieved by using only addition, subtraction and bit shift operations.

$$\mathbf{Y} = \mathbf{AXA}^{\mathbf{T}} \tag{4.1}$$

$$T_{4x4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \qquad T_{8x8} = \begin{bmatrix} 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 12 & 10 & 6 & 3 & -3 & -6 & -10 & -12 \\ 8 & 4 & -4 & -8 & -8 & -4 & 4 & 8 \\ 10 & -3 & -12 & -6 & 6 & 12 & 3 & -10 \\ 8 & -8 & -8 & 8 & 8 & -8 & -8 & 8 \\ 6 & -12 & 3 & 10 & -10 & -3 & 12 & -6 \\ 4 & -8 & 8 & -4 & -4 & 8 & -8 & 4 \\ 3 & -6 & 10 & -12 & 12 & -10 & 6 & -3 \end{bmatrix}$$

Figure 5.3 4x4 and 8x8 Transform Matrixes

The inverse integer transform operation shown in equation 4.2 is similar to the integer transform operation. Here **W** is the inverse transform matrix, **X** is the matrix obtained after the inverse quantization process and **Z** is the result of the inverse transformation.

$$\mathbf{Z} = \mathbf{W}^{\mathbf{T}}\mathbf{XW} \tag{4.2}$$

5.1.1 Luma 4x4 Transform Processes

The default 4x4 forward and inverse transform processes for luma samples are shown in Figure 5.4 and Figure 5.5, respectively. In this case, a macroblock is divided into 16 4x4 blocks and each 4x4 block is transformed, scaled and quantized. The quantized coefficients blocks are coded and transmitted in the order shown, from 0 to 15.

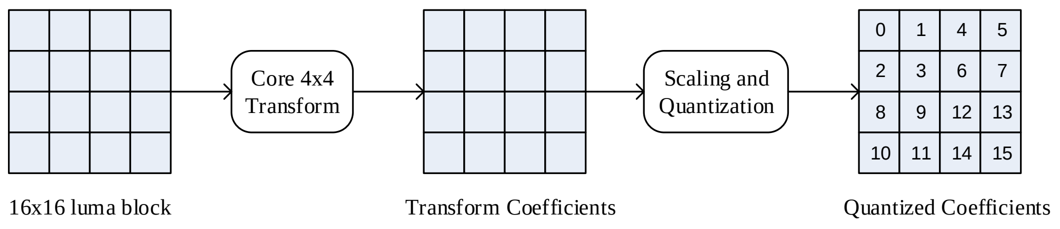


Figure 5.4 Luma Forward Transform: Default

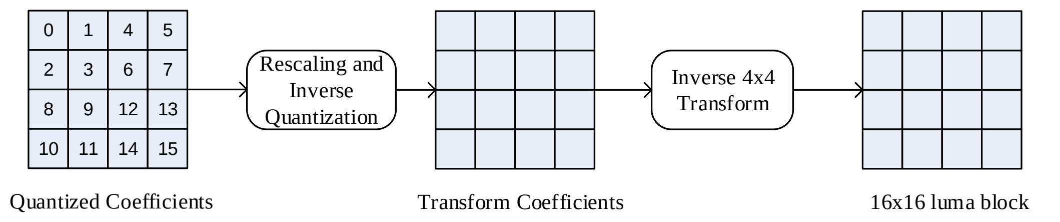


Figure 5.5 Luma Inverse Transform: Default

If the macroblock is predicted using one of the four 16x16 intra prediction modes (Figure 5.6), by using the DC coefficients of the first transform one more transform is performed.

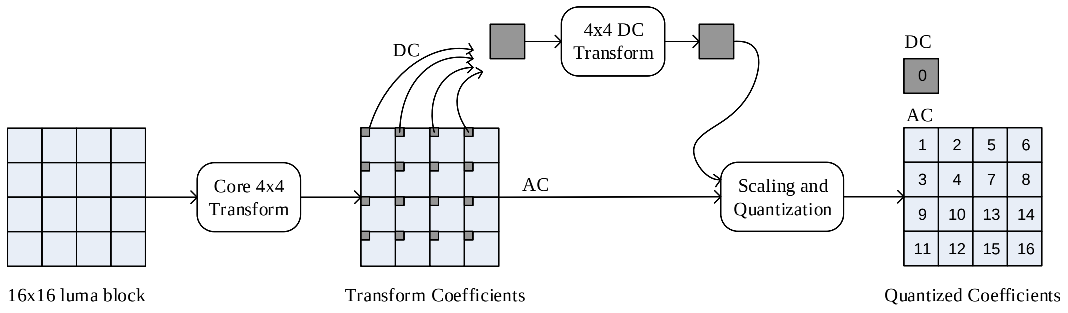


Figure 5.6 Luma Forward Transform: Intra 16x16 Mode

As shown in Figure 5.6, first each 4x4 block is transformed by using 4x4 core transform matrix. Then, the DC coefficients of each 4x4 transformed block are collected to form a 4x4 DC coefficient block. This DC block is further transformed using a 4x4 Hadamard transform. The DC block and 15 AC blocks are scaled and quantized and transmitted in the order shown in Figure 5.6.

In the reverse path (Figure 5.7), first the DC block is inverse Hadamard transformed and then rescaling, inverse quantization and inverse transform operations are applied.

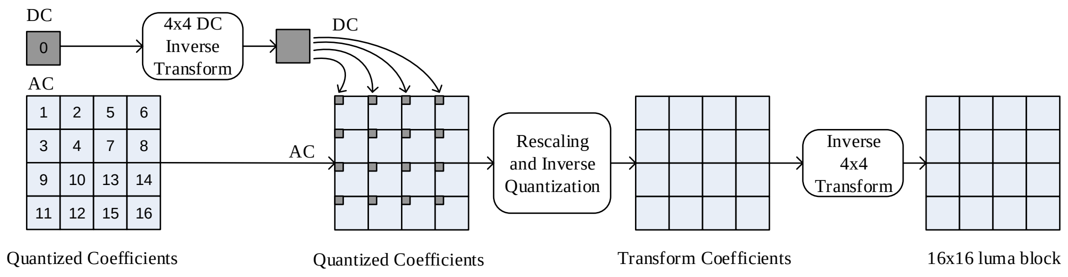


Figure 5.7 Luma Inverse Transform: Intra 16x16 Mode

5.1.2 Chroma Transform Processes

While coding the chroma components of an image, the number of 4x4 blocks that will be coded and transmitted varies with respect to the chroma sampling format (Table 5.1).

Table 5.1 Chroma Sampling Formats and Corresponding Block Sizes

Sampling Format	Chroma Macroblock Size	Number of 4x4 Blocks
4:2:0	8x8	4
4:2:2	16x8	8
4:4:4	16x16	16

The chroma transform operation is similar to the transform process of the intra 16x16 luma blocks. In Figure 5.8 and Figure 5.9, the block diagrams of the chroma forward and inverse transform processes are represented when the sampling format is 4:2:0. As shown from these figures, first the chroma blocks are 4x4 transformed and the DC coefficients of the transformed blocks are collected and further transformed by using a 2x2 Hadamard matrix (the size of this Hadamard matrix changes with respect to the sampling format). Then, the DC blocks and 8 AC blocks are scaled, quantized and transmitted in the order shown in Figure 5.8.

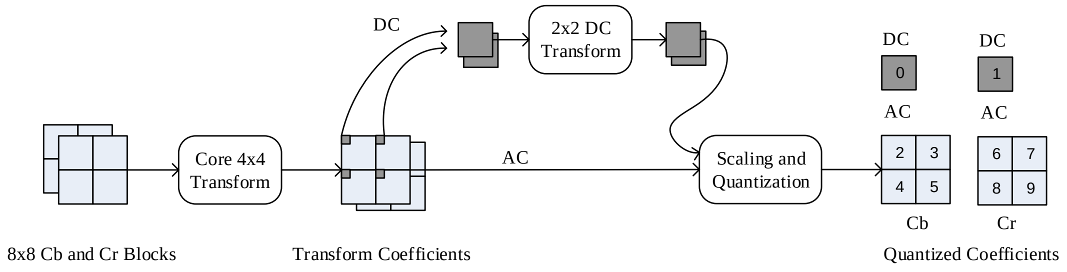


Figure 5.8 Chroma Forward Transform: 4:2:0 Sub-sampling

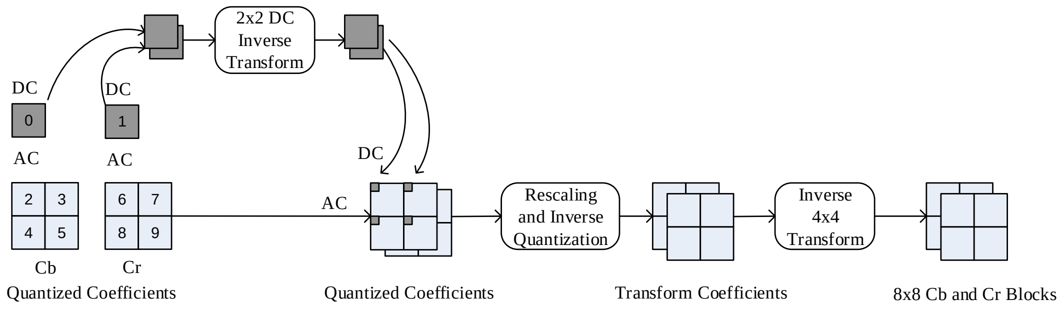


Figure 5.9 Chroma Inverse Transform: 4:2:0 Subsampling

In the reverse path (Figure 5.9), first the DC blocks are inverse Hadamard transformed and then rescaling, inverse quantization and inverse transform operations are applied.

5.2 Quantization

After the transform operation, the transformed coefficients are quantized. This is the point after which lossy compression is achieved. The loss of insignificant data starts from this point depending upon the parameter known as Quantization Parameter (QP).

H. 264/AVC supports 52 different QP values. By using bigger QP values, one can represent a video with a less number of bits at the expense of reduced video quality.

H.264 uses a scalar quantizer. The forward quantization operation is described by equation 4.3. In this equation, MF is the Multiplication Factor, >> indicates a binary shift right, f controls the rounding and qbits is described as shown in equation 4.4. The floor operation in equation 4.4 rounds the input to the nearest integer less than or equal to this input. In the reference software model, f is $2^{qbits/3}$ for intra blocks or $2^{qbits/6}$ for inter blocks. The Multiplication Factor (MF) depends on the QP value and is given in Table 5.2.

$$|Z_{ij}| = (|W_{ij}|.MF + f) >> qbits$$
$$sign(Z_{ij}) = sign(W_{ij}) \qquad (4.3)$$

$$qbits = 15 + floor(QP/6) \qquad (4.4)$$

Table 5.2 Multiplication Factor (MF)

QP%6	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

The inverse quantization operation is described by equation 4.5. In this equation, V is the rescaling factor and given in Table 5.3.

$$W_{ij} = Z_{ij}.V_{ij}.2^{floor(QP/6)} \qquad (4.5)$$

Table 5.3 Rescaling Factor (V)

QP%6	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

5.3 4x4 Transform and Quantization Examples

In Table 5.4, one block of luma or chroma samples is shown. 4x4 transform, quantization, inverse quantization and inverse transform operations are applied to this sample block and the results are shown in Table 5.5.

As can be seen from these tables, when the QP parameter increases, more quantized coefficients get zero (more compression). However, the difference (error) between the original block and the recovered block increases.

Table 5.4 Block of Luma or Chroma Samples

85	83	79	91
76	76	75	81
79	83	86	89
80	85	81	56

Table 5.5 Results of Each Step (Example)

🟥	QP=5				QP=10				QP = 20				QP=40			
Core Transform	1285	12	-11	-9	1285	12	-11	-9	1285	12	-11	-9	1285	12	-11	-9
	43	-106	95	-63	43	-106	95	-63	43	-106	95	-63	43	-106	95	-63
	-5	76	-21	13	-5	76	-21	13	-5	76	-21	13	-5	76	-21	13
	94	-88	30	-24	94	-88	30	-24	94	-88	30	-24	94	-88	30	-24
Quantization	285	2	-2	-1	160	1	-1	-1	49	0	0	0	5	0	0	0
	6	-9	13	-5	3	-5	7	-3	1	-2	2	-1	0	0	0	0
	-1	10	-4	2	0	6	-2	1	0	2	-1	0	0	0	0	0
	13	-8	4	-2	7	-4	2	-1	2	-1	1	0	0	0	0	0
Inverse Transform&Quantization	85	82	79	91	85	83	78	90	84	82	79	91	80	80	80	80
	76	76	75	80	75	76	74	81	74	74	76	83	80	80	80	80
	79	83	86	89	78	83	85	88	81	78	84	88	80	80	80	80
	80	85	80	57	81	85	80	58	81	85	80	58	80	80	80	80

5.4 FPGA Implementation of Transform Coding

In this section, FPGA design of forward and inverse integer transforms are illustrated. The HDL implementation is written in VHDL language.

In Figure 5.10 and Figure 5.11, the block diagrams of forward transform and inverse transform are given. As shown from these figures, in each case the two dimensional transform operations are divided into two parts: vertical and horizontal. When the data at the input port is valid, first the vertical transform is applied and then the horizontal transform operation is applied by using the results of the vertical transform. By doing so, same FPGA resources are used for the horizontal and the vertical transform operations. Transform operation is realized by using sixteen 13 bit 4 input adders.

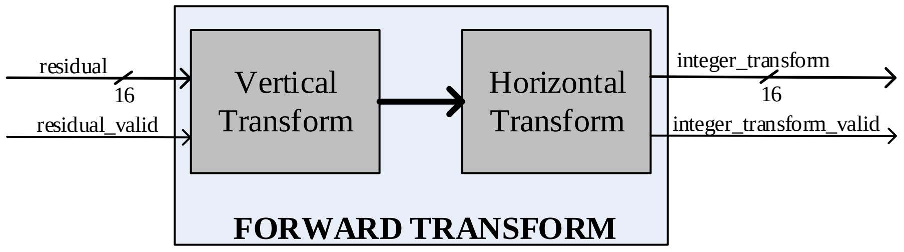


Figure 5.10 Block Diagram of Forward Transform

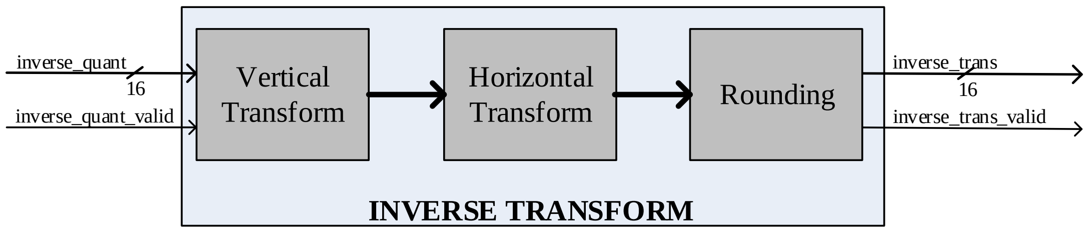


Figure 5.11 Block Diagram of Inverse Transform and Rounding

As shown from Figure 5.11, rounding operation is applied after vertical and horizontal inverse transforms. This rounding block shifts left the results of the horizontal transformed values by 6 bits.

In Figure 5.12 and Figure 5.13, HDL designer view of integer transform and inverse integer transform blocks are represented. HDL designer is an FPGA design tool that visualizes the FPGA designs and commonly used in team based designs. HDL Designer also makes the design of hierarchical blocks easier and increases design reusability.

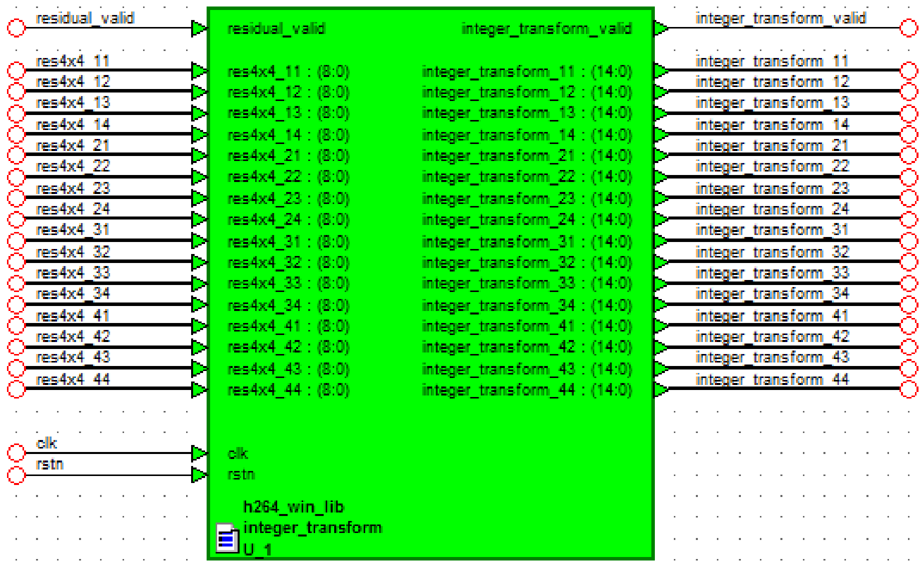
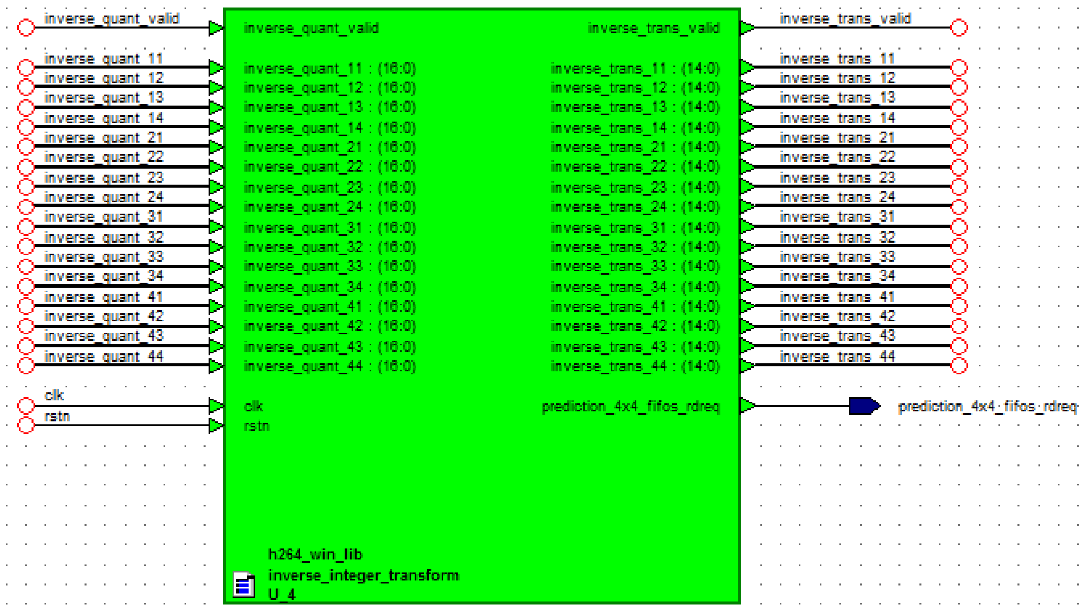


Figure 5.12 HDL Designer View of “Integer Transform” Block



inverse_quant_valid
inverse_quant_11
inverse_quant_12
inverse_quant_13
inverse_quant_14
inverse_quant_21
inverse_quant_22
inverse_quant_23
inverse_quant_24
inverse_quant_31
inverse_quant_32
inverse_quant_33
inverse_quant_34
inverse_quant_41
inverse_quant_42
inverse_quant_43
inverse_quant_44
clk
rstn
inverse_quant_valid
inverse_quant_11 : (16:0)
inverse_quant_12 : (16:0)
inverse_quant_13 : (16:0)
inverse_quant_14 : (16:0)
inverse_quant_21 : (16:0)
inverse_quant_22 : (16:0)
inverse_quant_23 : (16:0)
inverse_quant_24 : (16:0)
inverse_quant_31 : (16:0)
inverse_quant_32 : (16:0)
inverse_quant_33 : (16:0)
inverse_quant_34 : (16:0)
inverse_quant_41 : (16:0)
inverse_quant_42 : (16:0)
inverse_quant_43 : (16:0)
inverse_quant_44 : (16:0)
clk
rstn
inverse_trans_valid
inverse_trans_11 : (14:0)
inverse_trans_12 : (14:0)
inverse_trans_13 : (14:0)
inverse_trans_14 : (14:0)
inverse_trans_21 : (14:0)
inverse_trans_22 : (14:0)
inverse_trans_23 : (14:0)
inverse_trans_24 : (14:0)
inverse_trans_31 : (14:0)
inverse_trans_32 : (14:0)
inverse_trans_33 : (14:0)
inverse_trans_34 : (14:0)
inverse_trans_41 : (14:0)
inverse_trans_42 : (14:0)
inverse_trans_43 : (14:0)
inverse_trans_44 : (14:0)
prediction_4x4_fifos_rdreq
h264_win_lib
inverse_integer_transform
U_4
inverse_trans_valid
inverse_trans_11
inverse_trans_12
inverse_trans_13
inverse_trans_14
inverse_trans_21
inverse_trans_22
inverse_trans_23
inverse_trans_24
inverse_trans_31
inverse_trans_32
inverse_trans_33
inverse_trans_34
inverse_trans_41
inverse_trans_42
inverse_trans_43
inverse_trans_44
prediction_4x4_fifos_rdreq

Figure 5.13 HDL Designer View of "Inverse Integer Transform" Block

The Modelsim simulation results of forward and inverse integer transform operations are given in Figure 5.14 and Figure 5.15, respectively. As shown from these figures, integer transform operation takes 2 clock cycles and inverse integer transform operation takes 3 clock cycles: one clock cycle for vertical transform, one clock cycle for horizontal transform and one clock cycle for rounding operation.

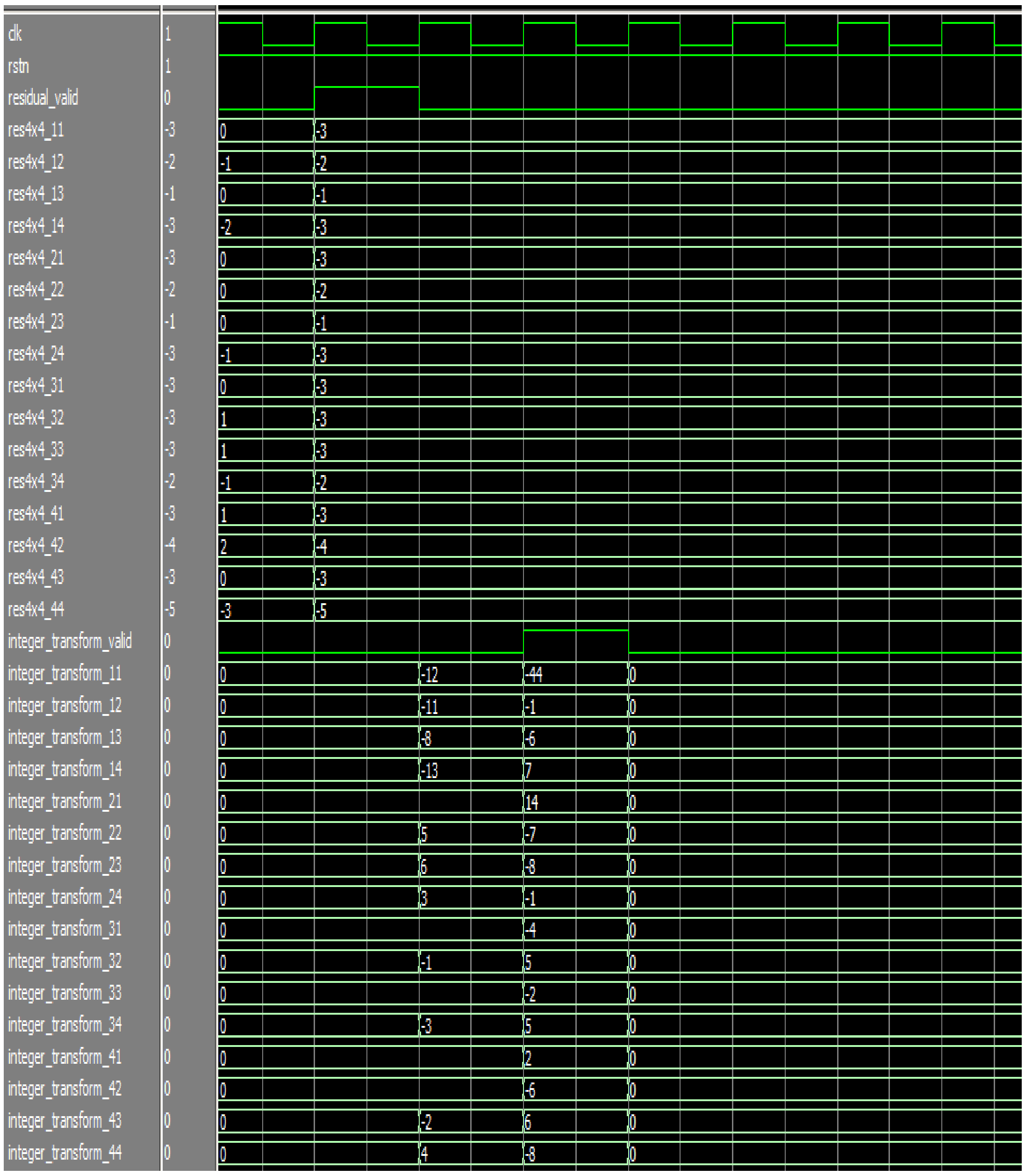

clk 1
rstn 1
residual_valid 0
res4x4_11 -3 0 -3
res4x4_12 -2 -1 -2
res4x4_13 -1 0 -1
res4x4_14 -3 -2 -3
res4x4_21 -3 0 -3
res4x4_22 -2 0 -2
res4x4_23 -1 0 -1
res4x4_24 -3 -1 -3
res4x4_31 -3 0 -3
res4x4_32 -3 1 -3
res4x4_33 -3 1 -3
res4x4_34 -2 -1 -2
res4x4_41 -3 1 -3
res4x4_42 -4 2 -4
res4x4_43 -3 0 -3
res4x4_44 -5 -3 -5
integer_transform_valid 0
integer_transform_11 0 0 -12 -44 0
integer_transform_12 0 0 -11 -1 0
integer_transform_13 0 0 -8 -6 0
integer_transform_14 0 0 -13 7 0
integer_transform_21 0 0 14 0
integer_transform_22 0 0 5 -7 0
integer_transform_23 0 0 6 -8 0
integer_transform_24 0 0 3 -1 0
integer_transform_31 0 0 -4 0
integer_transform_32 0 0 -1 5 0
integer_transform_33 0 0 -2 0
integer_transform_34 0 0 -3 5 0
integer_transform_41 0 0 2 0
integer_transform_42 0 0 -6 0
integer_transform_43 0 0 -2 6 0
integer_transform_44 0 0 4 -8 0

Figure 5.14 Modelsim Simulation Result of “Integer Transform” Block

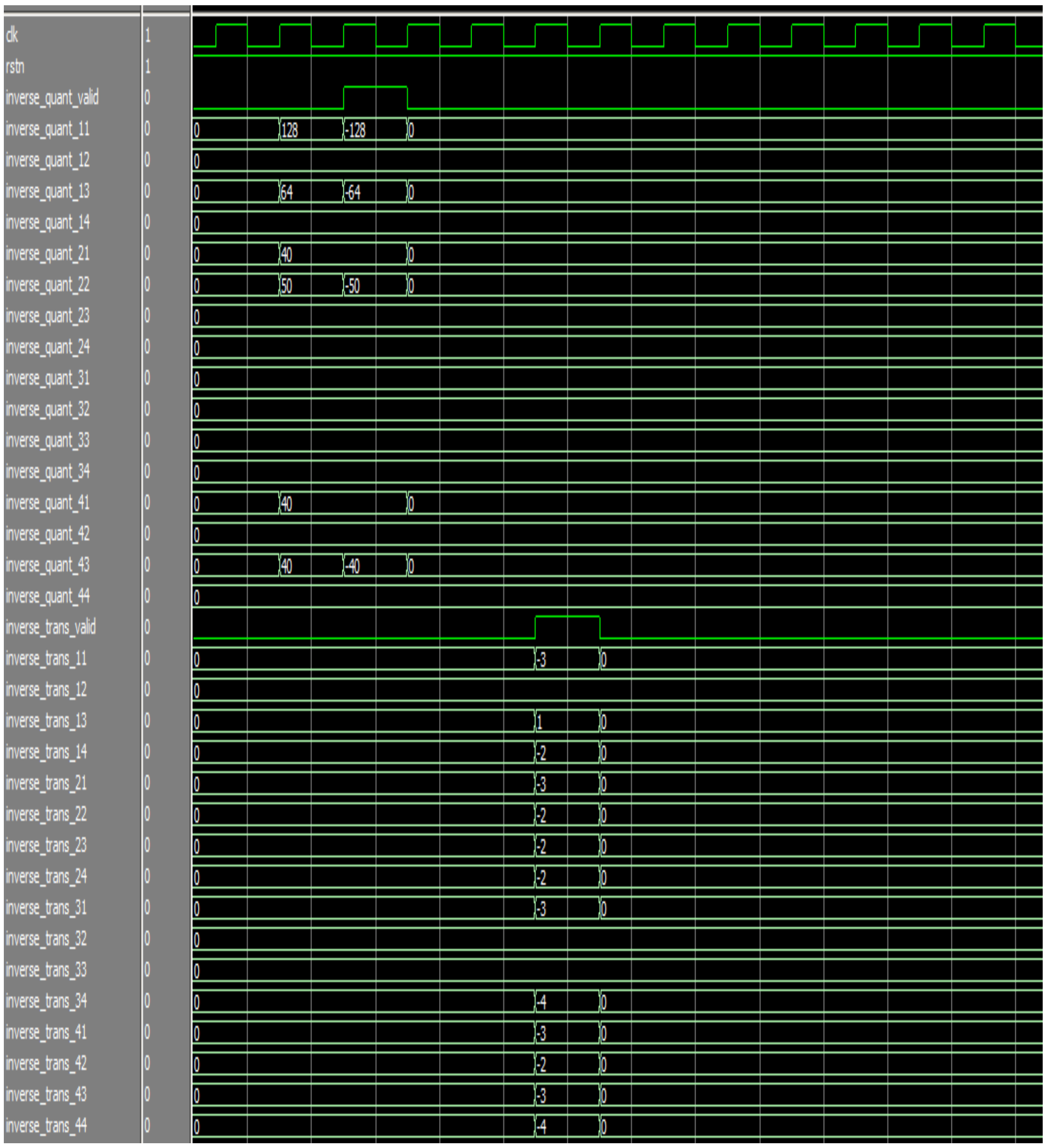

clk 1
rstn 1
inverse_quant_valid 0
inverse_quant_11 0 0 128 -128 0
inverse_quant_12 0 0
inverse_quant_13 0 0 64 -64 0
inverse_quant_14 0 0
inverse_quant_21 0 0 40 0
inverse_quant_22 0 0 50 -50 0
inverse_quant_23 0 0
inverse_quant_24 0 0
inverse_quant_31 0 0
inverse_quant_32 0 0
inverse_quant_33 0 0
inverse_quant_34 0 0
inverse_quant_41 0 0 40 0
inverse_quant_42 0 0
inverse_quant_43 0 0 40 -40 0
inverse_quant_44 0 0
inverse_trans_valid 0
inverse_trans_11 0 0 -3 0
inverse_trans_12 0 0
inverse_trans_13 0 0 1 0
inverse_trans_14 0 0 -2 0
inverse_trans_21 0 0 -3 0
inverse_trans_22 0 0 -2 0
inverse_trans_23 0 0 -2 0
inverse_trans_24 0 0 -2 0
inverse_trans_31 0 0 -3 0
inverse_trans_32 0 0
inverse_trans_33 0 0
inverse_trans_34 0 0 -4 0
inverse_trans_41 0 0 -3 0
inverse_trans_42 0 0 -2 0
inverse_trans_43 0 0 -3 0
inverse_trans_44 0 0 -4 0

Figure 5.15 Modelsim Simulation Result of "Inverse Integer Transform" Block

5.5 FPGA Implementation of Quantization

5.5.1 Forward Quantization

In order to achieve the quantization operation, the algorithmic steps shown in Figure 5.16 are implemented. As shown in this figure, there are four states: IDLE, WAIT1, ADD_OFFSET and INV_QUANT_MULT. In the IDLE state, the sign of each transformed coefficients are stored, the transform coefficients and the corresponding multiplication factors are sent to the multiplier blocks and the rounding offset f (equation 4.3) is calculated as described in equation 4.6. Actually this f value adds 1/3 to each coefficient before rounding. By doing this operation as shown in equation 4.6, the division operation is achieved only using the bit shift operations. So we do not need a divider block.

$$f = 682*2^{(4 + \mathrm{floor}(QP,6))} \tag{4.6}$$

In the WAIT1 state, we wait for the multiplication result and then go to the ADD_OFFSET state. In this state, the calculated offset in the IDLE state and the multiplication result is added. In the final state, INV_QUANT_MULT, the coefficients are right shifted with qbits and the result is written into a FIFO for entropy coding.

In H.264/AVC, if the entropy coding type is CAVLC, the quantized coefficients must be in the range of (-2063, 2063). So if a coefficient is not in this range, this coefficient is limited with these boundaries before writing it into the FIFO. This check is also done in INV_QUANT_MULT state.

In our design, inverse quantization and rescaling operation starts just after the quantization operation. So in order to save 1 clock cycle, we feed each multiplier block with the quantized coefficients and the corresponding inverse multiplication factor (V) in INV_QUANT_MULT state.

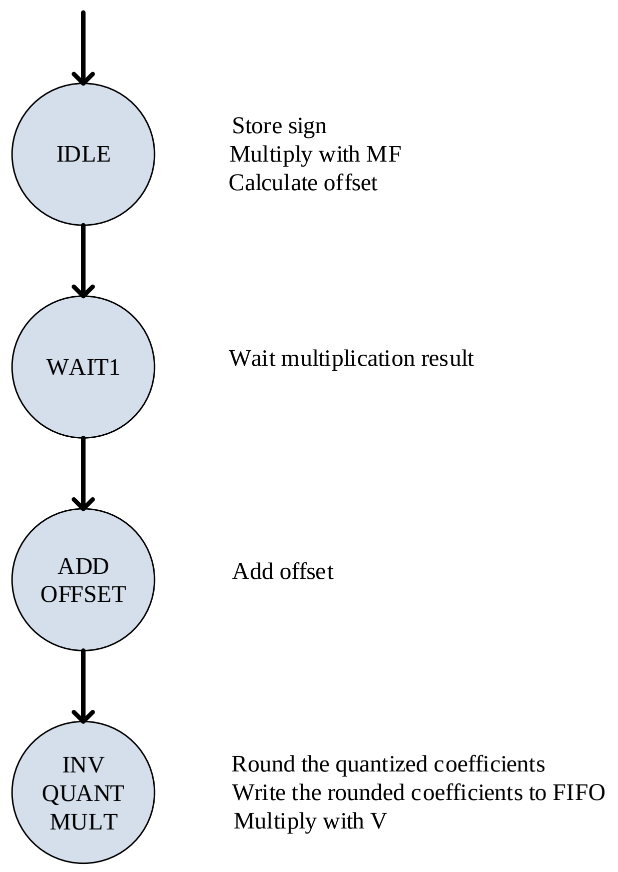


Figure 5.16 State Diagram of Quantization Process

5.5.2 Inverse Quantization

In this part, the inverse quantization and rescaling equation (equation 4.5) is implemented. The state diagram of this process is represented in Figure 5.17. As shown in this figure, the inverse quantization operation is divided into three states. In the IDLE state, the multiplication result of the quantized coefficients with the reverse multiplication factor ($Z_{ij}.V_{ij}$ in equation 4.5) is waited. In the next state, PROCESS_INV_SCALED_DATA, the rescaling of the multiplication result with a factor of $2^{floor(QP/6)}$ is done. In the final state, the sign of each coefficient is added. In this step, the stored sign data in the quantization process is used.

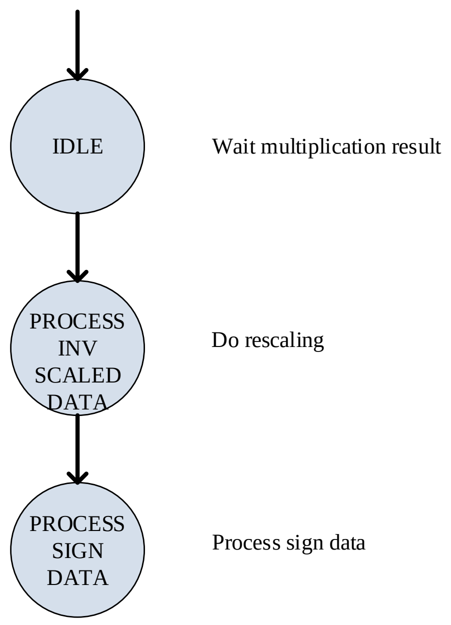


Figure 5.17 State Diagram of Inverse Quantization Process

In Figure 5.18, Modelsim simulation results of quantization and inverse quantization process are given. The quantization process takes 4 clock cycles and inverse quantization operation takes 3 clock cycles. So obtaining the inverse quantization coefficients from the inverse transformed coefficients takes totally 7 clock cycles.

In the design of quantization process sixteen 18x18 multipliers are used and the same multipliers are also used during the inverse quantization process in order to save FPGA’s multipliers.

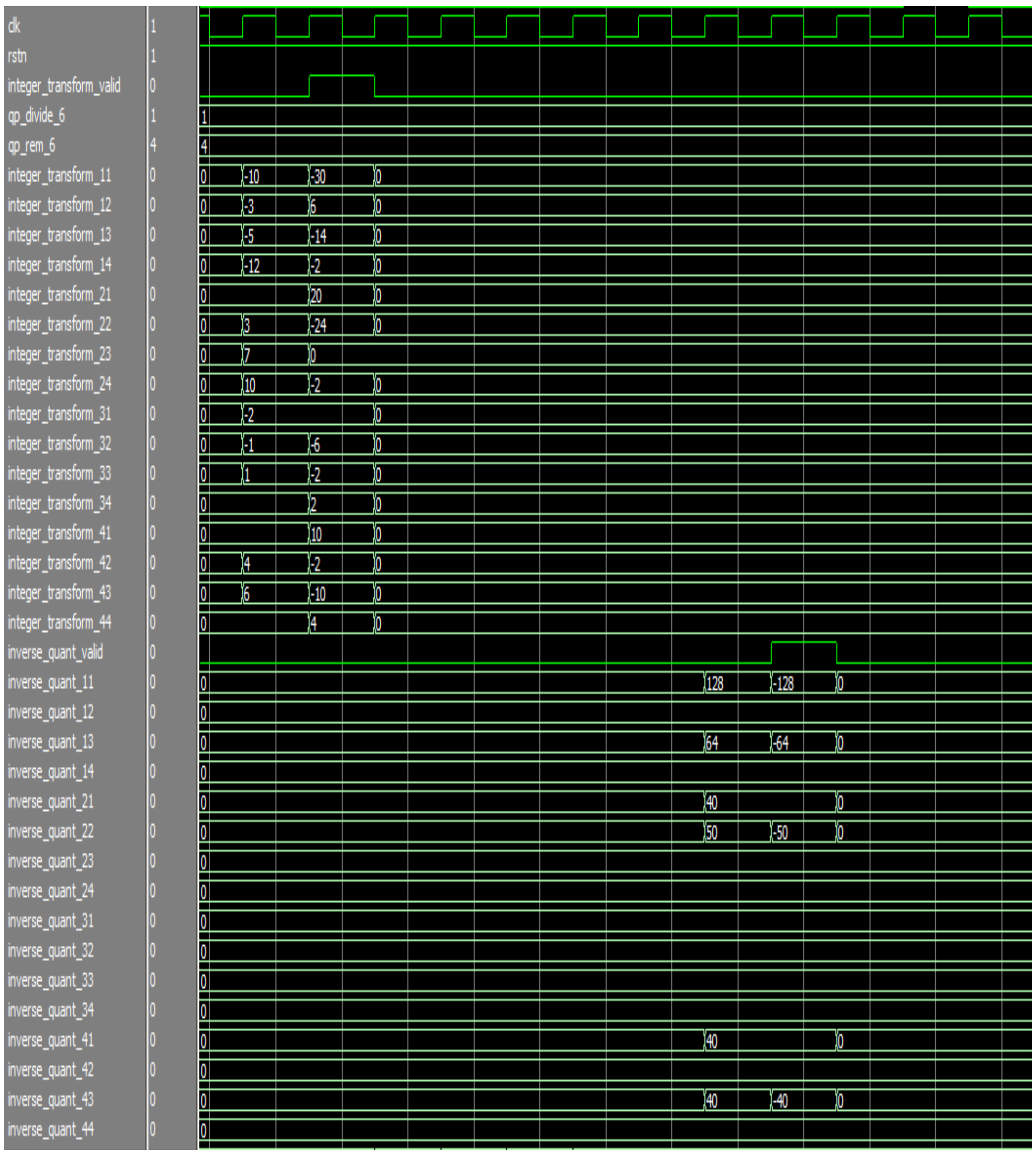


Figure 5.18 Modelsim Simulation Results of "Quantization" and "Inverse Quantization" Process

5.6 Resource Usage Summary of Transform and Quantization

FPGA resource usage summary of integer transform, inverse integer transform, quantization and inverse quantization blocks are given in Table 5.6. Actually, forward integer transform and backward integer transform are same in H.264 literature. But in our implementation (see Figure 5.11), “Inverse Integer Transform” block includes one more block besides “Horizontal Transform” and “Vertical Transform” blocks which is called as “Rounding”. This block is needed to scale the inverse transformed coefficients. For this reason, logic usage of “Inverse Integer Transform” block is higher than logic usage of “Integer Transform”.

Table 5.6 FPGA Resource Usage Summary of Transform and Quantization

Block Name	ALMs	Dedicated Logic Registers	Block Memory Bits	M20Ks	DSP Blocks
Integer Transform	832/234720	824/469440	0/52428800	0/2560	0/256
Inverse Integer Transform	1458/234720	1318/469440	0/52428800	0/2560	0/256
Quantization&Inverse Quantization	2572/234720	1039/469440	0/52428800	0/2560	16/256

CHAPTER 6

BASELINE PROFILE ENTROPY CODING TECHNIQUES AND THEIR FPGA IMPLEMENTATIONS

As mentioned in Chapter 2 (Figure 2.1), the last part of an H.264/AVC video encoder is the entropy coding part. In this part, the previously coded symbols (parameters, identifiers, prediction types, motion vectors and quantized transform coefficients) are coded to generate the H.264/AVC syntax described in section 2.3. Entropy coding is a lossless data compression technique. Entropy coding algorithms try to assign shortest codes to the most commonly occurred symbols at the input in order to produce smaller bitstream.

The H.264/AVC standard specifies several entropy coding methods. These methods are fixed length code, Exponential-Golomb variable length code, CAVLC (Context-based Adaptive Variable Length Coding) and CABAC (Context-based Adaptive Binary Arithmetic Coding). As mentioned in section 2.2.2.4 (Entropy Coding), an encoder may use CAVLC or CABAC while coding the symbols at the slice data level and below (see Figure 2.17). CABAC algorithm is more complex but on the average gives about 10% better results when compared to CAVLC [9]. In the baseline profile, only CAVLC method is allowed and CABAC tool is not supported. So in our encoder implementation CAVLC method is used.

In the following sections of this chapter, firstly the details of fixed-length coding, Exponential-Golomb variable length coding and CAVLC coding methods are given and then FPGA implementations of these algorithms are studied.

6.1 Fixed Length Code

In fixed length coding, a binary code is generated for a symbol with a specific length (n bits). In H.264, the descriptor f(n), i(n), u(n) and b(8) are used for syntax elements which are coded with fixed length code. Here “n” represents the length of the coded symbol. In Table 6.1, fixed length coding descriptors used in H.264/AVC and their meanings are given.

Table 6.1 H.264 Fixed Length Coding Descriptors

Descriptor	Description
f(n)	Fixed-pattern bit string using n bits.
i(n)	Signed integer bit string using n bits. When “n” is “v”, the number of bits varies in a manner dependent on the value of other syntax elements.
u(n)	Unsigned integer using n bits. When “n” is “v” the number of bits varies in a manner dependent on the value of other syntax elements.
b(n)	Byte having any pattern of bit string (8 bits).

In Table 6.2, some H.264/AVC syntax elements which are coded by using the fixed length coding method are shown.

Table 6.2 Some Fixed Length Coded Syntax Elements

Syntax Element	Descriptor
forbidden_zero_bit	f(1)
emulation_prevention_three_byte	f(8)
cabac_zero_word	f(16)
rbsp_stop_one_bit	f(1)
rbsp_alignment_zero_bit	f(1)
time_offset	i(v)
nal_ref_idc	u(2)
nal_unit_type	u(2)
profile_idc	u(8)
level_idc	u(8)
rbsp_byte	b(8)
user_data_payload_byte	b(8)

6.2 Exponential-Golomb Code

Exponential Golomb codes are binary codes with varying lengths generated using a regular pattern. Short codewords are assigned to frequently-occurring symbols. In this way, data can be denoted in a compressed form. In Table 6.3 some symbols represented by code_num and their corresponding Exp-Golomb codewords are given.

The structure of an Exp-Golomb codeword is shown in Figure 6.1. The codeword consist of M zeros, a 1 and M-bit information field, INFO. Codewords are generated from the parameter code_num as specified below:

$$M = \text{floor}(\log_2(\text{code\_num} + 1))$$
$$\text{INFO} = \text{code\_num} + 1 - 2^M$$

First by using code_num, the length of INFO (M) is calculated and then using M and code_num the INFO field is calculated.

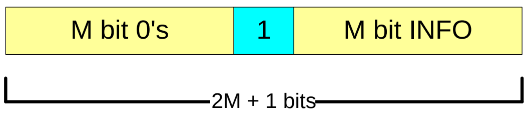


Figure 6.1 Exp-Golomb Codeword Structure

In Table 6.3, some code_num and their corresponding Exp-Golomb codewords are given. As shown from this table, codeword length increases when the parameter code_num increases. So in order to use Exp-Golomb codes efficiently, one should assign smaller code_num to most probable symbols.

H.264/AVC standard defines that, for each symbol "k" to be coded with Exp-Golomb, there is a rule that maps the "k" value to non-negative and integer values. This rule indicates how the symbol "k" must be mapped to a CodeNum value. There are four mapping possibilities, depending on the element type: ue(v) (unsigned Exp-Golomb), te(v) (truncated Exp-Golomb), se(v) (signed Exp-Golomb) and me(v) (mapped Exp-Golomb).

ue(v): Unsigned direct mapping. Code_num = k. Used for mb_type, intra_chroma_pred_mode and others.

te(v): Truncated mapping. If the largest possible value of "k" is 1, then a single bit b is sent where b =! code_num (! Means Boolean logical "not"), otherwise "ue" mapping is used. Reference picture index is coded with truncated mapping.

se(v): Signed mapping. "k" is mapped to code_num as specified below:

code_num = 2|k| (k≤0)

code_num = 2|k| - 1 (k>0)

and code_num is mapped to "k" as follows:

$k = (-1)^{\text{code\_num} + 1} \text{ceil} (\text{code\_num}/2)$

Motion vector differences and quantization parameters are coded with signed Exp-Golomb coding.

me(v): Mapped symbols, k is mapped to code_num according to a table. coded_block_pattern is coded with mapped Exp-Golomb coding (Appendix A).

Table 6.3 Exp-Golomb Codewords

code_num	**Codeword**
0	1
1	010
2	011
3	00100
4	00101
5	00110

6.3 Context-based Adaptive Variable Length Code (CAVLC)

CAVLC is a form of entropy coding methods used in H.264/AVC. CAVLC is used to encode residual, scan ordered from 4x4 or 2x2 blocks of transform coefficients. CAVLC is supported in all H.264 profiles.

CAVLC is designed to take advantage of several characteristics of quantized coefficient blocks [10]: Figure 6.2 shows the general flow of CAVLC coding. It takes 4x4 or 2x2 quantized coefficients and performs the following operations:

1. The input quantized block is reordered using zigzag or field scan.
2. The number of trailing ones (T1's) and the total number of non-zero coefficients are coded together (coeff_token).
3. Each sign of T1's are encoded.

4. The level, sign and magnitude, of each remaining non-zero coefficients are encoded.
5. The total number of zero coefficients before the first non-zero coefficient is encoded (total_zeros).
6. The number of zero valued coefficients before each non-zero coefficient (run before) is encoded.

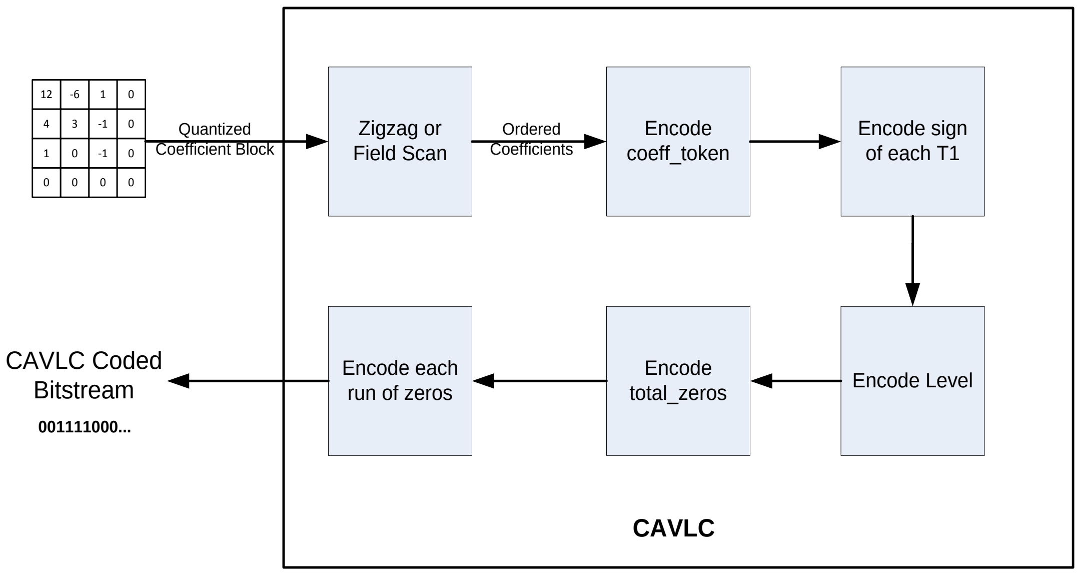


Figure 6.2 Algorithmic Flow of CAVLC

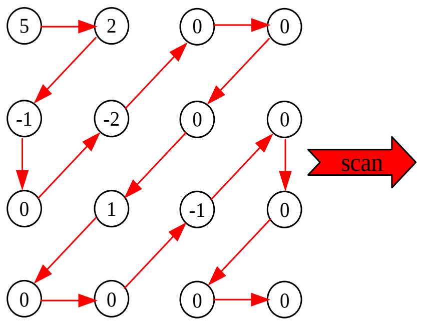
The details of CAVLC coding algorithm (shown in Figure 6.2) are not explained in this thesis work. You can find the details of this algorithm in [9]. In the next section, a 4x4 block is coded by using CAVLC method.

6.3.1 Example CAVLC Coding of a 4x4 Block

In this section, CAVLC coding of the 4x4 block shown below is explained in details.

5	2	0	0
-1	-2	0	0
0	1	-1	0
0	0	0	0

Step 1: Zigzag Scan

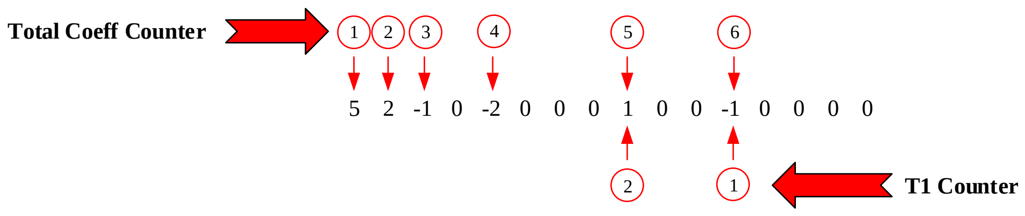


Reordered Block

5 2 -1 0 -2 0 0 0 1 0 0 -1 0 0 0 0

Step 2: Encode coeff_token

As shown below, the total number of non-zero coefficients is 6 and the total number of trailing ones is 2. As explained in [9], the table that will be used to code the symbol coeff_token depends on the parameter nC. In this example, nL and nU are assumed to be zero, so the value of nC is also zero and the VLC table 1 (see Table A-3) is chosen while coding the parameter coeff_token. When we look at the VLC table 1, the corresponding codeword is ‘0000000101’.



Step 3: Encode Sign of Each Trailing Ones

There are two trailing ones. The sign of the first one is –, and the second one is +. So the code for T1 is '10'.

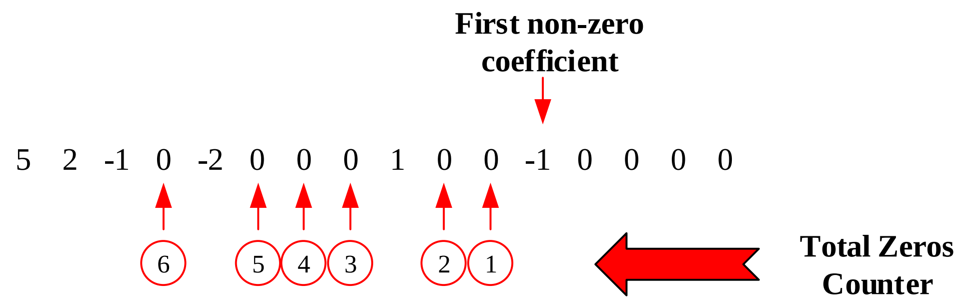
Step 4: Encode Level

The four level values and the corresponding codes are shown below. So the codeword for level parameter is '0111010000010'.

level	level_prefix	level_suffix	code
-2	01	-	01
-1	1	1	11
2	01	0	010
5	00001	0	000010

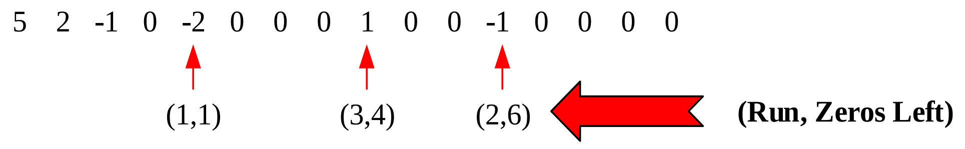
Step 5: Encode total_zeros

The number of total zeros is six, so the corresponding codeword is '011'.



Step 6: Encode run of zeros

The three (run, zeros_left) pairs and the corresponding codes are shown below. The code for run parameter is '0010010'.



(run, zeros_left)	code
(2,6)	001
(3,4)	001
(1,1)	0

6.4 FPGA Implementation of Exponential-Golomb Coding

FPGA design architecture of Exponential-Golomb coding block is represented in Figure 6.3. As shown from this figure, Exponential-Golomb coding block consists of three subblocks: EXP-GOLOMB STATE MACHINE, ROM INTRA CBP and ROM INTER CBP.

In Table 6.4, the input and output signal definitions, length and modes are given. EXP-GOLOMB CODING block works as follows: If the start flag (exp_golomb_start) is high, EXP-GOLOMB STATE MACHINE block reads the parameter that will be coded (exp_golomb_k_param), parameter mapping type (exp_golomb_mode) and prediction type (exp_golomb_mode_p) of the current block. By using these signals, it constructs the code that corresponds to the current symbol and outputs the code (exp_golomb_output) and its length (exp_golomb_length) with a ready flag (exp_golomb_valid). Users should observe the ready flag to understand the end of the coding operation and to code the next symbol.

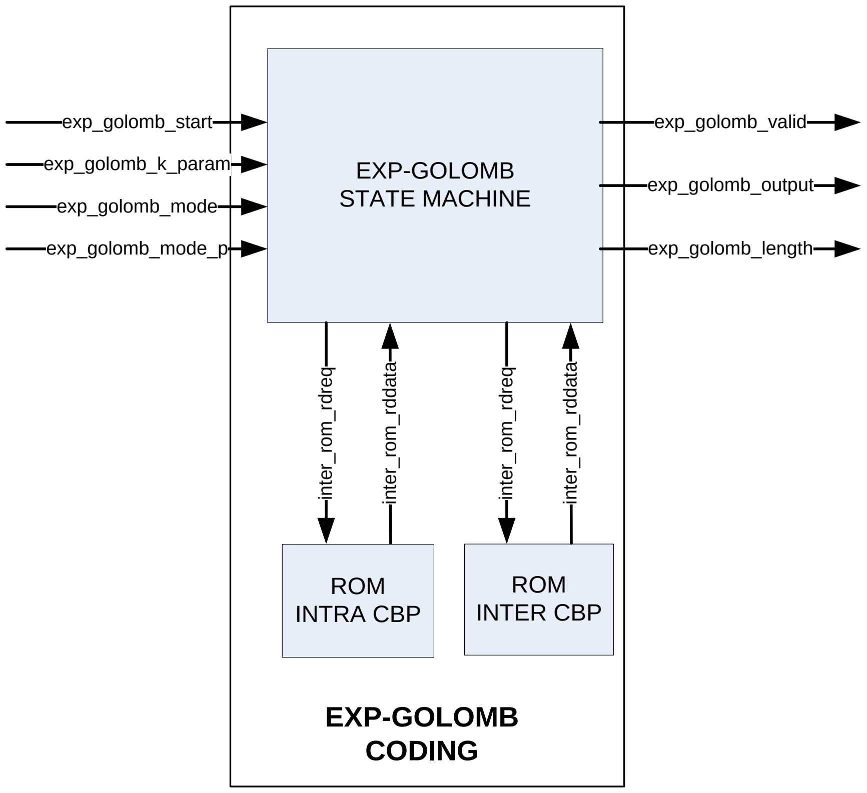


Figure 6.3 Block Diagram of Exponential-Golomb Coding

Table 6.4 I/O Signal Descriptions of the EXP-GOLOMB CODING Block

Signal	Length	Description	Input/Output
exp_golomb_start	1	Indicates the start of coding operation.	Input
exp_golomb_k_param	6	The value of coding parameter.	Input
exp_golomb_mode	2	Indicates the mapping type of the coding parameter. "00" => ue(v) "01" => sev(v) "10" => me(v) "11" => te(v)	Input
exp_golomb_mode_p	1	Prediction type of the coding parameter. Valid only if the mapping type is me(v). '0' => Intra '1' => Inter	Input
exp_golomb_valid	1	If high indicates that the coding result of the current symbol is ready.	Output
exp_golomb_length	4	The length of the coded symbol.	Output
exp_golomb_output	13	The generated code for the current symbol.	Output

6.4.1 EXP-GOLOMB CODING STATE MACHINE

This block is responsible from the generation of the coded bitstream. It includes four states (Table 6.4). In the first state, the code_num is calculated by using the exp_golomb_k_param and exp_golomb_mode. In the second state, the length of INFO field (M) is calculated and in the third state, by using code_num and M parameters INFO field of the bitstream is constructed. In the last state, the output codeword and its length are generated by using M and INFO variables.

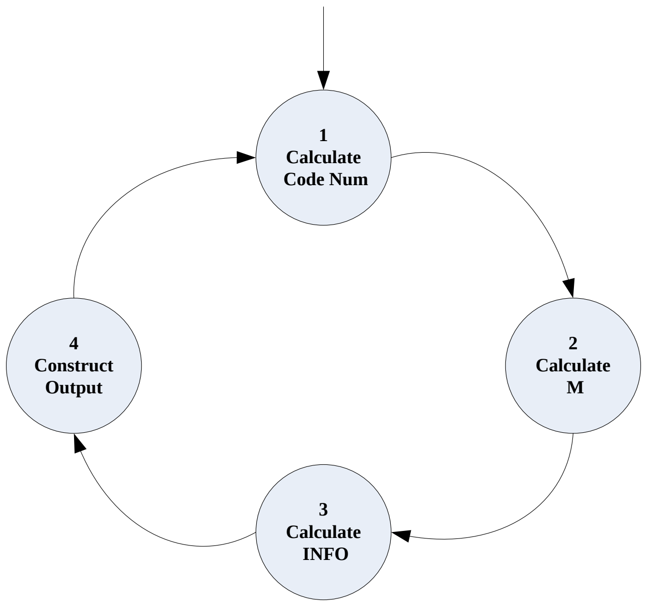


Figure 6.4 State Diagram of Exponential-Golomb State Machine

6.4.2 ROM INTRA CBP

This ROM keeps the corresponding value of code_num parameters for the coded block pattern (CBP) symbol when the prediction type is INTRA. While reading the code_num value, ROM address must be set to the CBP value. The ROM size is 48x6.

6.4.3 ROM INTER CBP

This ROM keeps the corresponding value of code_num parameters for the coded block pattern (CBP) symbol when the prediction type is INTER. While reading the code_num value, ROM address must be set to the CBP value. The ROM size is 48x6.

6.4.4 Simulation and Implementation Results of Exponential-Golomb Coding Block

In Figure 6.5, Figure 6.6 and Figure 6.7, the Modelsim simulation results of the Exp-Golomb block are given when the mapping type is ue(v), se(v) and me(v), respectively. In Table 6.5, the required clock delay of the Exp-Golomb block is given for various mapping types.

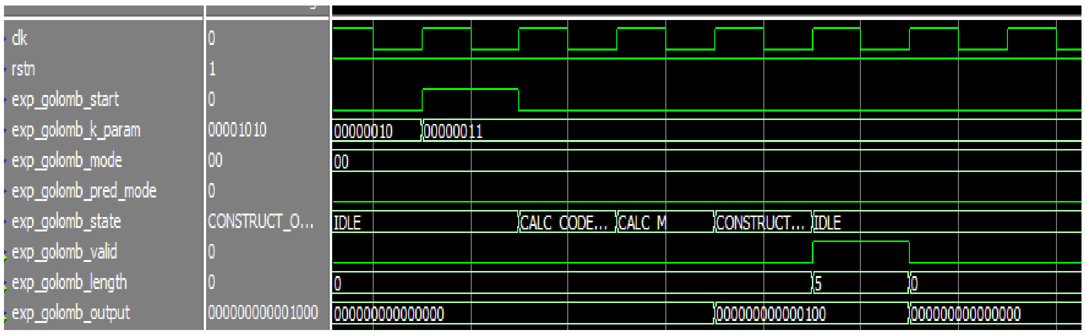


Figure 6.5 Simulation Result When the Mapping Type is ue(v)

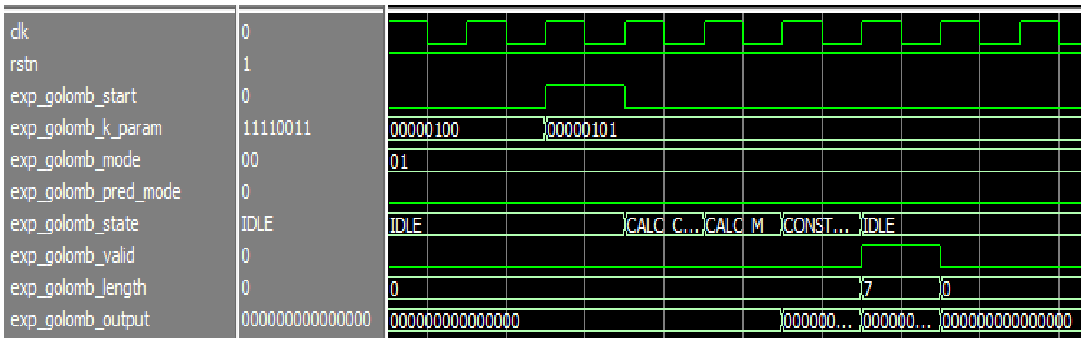


Figure 6.6 Simulation Result When the Mapping Type is se(v)

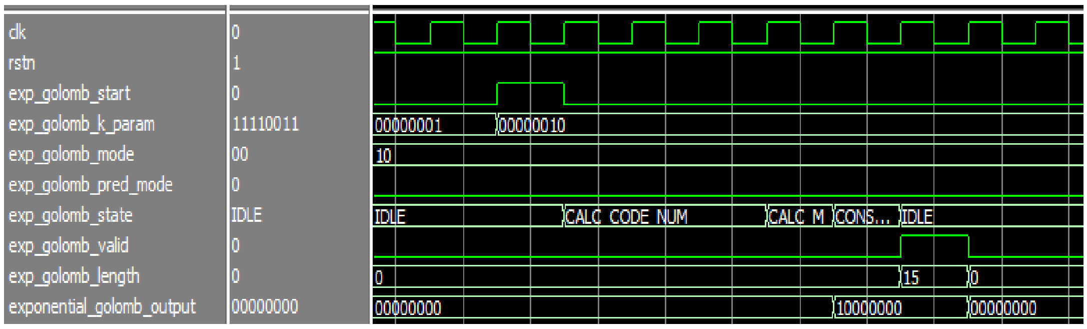


Figure 6.7 Simulation Result When the Mapping Type is me(v)

Table 6.5 Exp-Golomb Block Delay for Different Mapping Types

Mapping Type	Delay(system_clk)
ue(v)	4
se(v)	4
me(v)	6
te(v)	2 or 4

The logic utilization of Exponential-Golomb coding algorithm is given in Table 6.6.

Table 6.6 FPGA Resource Usage of Exp-Golomb Coding Block

Block Name	ALMs	Dedicated Logic Registers	Block Memory Bits	M20Ks	DSP Blocks
Exp-Golomb Coding	100/234720	60/469440	288/52428800	1/2560	0/256

6.5 FPGA Implementation of CAVLC

In this section, circuit design of CAVLC coding is studied. CAVLC block is also written in VHDL language.

The FPGA architecture of CAVLC design is shown in Figure 6.8 and I/O (Input/Output) signals of this block are given in Table 6.7. This block includes several subblocks in order to efficiently generate the H.264 coded bitstream from the quantized coefficients. These blocks are: “Zigzag Scan”, “Scanned Data Control”, “Total Coeff”, “Total Zeros”, “Level Code Control”, “Run Code Control”, “Bitstream Control” and “ROM Blocks”. The details of each block are given in the following sections.

CAVLC coding block starts the coding operation when the cavlc_start signal is high. When this signal is driven high, all the remaining input signals must be set to the true values to code the current quantized coefficients block correctly. If the data at the input ports are ready, firstly the quantized coefficients are zigzag scanned. Then the number of trailing ones, the total number of nonzero coefficients and the total number of zero valued coefficients before the highest nonzero coefficient is calculated by the scanned data control block. Scanned data control block also writes the run and level data to the corresponding FIFOs. Finally, all the VLC parameters are generated by using the scanned data control block and driven as output under the control of bitstream control block.

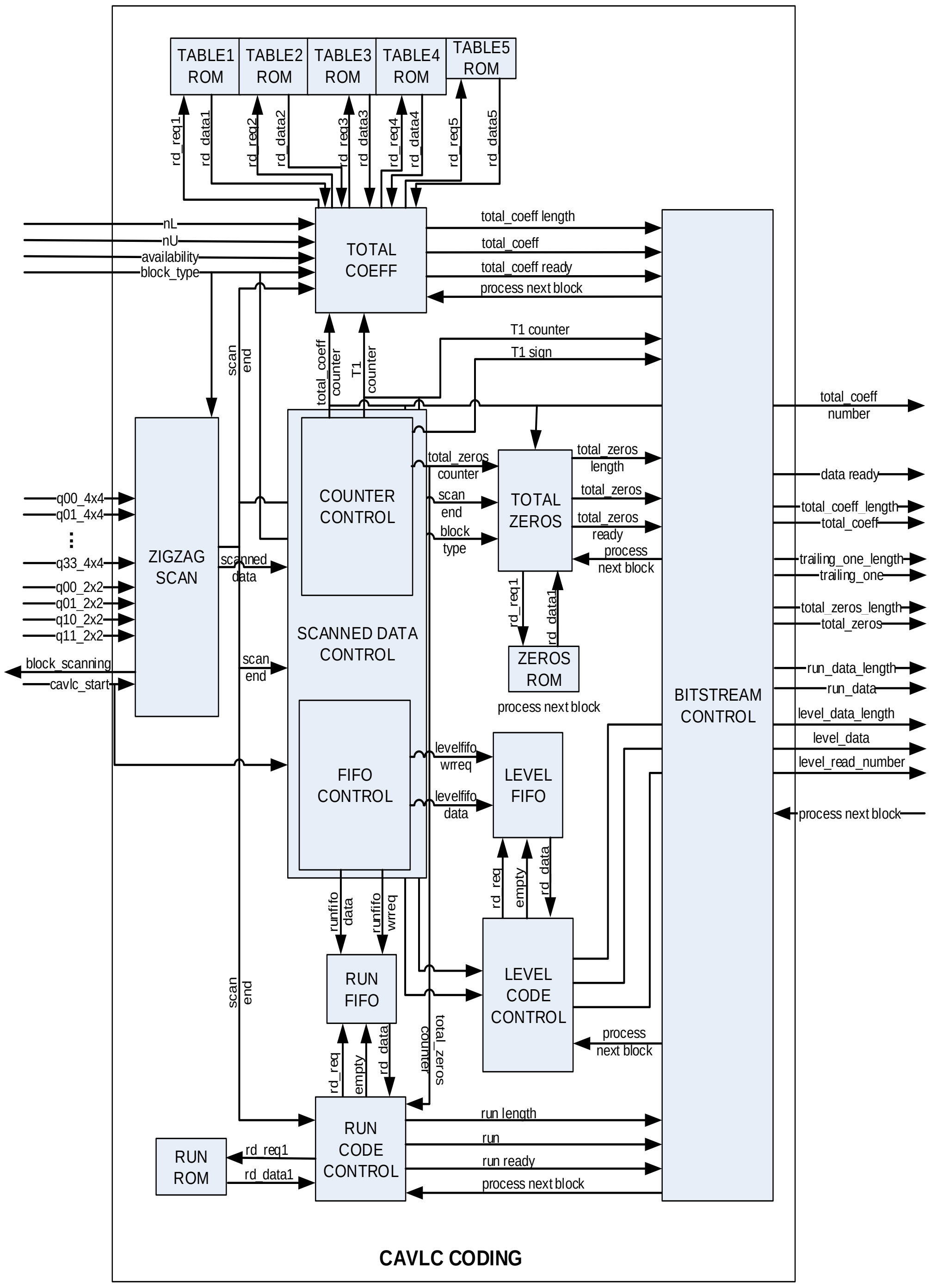


Figure 6.8 FPGA Architecture of CAVLC Block Design

Table 6.7 I/O Signal Descriptions of the CAVLC CODING Block

Sinyal	Length	Description	Input/Output
q00_4x4 q01_4x4 ⋮ q33_4x4	16	Quantized 4x4 Luma, LumaDC or ChromaAC block.	Input
q00_2x2 q01_2x2 q10_2x2 q11_2x2	16	Quantized 2x2 ChromaDC block	Input
cavlc_start	1	When this signal is high, input parameters are valid and start to code quantized input data.	Input
nL	6	Number of nonzero coefficients of the left block.	Input
nU	6	Number of nonzero coefficients of the upper block.	Input
availability	2	Indicates the availabity of the previously coded left and upper blocks with respect to the current block. "00" both of them are not available. "01" left available, upper not. "10" upper available, left not. "11" both of them are available.	Input
block_type	2	Indicates the type of the block. "00" luma4x4 "01" luma4x4AC or chroma4x4AC "10" chromaDC	Input
total_coeff_number	4	Number of nonzero coefficients of the currently coded block.	Output
data_ready	1	Indicates the output data is ready or not. '1' ready, '0' not ready.	Output
process_next_block	1	If high the output data is read by upper block and the coding block can process the next data.	Output
block_scanning	1	If high block scanning process of the current block is not finished yet. Observe this signal before sending new quantized coefficients.	Output
total_coeff_length	5	Length of total_coeff data.	Output
total_coeff	16	Total_coeff data.	Output
trailing_one_length	2	Length of trailing ones data.	Output
trailing_one	3	Trailing ones data.	Output
total_zeros_length	4	Length of total_zeros data.	Output
total_zeros	9	total_zeros data.	Output
run_data_length	5	Length of run data.	Output
run_data	25	Run data.	Output
levelfifo_wrreq	1	Level FIFO write request.	Output
levelfifo_data	33	Level FIFO write data. [27:0] coded level data, [31:28] length of the coded level data.	Output
level_read_number	5	Number of level data written to the level data FIFO.	Output

The design of CAVLC coding block is very critical. It should process a block in limited clock cycles and use acceptable FPGA resources. Otherwise, the working frequency must be increased. This situation is not desired, especially coding an HD (1920x1080) video. In order to meet all these requirements, CAVLC coding block designed to work in a parallel and pipelined manner. The design is separated into three independent blocks (Figure 6.9) which may run in parallel. While scanning a quantized block, another quantized block can be processed or the results can be sent to the output ports. By doing so, the block delay is minimized without increasing logic usage.

The delay of each step varies because the number of coefficients that will be coded changes from one block to another. Coding a 2x2 block requires less clock cycles than coding a 4x4 clock, or coding a block that contains more zero coefficients will need less clock cycles than coding a block that contains less number of zero coefficients. In these cases, coding delay of a block will be the greatest one of all the three delay values.

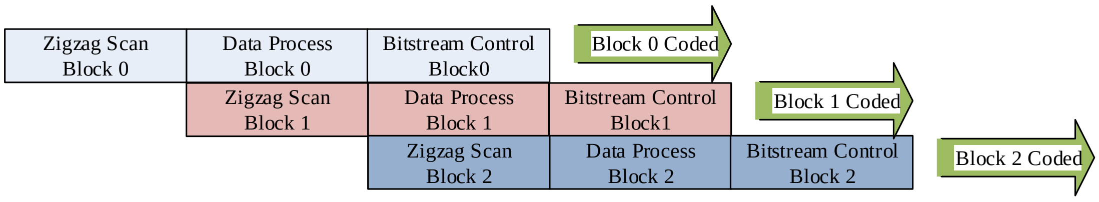


Figure 6.9 CAVLC Coding Block Working Principle

CAVLC coding methodology of a 2x2 block, 4x4 AC block or 4x4 DC block are very similar. There are small differences while processing these blocks. The implemented CAVLC coding block process all type of blocks by using same FPGA circuit. This feature reduces the logic usage. The block type of the current block is chosen by using the block_type input signal.

In the following sections, the details of CAVLC coding block are given.

6.5.1 Zigzag Scan

This block scans the 4x4 and 2x2 blocks of quantized coefficients in the order shown in Table 6.8 and Table 6.9, respectively. In order to start the scanning process, cavlc_start signal must be high and block_scanning signal must be low. After completing the block scanning operation, scan_end signal is set to high in order to declare the end of scanning process to the other blocks.

Table 6.8 4x4 Zigzag Scan Order

idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
zigzag	c00	c01	c10	c20	c11	c02	c03	c12	c21	c30	c31	c22	c13	c23	c32	c33

Table 6.9 4x4 Zigzag Scan Order

idx	0	1	2	3
zigzag	c00	c01	c10	c11

6.5.2 Scanned Data Control

This block is composed of two subblocks: “Counter Control” and “FIFO Control”.

“Counter Control” block counts the number of nonzero coefficients, the number of trailing ones and the number of zeros before the highest frequency nonzero coefficient in a quantized block.

“FIFO Control” block writes the level and run values of a scanned block to the corresponding FIFOs to be processed later by the level code control and run code control blocks.

6.5.3 Total Coeff

This block reads the total_coeff parameter from the ROMs by using the total_coeff_counter and T1_counter. The total_coeff parameter is stored in five different ROMs. Encoder chooses one of them by using the signals nL and nU.

6.5.4 Total Zeros

This block reads the total_zeros parameter from the ROM by using the total_coeff_counter and total_zeros_counter signals.

6.5.5 Run Code Control

Firstly, this block reads each run value in block from the run FIFO. Then, the corresponding codeword of each run value are read from the RUN ROM block. Finally, all codeword are combined and sent to the bitstream control block. The maximum number of required bits to code all run values are 28. So, the length of the output signal is 33. First 28 bits represents the run code and the next 5 bits indicates its length.

6.5.6 Level Code Control

This block reads the level signals from the level FIFO and constructs the corresponding level code. In order to start the coding of level values, total number of nonzero coefficients (total_coeff_counter) and the total number of trailing ones must be known. For this reason, this block must wait the end of the block scanning operation.

In Figure 6.10, the algorithmic flow in order to code the level parameters is shown. During the implementation of this algorithm, parallel processing architecture of the FPGA devices is used and a structure shown in Figure 6.11 is implemented. When we look at this figure, most of the operations are achieved in parallel which reduces the processing time.

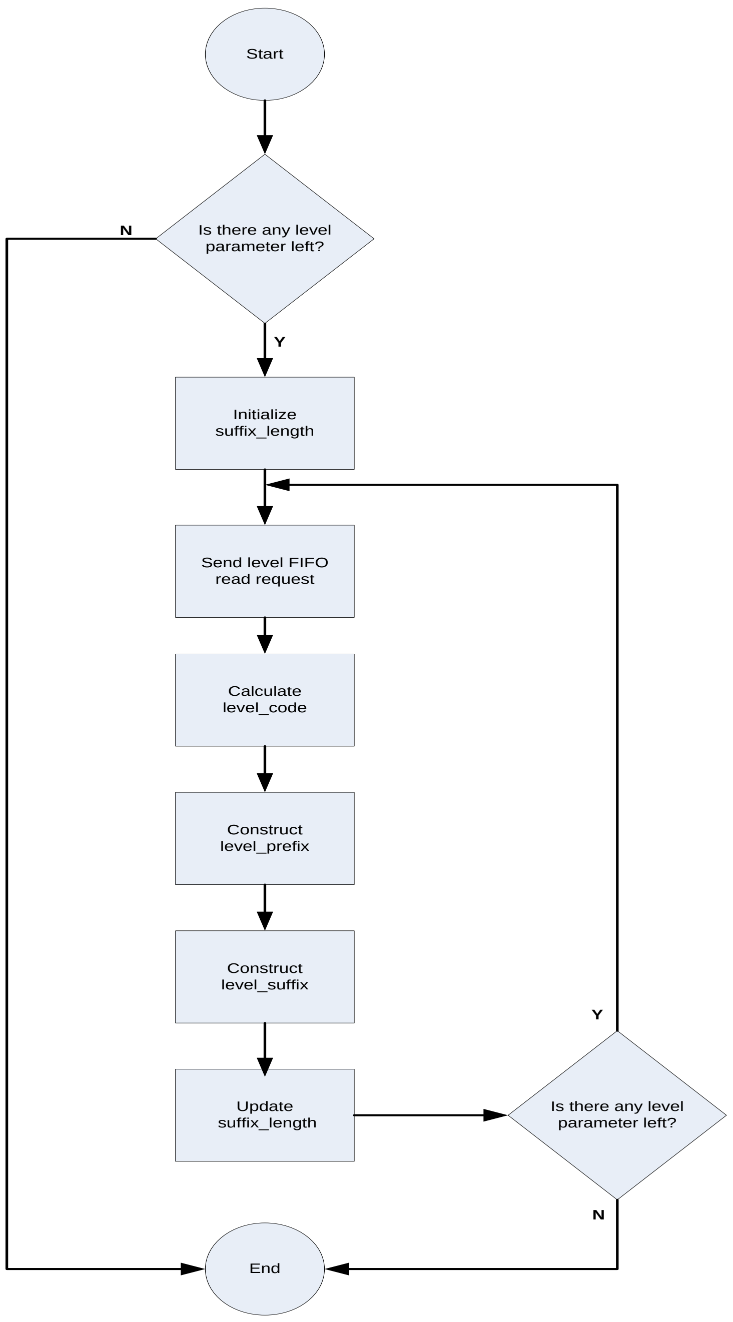


Figure 6.10 Algorithmic Flow of Level Parameter Coding

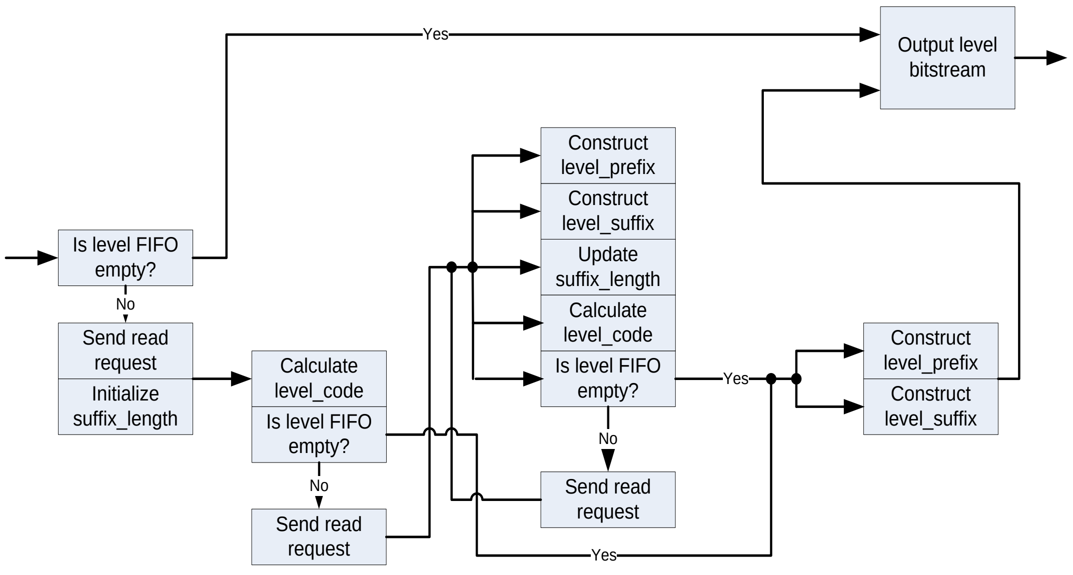


Figure 6.11 FPGA Implementation Architecture of Level Parameter Coding

6.5.7 Bitstream Control

This block reads all the VLC parameters from others blocks and sets the data_ready flag when all the data is ready.

6.5.8 Simulation and Implementation Results of CAVLC Coding Block

In Figure 6.12 and Figure 6.13, Modelsim simulation results of CAVLC coding implementation are given. In Figure 6.12, the coded block size is 4x4 (block_type is 00) and in Figure 6.13 block size is 2x2 (block_type is 10).

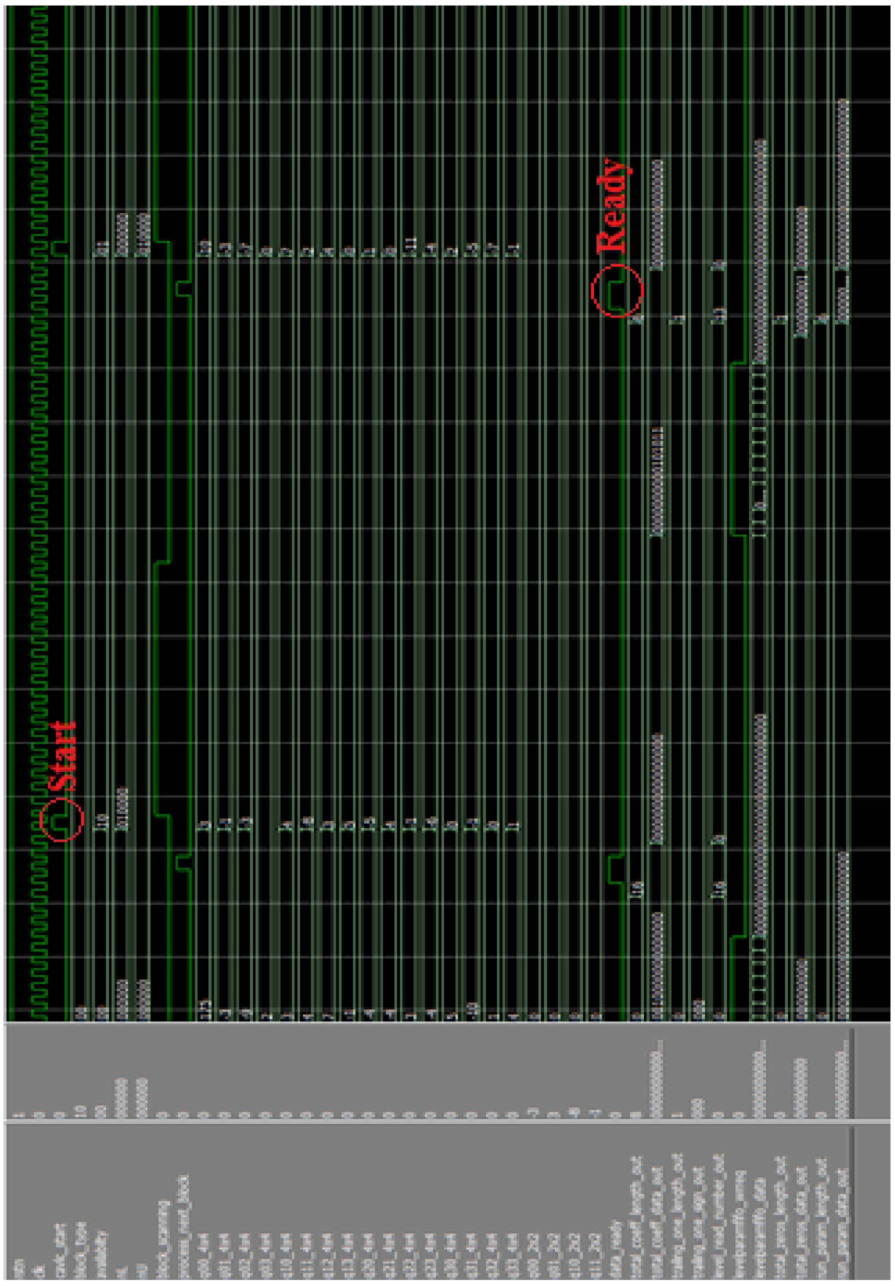


Figure 6.12 CAVLC Modelsim Simulation Result (Block Size: 4x4)

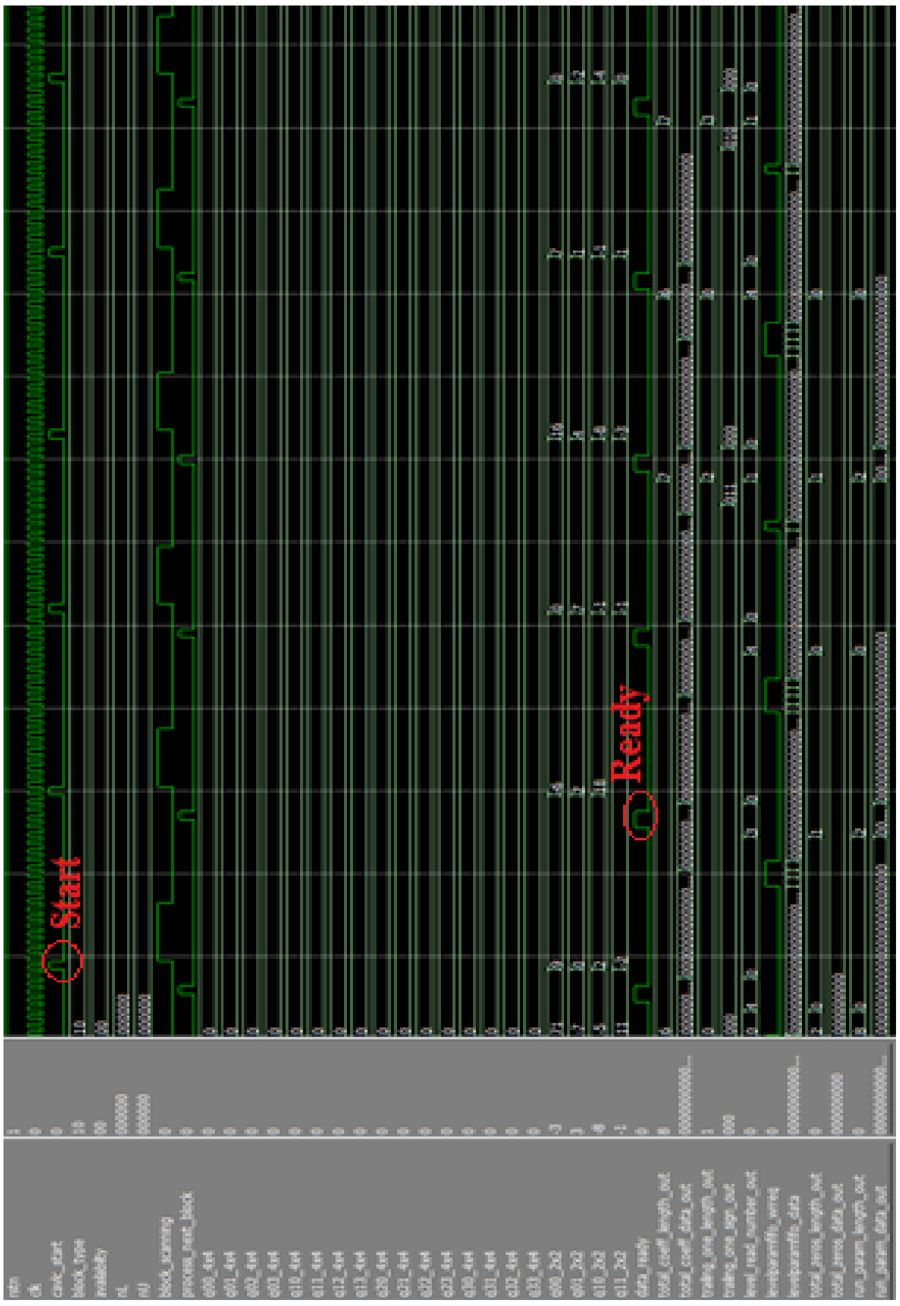


Figure 6.13 CAVLC Modelsim Simulation Result (Block Size: 2x2)

As mentioned earlier, CAVLC coding delay varies with respect to the block type (luma4x4, luma16x16 AC, luma16x16 DC, chroma AC, chroma DC) and the content of the block (the number of zero valued coefficients, the number of trailing ones etc.). In Figure 6.14, Figure 6.15 and Figure 6.16, the worst case delays are given when the maximum numbers of coefficients in a block are 16, 15 and 4, respectively.

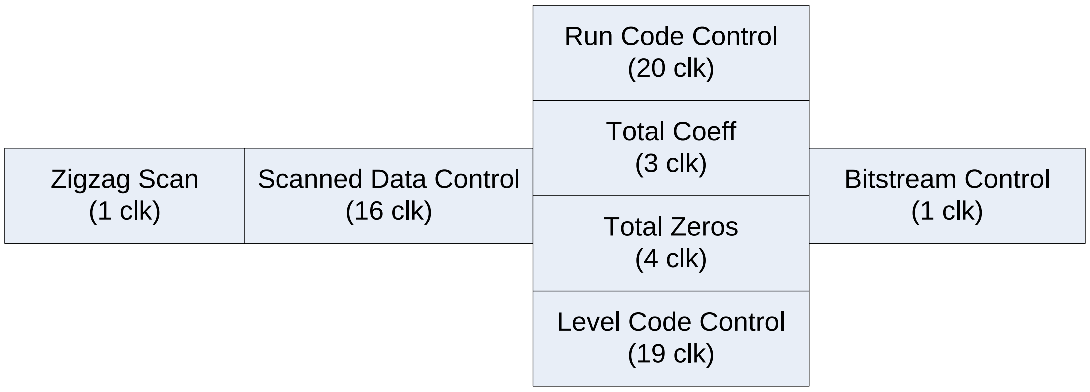

Run Code Control (20 clk)
Zigzag Scan (1 clk)
Scanned Data Control (16 clk)
Total Coeff (3 clk)
Total Zeros (4 clk)
Bitstream Control (1 clk)
Level Code Control (19 clk)

Figure 6.14 Maximum Block Delays (Number of Quantized Coefficient: 16)

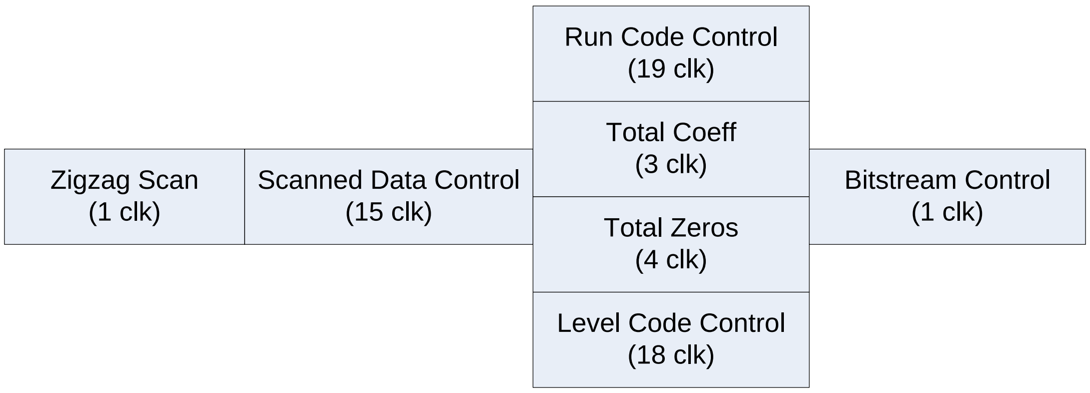

Run Code Control (19 clk)
Zigzag Scan (1 clk)
Scanned Data Control (15 clk)
Total Coeff (3 clk)
Total Zeros (4 clk)
Bitstream Control (1 clk)
Level Code Control (18 clk)

Figure 6.15 Maximum Block Delays (Number of Quantized Coefficient: 15)

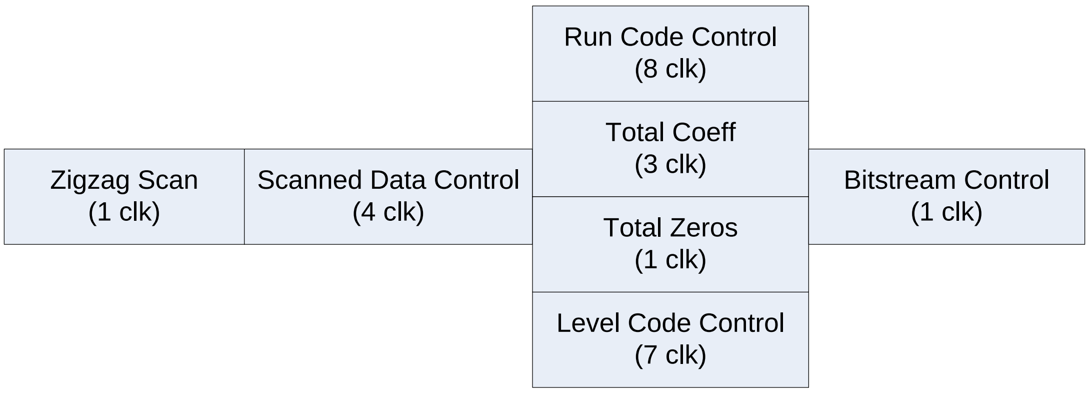


Figure 6.16 Maximum Block Delays (Number of Quantized Coefficient: 4)

The logic utilization of CAVLC coding algorithm is given in Table 6.10.

Table 6.10 FPGA Resource Usage of CAVLC Coding Block

Block Name	ALMs	Dedicated Logic Registers	Block Memory Bits	M20Ks	DSP Blocks
CAVLC	1460/234720	512/469440	320/52428800	2/2560	1/256

CHAPTER 7

HARDWARE IMPLEMENTATION AND RESULTS

This chapter discusses the test setup implementation and test results obtained from the designed H.264 hardware encoder.

7.1 Test Setup

In order to test the designed H.264 encoder on hardware, the block diagram shown in Figure 7.1 is implemented. In this implementation, a video is captured from a camera. Then, this captured video is received, encoded and transmitted from the FPGA in real time. The encoded video is decoded on a standard PC by using VLC Player software.

The proposed architecture is implemented in VHDL language and tested by using the ALTERA STRATIX V FPGA development board shown in Figure 7.2. In this board, there is a STRATIX V 5SGXEA7C2F40 FPGA, SDRAMs, Ethernet PHY and two HSMC (High Speed Mezzanine Connector) connectors. In order to take the sensor data into FPGA board, one more board is used. This board receives camera link signals and sends to FPGA board by using one of the HSMC connectors. The project is analyzed, synthesized, placed and routed to the STRATIX V FPGA using ALTERA QUARTUS 13.1 software.

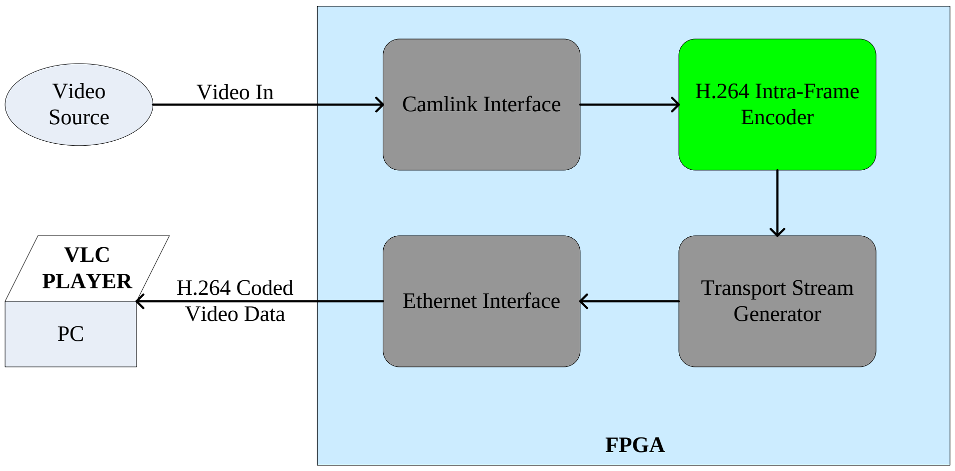


Figure 7.1 H.264 Encoder Hardware Block Diagram

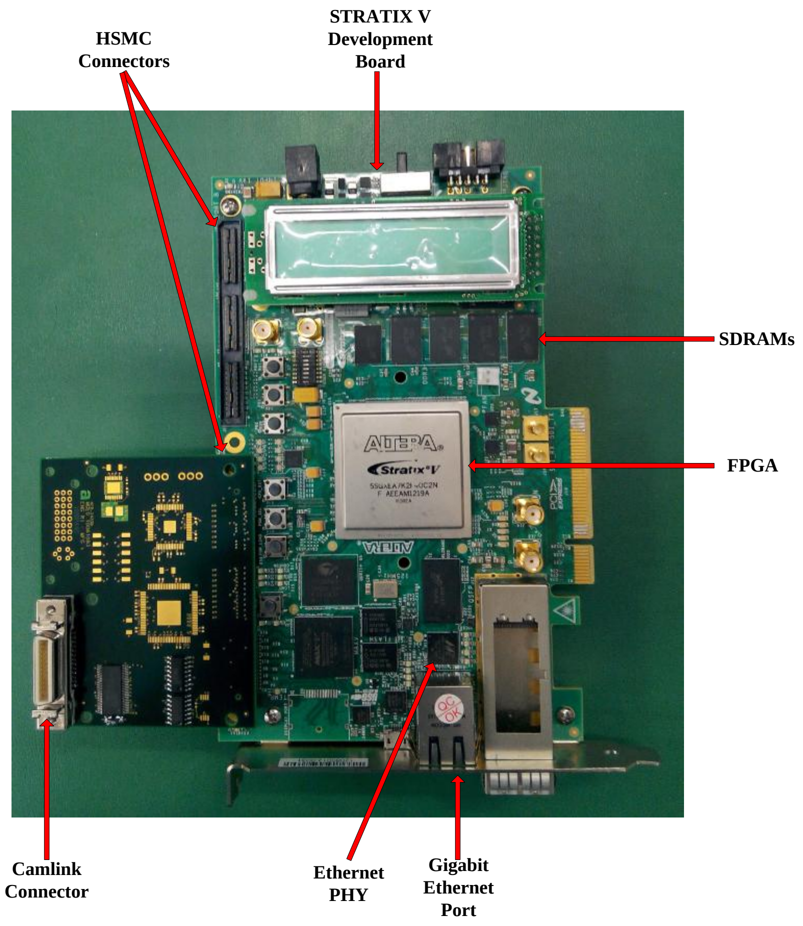


Figure 7.2 STRATIX V FPGA Development Board

7.1.1 Video Source

The camera shown in Figure 7.3 is used as the video source. This is a commercial camera and outputs a video of resolution 640x512 at 25 frames per second.

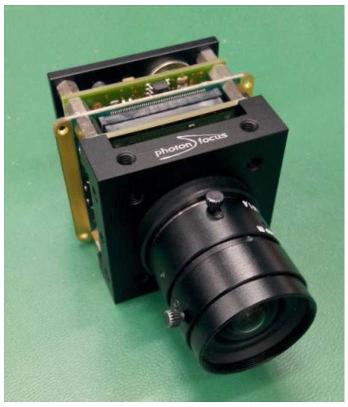


Figure 7.3 Video Source

7.1.2 Camera Link Interface

This FPGA block is a bridge between the camera sensor and video encoder. It analyses the camera link [11] signals and converts these signals into a format that is appropriate to the encoder input.

This block is fully implemented during this thesis work to implement the test set up shown in Figure 7.1.

7.1.3 H.264 Encoder

This block encodes video at the input ports. The output data length is 32 which means the coded data is sent in a four byte aligned format. This block is completely implemented during this thesis work.

7.1.4 Transport Stream Generator

This block generates Transport Stream (TS) packets from the coded video data. An existing VHDL code for this block is modified in order to implement the test setup shown in Figure 7.1.

7.1.5 Ethernet Interface

This block takes Transport Stream packets, encapsulates them into UDP (User Datagram Protocol) packets and sends the UDP packets over the gigabit Ethernet interface. UDP protocol is commonly applied in real-time applications, because dropping packets is preferable to waiting delayed packets in time-sensitive applications.

This block is fully implemented during this thesis work to implement the test set up shown in Figure 7.1.

7.2 Results

In Figure 7.4, our encoder model compression results are compared with High Complexity and Low Complexity modes of JM 18.4 reference software. In this figure, x axis and y axis represents bitrate and PSNR values, respectively. In this comparison, 13 QCIF frames (foreman test sequence) are coded in all modes using several QP values.

In the high complexity mode, prediction results of all the prediction modes are transformed, quantized, inverse transformed, inverse quantized and entropy coded.

Mode selection is done by using the results of these operations. As shown from Figure 7.4, using this technique increases compression efficiency; however, the complexity of the encoder increases much more. Implemented encoder architecture is similar to the low complexity mode of the reference software. In both implementations, mode selection is done just after the block prediction operation and same algorithms are used in the mode selection process. But in low complexity mode many algorithms, such as adaptive rounding, RDO-based (Rate Distortion Optimization) quantization etc., may be used which is not implemented in our encoder.

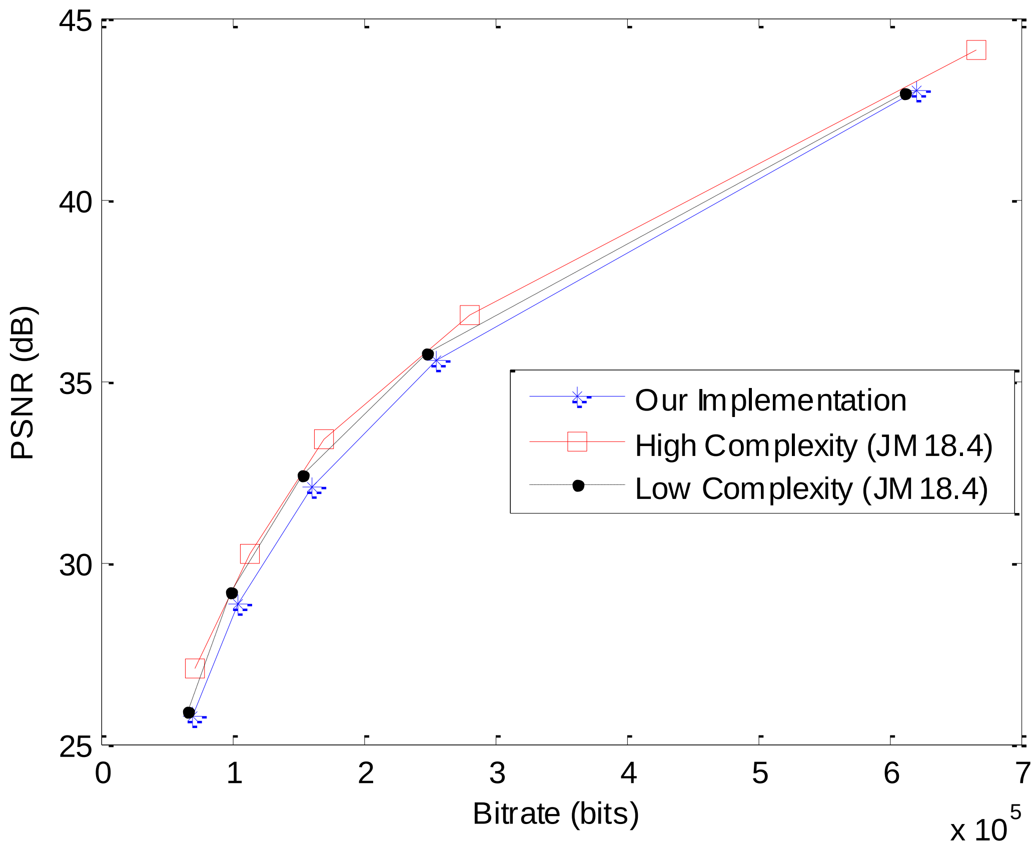


Figure 7.4 Comparison of Implementation Results

In Table 7.1, bit reductions of "High Complexity Mode" are compared with the "Low Complexity Mode" and "Our Implementation" results at 3 different qualities (28 dB, 35 dB, 40 dB). This table shows that "High Complexity Mode" results with 2.6% bitrate reduction when compared to "Low Complexity Mode" and 7.3% bitrate reduction when compared to "Our Implementation" on the average.

Table 7.1 Bitrate Comparison of Three Encoder Models

	28 dB		35 dB		40 dB	
	Low Complexity	Our Implementation	Low Complexity	Our Implementation	Low Complexity	Our Implementation
High Complexity	2.46%	8.80%	2.16%	7.60%	3.14%	5.39%

Implemented H.264 encoder has a low delay. It is about 1 ms which is almost negligible. The resource usage summary of the implemented H.264 encoder design is given in Table 7.2. All video resolutions and frame rates defined in H.264 standard are supported by the implemented encoder. Finally, the maximum frequency used in FPGA fabric is equal to the component frequency of the input video.

Table 7.2 Resource Usage of Implemented H.264 Encoder

	Used	Available
Logic Usage (ALM)	~19000 (~8%)	234720
DSP Block	17(<7%)	256
Memory Bits (Mbits)	0.96(<2%)	50
PLL	0(0%)	28

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In this thesis work, an H.264 compliant intra frame coder hardware has been implemented on FPGA devices targeting all levels of baseline profile. First, a reference encoder which includes encoding algorithms such as intra prediction, intra mode selection, transform, quantization and entropy coding, are implemented and tested in MATLAB environment. Then, the reference encoder is coded in VHDL language and tested using the Mentor Graphics Modelsim HDL simulation tool. Next, the overall FPGA implementation is tested by putting the H.264 coded bitstream into transport stream packets, streaming with UDP over Ethernet and decoding with VLC Player software on a PC. During the hardware verification process, a 640x512 video at 25 frames per second is coded on Altera Stratix V FPGA development kit.

We can summarize the critical tasks that we have faced during the implementation stage and our contributions to these tasks as follows:

- Storing the quantized block data in FIFOs allows modules to remain weakly coupled. This made the design and test of our encoder easier. The quantized coefficients are stored in a FIFO. CAVLC block reads these data and generates the coded block data.

- An encoder codes different parameters and generates H.264 compliant bitstream which must be byte aligned. Our encoder outputs the coded data in 32 bit format. Firstly, we combined all the coded parameters in an 8 bit format but this created timing problems when the bitrate is high (QP is smaller than 10). So we have processed the coded parameters in 16 bit format rather than 8 bit. Resource usage increased a bit; however, our encoder now can encode high bitrate video data.
- While implementing an encoder on the hardware, the target application and video properties are very important. For example, coding a QCIF video will be completely different than coding an HD video. We can make more improvements in QCIF video if we desire to lower the resource usage or power. Because the pixel frequency of the QCIF video is very small when compared to the HD video.
- In our encoder design, about 50 percent of all the used FPGA resources are consumed by the intra mode selection block. This represents that, mode selection stage is the most challenging part while designing an encoder.

8.2 Future Work

Several improvements can be made to our design:

- An efficient intra mode selection algorithm can be developed and implemented to decrease logic usage.
- Inter frames can be added to increase the encoder performance. Correlation between frames in a video (inter prediction) is higher than correlation between pixels in a frame (intra prediction). So using inter-coded frames commonly increases coding efficiency.
- CABAC algorithm can be used instead of CAVLC. CABAC algorithm can represent the same video data 10% fewer bits when compared to CAVLC [5].

- Deblocking filter can be applied to reduce blocking artifacts.
- A constant bitrate algorithm can be implemented for the applications which desire a specific value of bitrate.
- The implemented design can be modified as an ASIC implementation.
- The power consumption of the implemented design can be analyzed and some techniques can be applied to reduce its power consumption.

REFERENCES

[1] International Standard Organization and Information Technology-Coding of Audio-Visual Objects. Part10-Advanced Video Coding. ISO/IEC 14496-10.

[2] http://en.wikipedia.org/wiki/H.264/MPEG-4_AVC, last access: 12 April 2014

[3] P. List, A. Joch, J. Lainema, G. Bjontegaard and M. Karczewicz, 'Adaptive Deblocking Filter', IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 7, July 2003, pp. 614–619.

[4] N. Ahmed, T. Natarajan, and R. Rao, "Discrete Cosine Transform" IEEE Transactions on Computers, vol. C-23, pp. 90-93, Jan. 1974.

[5] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi, "Video Coding with H.264/AVC: Tools, Performance, and Complexiy", IEEE Circuits and Systems Magazine, 2004.

[6] I. E. Richardson (2010). H.264 Syntax. In: The H.264 Advanced Video Compression Standard. 2nd ed. UK: John Wiley&Sons Ltd. 100-134

[7] O. and K. Ramchandran , "Rate-distortion methods for image and video compression, " IEEE Signal Processing Magazine, vol. 15, no.6, pp. 23-50, Nov. 1998.

[8] G.J. Sullivan and T. Wiegand, "Rate-distortion optimization for video compression," IEEE Signal Processing Magazine, vol. 15, pp. 74-90, Nov. 1998.

[9] I. E. Richardson (2010). Context Adaptive Variable Length Coding. In: The H.264 Advanced Video Compression Standard. 2nd ed. UK: John Wiley&Sons Ltd. 100-134.

[10] http://en.wikipedia.org/wiki/H.264/Camera_Link, last access: 3 August 2014.

[11] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-ComplexityTransform and Quantization in H.264/AVC," IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 7, July 2003.

[12] T. Wiegand, G. J. Sullivan, G. Bjontegaard, A. Luthra, "Overview of the H.264/AVC Video Coding Standard," IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 7, July 2003.

[13] Y. Katayama, "Protection From IDCT Mismatch," Tech. Rep. MPEG 93/283, ISO/IEC JTC1/SC2/WG11, 1993.

[14] "Video and Image Processing Design Using FPGAs (white paper)," ALTERA WP-VIDEO0306-1.1 March 2007.

[15] B. Carle, "How Does H.264 Work? (white paper)," Salient Systems Corp.

[16] I. E. Richardson, "4x4 Transform and Quantization in H.264/AVC (white paper)," Vcodex Limited,Ver. 1.2, November 2010.

[17] I. E. Richardson, "H.264/AVC Context Adaptive Variable Length Coding (white paper)," Vcodex, 2002-2010.

[18] I. E. Richardson, "An Overview of H.264 Advanced Video Coding (white paper)," Vcodex/OneCodec, 2007-2011.

[19] I. E. Richardson, "H.264/AVC Intra Prediction (white paper)," Vcodex, 2002 2011.

[20] I. E. Richardson, "H.264/MPEG-4 Part 10: Transform&Quantization (white paper)," Vcodex, 2003.

[21] M. Tikekar, II 2012, 'Circuit Implementations for High-Efficiency Video Coding Tools', Massachusetts Institute of Technology, Boston.

[22] E. Şahin, II 2006, 'An Efficient H.264 Intra Frame Coder Hardware Design', Sabancı University, İstanbul.

[23] JM reference software version 18.4, http://iphome.hhi.de/suehring/tml/, July 2013

[24] S. W. Golomb, 'Run-length encoding', IEEE Transactions on Information Theory, vol. IT-12, pp. 399–401, 1966.

[25] S. Nargundmath, A. Nandibewoor, "Entropy Coding of H.264/AVC using Exp-Golomb Coding and CAVLC Coding," ICANMEET-2013, July 2013.

[26] W. Di, G. Wen, H. Mingzeng, and J. Zhenzhou, "An Exp-Golomb Coder and Decoder Architecture for JVT/AVS," IEEE Trans. On Circuits and Systems for Video Technology, vol.2, n.21-24, p. 910-913, 2003.

APPENDIX A

EXPONENTIAL- GOLOMB AND CAVLC CODING TABLES

Table A-1 CBP Table (ChromaArrayType is Equal to 1 or 2)

codeNum	coded_block_pattern	
	Intra_4x4, Intra_8x8	Inter
0	47	0
1	31	16
2	15	1
3	0	2
4	23	4
5	27	8
6	29	32
7	30	3
8	7	5
9	11	10
10	13	12
11	14	15
12	39	47
13	43	7
14	45	11
15	46	13
16	16	14
17	3	6
18	5	9
19	10	31
20	12	35
21	19	37
22	21	42
23	26	44
24	28	33
25	35	34
26	37	36
27	42	40
28	44	39
29	1	43
30	2	45
31	4	46
32	8	17
33	17	18
34	18	20
35	20	24
36	24	19
37	6	21
38	9	26
39	22	28
40	25	23
41	32	27
42	33	29
43	34	30
44	36	22
45	40	25
46	38	38
47	41	41

Table A-2 CBP Table (ChromaArrayType is Equal to 0 or 3)

codeNum	coded_block_pattern	
	Intra_4x4, Intra_8x8	Inter
0	15	0
1	0	1
2	7	2
3	11	4
4	13	8
5	14	3
6	3	5
7	5	10
8	10	12
9	12	15
10	1	7
11	2	11
12	4	13
13	8	14
14	6	6
15	9	9

Table A-3 Total Coeff Tables

TrailingOnes (coeff_token)	TotalCoeff (coeff_token)	Table 1 0 <= nC < 2	Table 2 2 <= nC < 4	Table 3 4 <= nC < 8	Table 4 8 <= nC	Table 5 nC = = −1	Table 6 nC = = −2
0	0	1	11	1111	0000 11	01	1
0	1	0001 01	0010 11	0011 11	0000 00	0001 11	0001 111
1	1	01	10	1110	0000 01	1	01
0	2	0000 0111	0001 11	0010 11	0001 00	0001 00	0001 110
1	2	0001 00	0011 1	0111 1	0001 01	0001 10	0001 101
2	2	001	011	1101	0001 10	001	001
0	3	0000 0011 1	0000 111	0010 00	0010 00	0000 11	0000 0011 1
1	3	0000 0110	0010 10	0110 0	0010 01	0000 011	0001 100
2	3	0000 101	0010 01	0111 0	0010 10	0000 010	0001 011
3	3	0001 1	0101	1100	0010 11	0001 01	0000 1
0	4	0000 0001 11	0000 0111	0001 111	0011 00	0000 10	0000 0011 0
1	4	0000 0011 0	0001 10	0101 0	0011 01	0000 0011	0000 0010 1
2	4	0000 0101	0001 01	0101 1	0011 10	0000 0010	0001 010
3	4	0000 11	0100	1011	0011 11	0000 000	0000 01
0	5	0000 0000 111	0000 0100	0001 011	0100 00	-	0000 0001 11
1	5	0000 0001 10	0000 110	0100 0	0100 01	-	0000 0001 10
2	5	0000 0010 1	0000 101	0100 1	0100 10	-	0000 0010 0
3	5	0000 100	0011 0	1010	0100 11	-	0001 001
0	6	0000 0000 0111 1	0000 0011 1	0001 001	0101 00	-	0000 0000 111
1	6	0000 0000 110	0000 0110	0011 10	0101 01	-	0000 0000 110
2	6	0000 0001 01	0000 0101	0011 01	0101 10	-	0000 0001 01
3	6	0000 0100	0010 00	1001	0101 11	-	0001 000
0	7	0000 0000 0101 1	0000 0001 111	0001 000	0110 00	-	0000 0000 0111
1	7	0000 0000 0111 0	0000 0011 0	0010 10	0110 01	-	0000 0000 0110
2	7	0000 0000 101	0000 0010 1	0010 01	0110 10	-	0000 0000 101
3	7	0000 0010 0	0001 00	1000	0110 11	-	0000 0001 00
0	8	0000 0000 0100 0	0000 0001 011	0000 1111	0111 00	-	0000 0000 0011
1	8	0000 0000 0101 0	0000 0001 110	0001 110	0111 01	-	0000 0000 0101
2	8	0000 0000 0110 1	0000 0001 101	0001 101	0111 10	-	0000 0000 0100
3	8	0000 0001 00	0000 100	0110 1	0111 11	-	0000 0000 100
0	9	0000 0000 0011 11	0000 0000 1111	0000 1011	1000 00	-	-
1	9	0000 0000 0011 10	0000 0001 010	0000 1110	1000 01	-	-
2	9	0000 0000 0100 1	0000 0001 001	0001 010	1000 10	-	-
3	9	0000 0000 100	0000 0010 0	0011 00	1000 11	-	-
0	10	0000 0000 0010 11	0000 0000 1011	0000 0111 1	1001 00	-	-
1	10	0000 0000 0010 10	0000 0000 1110	0000 1010	1001 01	-	-
2	10	0000 0000 0011 01	0000 0000 1101	0000 1101	1001 10	-	-
3	10	0000 0000 0110 0	0000 0001 100	0001 100	1001 11	-	-
0	11	0000 0000 0001 111	0000 0000 1000	0000 0101 1	1010 00	-	-
1	11	0000 0000 0001 110	0000 0000 1010	0000 0111 0	1010 01	-	-
2	11	0000 0000 0010 01	0000 0000 1001	0000 1001	1010 10	-	-
3	11	0000 0000 0011 00	0000 0001 000	0000 1100	1010 11	-	-
0	12	0000 0000 0001 011	0000 0000 0111 1	0000 0100 0	1011 00	-	-
1	12	0000 0000 0001 010	0000 0000 0111 0	0000 0101 0	1011 01	-	-
2	12	0000 0000 0001 101	0000 0000 0110 1	0000 0110 1	1011 10	-	-
3	12	0000 0000 0010 00	0000 0000 1100	0000 1000	1011 11	-	-
0	13	0000 0000 0000 1111	0000 0000 0101 1	0000 0011 01	1100 00	-	-
1	13	0000 0000 0000 001	0000 0000 0101 0	0000 0011 1	1100 01	-	-
2	13	0000 0000 0001 001	0000 0000 0100 1	0000 0100 1	1100 10	-	-
3	13	0000 0000 0001 100	0000 0000 0110 0	0000 0110 0	1100 11	-	-
0	14	0000 0000 0000 1011	0000 0000 0011 1	0000 0010 01	1101 00	-	-
1	14	0000 0000 0000 1110	0000 0000 0010 11	0000 0011 00	1101 01	-	-
2	14	0000 0000 0000 1101	0000 0000 0011 0	0000 0010 11	1101 10	-	-
3	14	0000 0000 0001 000	0000 0000 0100 0	0000 0010 10	1101 11	-	-
0	15	0000 0000 0000 0111	0000 0000 0010 01	0000 0001 01	1110 00	-	-
1	15	0000 0000 0000 1010	0000 0000 0010 00	0000 0010 00	1110 01	-	-
2	15	0000 0000 0000 1001	0000 0000 0010 10	0000 0001 11	1110 10	-	-
3	15	0000 0000 0000 1100	0000 0000 0000 1	0000 0001 10	1110 11	-	-
0	16	0000 0000 0000 0100	0000 0000 0001 11	0000 0000 01	1111 00	-	-
1	16	0000 0000 0000 0110	0000 0000 0001 10	0000 0001 00	1111 01	-	-
2	16	0000 0000 0000 0101	0000 0000 0001 01	0000 0000 11	1111 10	-	-
3	16	0000 0000 0000 1000	0000 0000 0001 00	0000 0000 10	1111 11	-	-

Table A-4 Total Zeros Table for 4x4 Blocks

total_zeros	TotalCoeff														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	111	0101	0001 1	0101	0000 01	0000 01	0000 01	0000 01	0000 1	0000	0000	000	00	0
1	011	110	111	111	0100	0000 1	0000 1	0001	0000 00	0000 0	0001	0001	001	01	1
2	010	101	110	0101	0011	111	101	0000 1	0001	001	001	01	1	1	-
3	0011	100	101	0100	111	110	100	011	11	11	010	1	01	-	-
4	0010	011	0100	110	110	101	011	11	10	10	1	001	-	-	-
5	0001 1	0101	0011	101	101	100	11	10	001	01	011	-	-	-	-
6	0001 0	0100	100	100	100	011	010	010	01	0001	-	-	-	-	-
7	0000 11	0011	011	0011	011	010	0001	001	0000 1	-	-	-	-	-	-
8	0000 10	0010	0010	011	0010	0001	001	0000 00	-	-	-	-	-	-	-
9	0000 011	0001	0001	0010	0000 1	001	0000 00	-	-	-	-	-	-	-	-
10	0000 010	0001	0001	0001 0	0001	0000 00	-	-	-	-	-	-	-	-	-
11	0000 0011	0000	0000	0000 1	0000 0	-	-	-	-	-	-	-	-	-	-
12	0000 0010	0000	0000	0000 0	-	-	-	-	-	-	-	-	-	-	-
13	0000 0001 1	0000	0000	-	-	-	-	-	-	-	-	-	-	-	-
14	0000 0001 0	0000	-	-	-	-	-	-	-	-	-	-	-	-	-
15	0000 0000 1	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A-5 Total Zeros Table for Chroma DC Blocks (4:2:0 Chroma Sampling)

total_zeros	TotalCoeff		
	1	2	3
0	1	1	1
1	01	01	0
2	001	00	-
3	000	-	-

Table A-6 Total Zeros Table for Chroma DC Blocks (4:2:2 Chroma Sampling)

total_zeros	TotalCoeff						
	1	2	3	4	5	6	7
0	1	000	000	110	00	00	0
1	010	01	001	00	01	01	1
2	011	001	01	01	10	1	-
3	0010	100	10	10	11	-	-
4	0011	101	110	111	-	-	-
5	0001	110	111	-	-	-	-
6	0000 1	111	-	-	-	-	-
7	0000 0	-	-	-	-	-	-

Table A-7 Run Before Parameter Table

run_before	zerosLeft						
	1	2	3	4	5	6	> 6
0	1	1	11	11	11	11	111
1	0	01	10	10	10	000	110
2	-	00	01	01	011	001	101
3	-	-	00	001	010	011	100
4	-	-	-	000	001	010	011
5	-	-	-	-	000	101	010
6	-	-	-	-	-	100	001
7	-	-	-	-	-	-	0001
8	-	-	-	-	-	-	00001
9	-	-	-	-	-	-	000001
10	-	-	-	-	-	-	0000001
11	-	-	-	-	-	-	00000001
12	-	-	-	-	-	-	000000001
13	-	-	-	-	-	-	0000000001
14		-	-	-	-	-	00000000001