A HIGH-SPEED, LIGHT WEIGHT HARDWARE ARCHITECTURE FOR H.264-COMPATIBLE COMPRESSION ON AN FPGA

by

AZAM TAYYEBI

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN ELECTRICAL AND COMPUTER ENGINEERING

2023

Oakland University
Rochester, Michigan

Doctoral Advisory Committee:

Darrin M. Hanna, Ph.D., Chair
Geoffrey Louie, Ph.D.
Shadi G. Alawneh, Ph.D.
Aycil Cesmelioglu, Ph.D.

To my mother and father

ACKNOWLEDGMENTS

I would like to thank my supervisor Professor Hanna for his support, motivation, guidance, and patience throughout my PhD studies. I really appreciate his invaluable advice and detailed reviews during this work. Also, I would like to thank my committee members, Dr. Geoffrey Louie, Dr. Shadi G. Alawneh, and Dr. Aycil Cesmelioglu for their and reviewing my work. I express my special thanks to my friend and collaborator, Bryant Jones, at Intrepid Control Systems for his support and valuable discussions and advice during the VHDL implementation process and test setup.

I also would like to thank my family for their constant support and encouragement during tough periods of my life. Also special thanks to my husband Matt for his patience and support during my PhD program.

Azam Tayyebi

ABSTRACT

A HIGH-SPEED, LIGHT WEIGHT HARDWARE ARCHITECTURE FOR H.264-COMPATIBLE COMPRESSION ON AN FPGA

by

AZAM TAYYEBI

Adviser: Darrin M. Hanna, Ph.D.

Video compression is a technique that reduces and removes spatial and temporal redundancy of video data, resulting in a reduction in transmission time and communication bandwidth across a network and efficient storage. H.264, a widely adopted video compression standard, is the result of collaborative efforts between the ISO (International Organization for Standardization) Moving Picture Experts Group (MPEG) and the ITU (International Telecommunication Union) Video Coding Experts Group (VCEG). H.264 uses an array of algorithms for coding digital video to achieve better compression efficiency compared to previous standards. However, this efficiency increase comes at the cost of higher computational complexity for H.264 encoders.

This thesis design and implement an H.264-compatible intra-frame video encoder on FPGA. The encoding algorithms, like intra prediction, transform, quantization, and entropy coding, are initially implemented and tested in MATLAB. Later, these algorithms are translated into VHDL language and evaluated through timing simulations

in Vivado. The FPGA implementation is then tested using various input pixel sizes and video resolutions across multiple FPGA devices. The encoder supports all video resolutions and frame rates.

TABLE OF CONTENTS

TABLE OF CONTENTS—Continued

TABLE OF CONTENTS—Continued

TABLE OF CONTENTS—Continued

LIST OF TABLES

LIST OF TABLES—Continued

LIST OF FIGURES

LIST OF FIGURES—Continued

LIST OF FIGURES—Continued

LIST OF ABBREVIATIONS

FPS	Frames per second
QCIF	Quarter Common Intermediate Format ($176 \times 144\ pixels$)
CIF	Common Intermediate Format ($352 \times 240\ pixels$)
HDTV	High-definition television
DVD	Digital video disc
VHDTV	Ultra-high-definition television
ITU	International Telecommunications Union
MPEG	Moving Pictures Experts Group
ISO	International organization for standardization
DCT	Discrete cosine transform
DAB	Digital audio broadcasting
MP3	MPEG-1 Audio Layer 3
MMS	Multimedia Message Services
ASP	Advanced Simple Profile
FPGA	Field Programmable Gate Arrays
HEVC	High Efficiency Video Coding
AV1	*AOMedia Video 1*
VVC	Versatile Video Coding
YUV	Color space
YCbCr	Luminance, blue chrominance, red chrominance color space

MSE	Mean square error
MAD	Mean absolute difference
PSNR	Peak to Signal Noise Ratio
SAD	Sum of absolute difference
QP	Quantization parameter
IDCT	Inverse discrete cosine
CAVLC	Context-adaptive variable-length coding
NAL	Network Abstraction Layer
PC	Personal Computer
VHDL	VHSIC Hardware Description Language
RAM	Random Access Memory

CHAPTER ONE

INTRODUCTION

This chapter introduces the importance of compression in technology and presents a short description and comparison of the various video coding standards.

1.1 The Importance of Video Compression

Digital video consists of a series of digital images (known as frames) displayed in rapid succession. Frame rate is the rate at which consecutive frames are displayed expressed as frames per second (FPS). Every frame is a bitmap image that represents a generally rectangular grid of pixels. Each pixel corresponds to only one value, its color, specified by a fixed number of bits. The more bits, the greater number of colors can be represented which is referred to as the color depth of the video. Table 1.1 compares how the uncompressed data rate increases with several video formats [4].

Table 1-2 Typical digital video formats with their uncompressed data rates with color depth of 12-bits/pixel.

Format	Resolution	Frames/Second	Uncompressed Data Rate (Mbit/S)
QCIF	176 × 144	15	4.6
CIF	352 × 240	30	30.4
ITU-R 601	720 × 480	30	124.4
HDTV	1280 × 720	30	331.8
HDTV	1920 × 1080	30	746.4

Currently, there are a wide variety of digital video applications, such as video game entertainment, broadcast television, internet video streaming, DVDs, 4K camera, ultra-high-definition television (UHDTV) [5], education, and medicine. Streaming a video consumes bandwidth. Recording a video consumes data storage. Table 1.2 lists the capacity of typical storage and transmission formats [6]. The amount of bandwidth or storage required is determined by the frame size, frame rate, and color depth.

Video compression plays a critical role in various aspects of modern digital media and communication. In many applications, video compression is required to:

1. Reduce Storage Requirements: Uncompressed video files can be very large, making them impractical for storage and transmission. Video compression reduces the file size, allowing for efficient storage on devices and servers.
2. Increase Bandwidth Efficiency: Compressed videos require less bandwidth to transmit over the internet or other communication channels. This is particularly important for streaming services, video conferencing, and online content delivery, where limited bandwidth is a common constraint.
3. Enable Faster Transmission: Smaller video files can be transmitted more quickly, reducing buffering times and improving the user experience for streaming, online gaming, and video calls.
4. Reduce Costs: Data storage and transmission costs are directly related to file size. By compressing videos, companies and individuals can save on storage expenses and reduce data transfer costs.

5. Achieve Compatibility: Compressed video formats are often more compatible with a wide range of devices and platforms, ensuring that content can be viewed on various screens and operating systems.

6. Support Live Streaming: Video compression is essential for live streaming events, sports, news broadcasts, and other real-time content delivery, ensuring a smooth viewing experience for viewers worldwide.

7. Enable Mobile Devices: Video compression is crucial for delivering video content to smartphones and tablets, which often have limited storage and varying screen resolutions.

8. Communicate in Real Time: Video compression is essential for video conferencing and telemedicine, enabling clear and fluid communication over the internet.

The process of compressing raw digital video is done by an encoder [7]. A decoder, on the other hand, receives the compressed video and decompresses it back to the raw format.

Table 1-2 Typical transmission/storage capacities

Format	Maximum Capacity
10/100/G Ethernet	10/100/1000 Mb/s
DSL	768kb/s-1.5Mb/s downstream 128kb/s upstream (typical)
V.90 modem	56 kb/s
Single Layer DVD-5	4.7 Gbytes
Dual Layer DVD-9	8.5 Gbytes

Over the past decades many different video compressions standards have been investigated. In the next section we review the most common standards.

1.2 Review of Video Compression Standards

Video compression standards have been established so that a bitstream generated by one manufacturer's encoder can be decoded by another manufacturer's decoder. The standards define the syntax of the encoded bit stream and constraints for transmitted and derived parameters along with the behavior of a compatible decoder. Manufacturers can implement different encoder designs provided the output syntax follows a standard. The market for standardized video codecs was dominated by the International Telecommunications Union (ITU) [8] and the Moving Pictures Experts Group (MPEG) [9] for several decades. The video codec standards developed by ITU includes the series of "H.26x" standards. ISO standards were formed by MPEG in the early 90's. These standards are designated by the label "MPEG-x.". Figure 1-1 shows a timeline of the development of video compression standards [10].

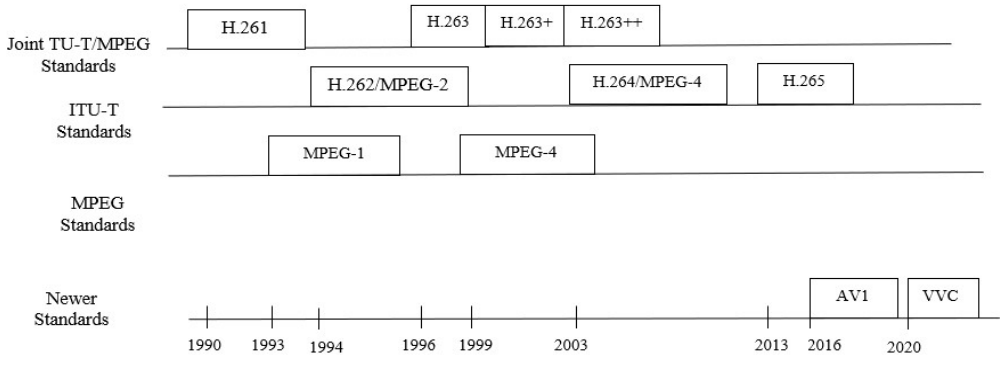


Figure 2-1 The timeline of video coding standards

1.2.1 H.261

H.261 [11] is the first practical video coding standard. It was developed by ITU-T in 1990. The standard supports interactive video communications over ISDN networks with a target data rate of 64 to 2048 Kbit/s. H.261 supports only CIF and QCIF input formats, the video formats used by ITU-T for visual telephony. A key factor for the success of H.261 was low complexity since the standard was developed at a time when hardware capability was limited. A macroblock of 16×16 pixels is the smallest coding unit in H.261. It uses block-based motion estimation for each MB in the encoder, while using a discrete cosine transform (DCT) of size 8×8 for residual difference coding, scalar quantization, and variable-length codes (i.e. Huffman codes) for entropy coding. The next IUT-T standard, H.263, superseded H.261 quickly due to its higher coding efficiency and flexibility.

1.2.2 MPEG-1

MPEG-1 [12] is the first standard for lossy compression of video and associated audio at up to 1.5Mbit/s. This standard, which was built on H.261 to get better quality using a more complex encoding method, was approved in 1993. It was designed primarily for the digital storage of video and audio, making video CDs, digital cable and satellite TV, and digital audio broadcasting (DAB) possible. MPEG-1 is used in many products and technologies, most notably the MP3 audio format.

1.2.3 H.262/MPEG-2

The H.262 [13] standard, also known as MPEG-2, was developed jointly by MPEG and ITU-T and was completed in 1994. It was intended to extend MPEG-1 technology to support full broadcast-quality video at high bitrates ranging from 4 to 30 Mbit/s and interlaced video, an encoding technique used in analog NTSC, PAL, and SECAM television systems. It was designed to be backwards compatible with the MPEG-1 standard, so any MPEG-2 decoder could play MPEG-1 videos. The standard is used broadly for many applications today, such as DVDs, broadcast/satellite TVs, a standard definition professional video recording format (MPEG IMX,) and a tape-based high-definition video recording format (HDV).

1.2.4 H.263

H.263 [14] was developed as a low bit rate compression format for videoconferencing and was completed in 1996. The standard was extended later in 1998 and 2000 to add some additional enhanced features, other smaller additions were made in 1997 and 2001. The standard is used in Multimedia Message Services (MMS) internet applications such as YouTube and Google videos. H.264 is the next enhanced format after H.263.

1.2.5 MPEG-4 Part2

MPEG-4 Part 2 [15] was introduced in 1999. It offers more efficient encoding and video compression compared to previous compression standards with one exception, it cannot provide compression for the AVC format. The standard is H.263 compatible, therefore, a basic H.263 bitstream can be correctly decoded by an MPEG-4 decoder. MPEG-4 Part2 is the preferred standard for Internet multimedia streaming over broadband networks.

Two famous compression algorithms, DivX and Xvid, also use MPEG-4 Part 2. The standard has two profiles, the Simple Profile (SP) and Advanced Simple Profile (ASP). ASP has the innovative quarter-pixel motion compensation feature.

1.2.6 H.264/MPEG-4 Part 10: Advanced Video Coding

H.264/MPEG-4 or H.264/AVC [16] is another joint effort between ITU-T and MPEG. The standard was approved in 2003 and has been enhanced over the years to add new features. In this work, H.264 is used to refer to this standard. The goal of this work is to optimize an H.264 encoder utilizing minimum hardware on an FPGA, compressing at a high speed, and achieving a high compression ratio while compressing raw uncompressed input video.

H.264 is used for many applications such as streaming Internet standards for videos on YouTube, Netflix, Hulu, iTunes, web software Adobe Flash Player and Microsoft Silverlight, direct broadcast satellite services, cable television services, and Blu-ray Discs [17]. This standard provides significantly lower bit rates in comparison to previous standards using flexible macroblock sizing, multiple reference picture capabilities, 1/4-pixel interpolation, and an in-loop filter. H.264 has three main profiles: Baseline, Main, and High [16]. The baseline profile is used in some videoconferencing and mobile applications. The main profile is used for broadcast methods such as HDTV. The high profile is used by the Blu-ray Discs and the DVD HDTV broadcast service[16]. The next chapter explains this standard in more detail.

1.2.7 H.265

H.265, also known as High Efficiency Video Coding (HEVC) [18], is a successor to H.264. It was developed by IUT-T and MPEG and released in 2013. HEVC provides 25% - 50% better compression efficiency compared to H.264 for the same video quality. H.265 achieves this coding efficiency but requires increased computational complexity.

1.2.8 AV1

Amazon, Cisco, Google, Intel, Microsoft, Mozilla and Netflix released the first version of a royalty free high-quality web video codec, known as AOMedia Video 1 (AV1) [19], on April 2016. Some tests show AV1 has a better compression ratio compared to H.265 and H.264. The operating speed of the AV1 encoder is significantly lower than competitors [20].

1.2.9 VVC

Versatile Video Coding (VVC) [21] is a future video compression standard being developed by MPEG and ITU-T and is planned to be finalize in 2020. The goal of the new standard is having 30%-50% better compression ratio than H.265 and support higher video resolution from 4K to 16K. The Versatile Test Model (VTM) is used to evaluate the standard's performance [20].

(hh:mm:ss)	Configuration	
	AI	MAX
X264(Medium)	00:00:01	00:00:03
X264(Placebo)	00:00:03	00:03:39
X265(Medium)	00:00:20	00:00:06
X265(Placebo)	00:00:46	00:05:45
HM	00:01:09	00:05:34
JEM	01:04:32	01:16:36
VTM	00:51:00	01:36:30
AV1	00:16:21	00:38:21

Figure 2-2 Absolute per-picture encoding times for the 4K video sequence

[20] compares the performance of most recent standards based on absolute per-picture encoding time of the 4k Toddler Fountain video sequence. As shown in figure 1-2, depending on the configuration and test sequence, either the JEM or the AV1 encoders are the slowest. The x264 (medium) encoder is the fastest.

H.264 employs block-based motion compensation, dividing each video frame into macroblocks and using motion vectors to estimate block motion between frames. It supports variable block sizes for motion estimation and utilizes techniques like Context-Adaptive Variable Length Coding (CAVLC) for efficient entropy coding, ensuring effective video data compression. A Discrete Cosine Transform (DCT) is applied to transform pixel values into frequency domain coefficients. Then, quantization and entropy coding are applied on these coefficients to achieve compression. Further details about the H.264 architecture will be provided in the next chapter.

On the other hand, H.265 introduces more flexible block partitioning, enabling smaller block sizes. It can partition a macroblock into smaller coding units (CU), prediction units (PU), and transform units (TU), providing finer granularity for compression. This standard enhances motion compensation with more precise motion vector prediction and

allows for greater search ranges. H.265 adopts a modified Discrete Cosine Transform (MDCT) that is more effective in reducing spatial redundancy. Also, it supports 8x8 and 16x16 block sizes to enhance coding and incorporates larger context sizes for entropy coding. H.265 introduces extra prediction modes, including intra angular prediction and bi-directional prediction.

Both standards share similar architectures, but H.265 offers improvements over H.264 in block partitioning, motion compensation, transform coding, entropy coding, and prediction modes. However, these enhancements also demand increased computational resources for encoding, which should be considered when selecting between the two standards.

In terms of FPGA implementation, standards like H.264 (AVC) or H.265 can be challenging due to the complexity of the algorithms involved and the hardware resources required. However, H.264 is more reasonably implemented in hardware compared to H.265 for several reasons:

1. Complexity: H.264 is generally less complex than H.265. H.264's encoding process is more straightforward and requires fewer computational resources, making it more suitable for FPGA implementations with limited resources.

2. Resource Efficiency: H.264 typically requires fewer FPGA resources to achieve real-time or near-real-time performance compared to H.265. This makes it a more practical choice for FPGA implementations.

3. Standard Profiles: H.264 offers various profiles (e.g., Baseline, Main, High) with different levels of complexity. FPGA implementations can target specific profiles, which allows for resource optimization based on the application's requirements.

Therefore, among mentioned standards, we choose H.264 because it has a better compression ratio compared to previous standards and less computational complexity compared to subsequent standards, namely H.265.

CHAPTER TWO

AN OVERVIEW OF THE H.264 SPECIFICATION

This chapter introduces fundamental concepts of video coding and outlined H.264 encoder block structure. A detailed explanation of H.264 syntax will be provided at the final section of the chapter.

2.1 Basics of Video Coding

Video coding aims to reduce the data size of a digital video, making it more compact for storage or transmission while maintaining video quality. As illustrated in figure 2-1, this compression process comprises two primary components: a video encoder, responsible for compression, and a video decoder, responsible for decompression. Together, these two components form what is known as a CODEC [22].

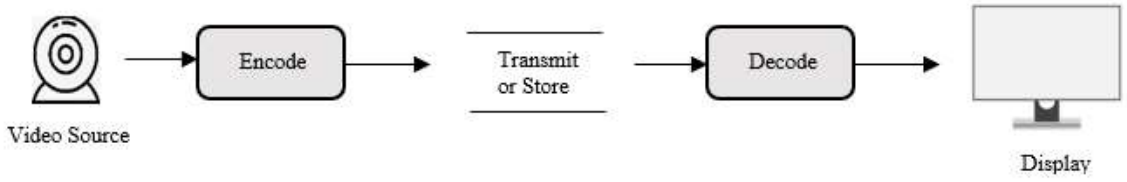


Figure 2-1 Video coding process

There are two primary categories of video compression techniques: lossless and lossy. Lossless compression aims to reduce the number of bits required to represent the input video without any loss of information. Consequently, the decoded video will precisely replicate the original input video [22]. Lossy compression, on the other hand, aims to minimize the number of bits while attempting to retain the maximum feasible amount of

information. The majority of video compression techniques are designed to be lossy, as these algorithms can achieve greater compression ratios [22]. However, this approach always involves a balance between the level of compression and the resulting video quality. Some video encoding approaches aim to maintain video quality while reducing the bit rate, though this comes at the expense of increased computational complexity.

2.1.1 Color Spaces

A pixel or picture element is the smallest unit of a digital video frame which is represented by numbers indicating its color value. For grayscale images, the color value is a number between 0 and 255. For color images, at least three numbers are required to indicate the color value. RGB and YCbCr are the most common color spaces [22].

RGB

In this color space, three numbers representing red, green, and blue are added together to make a broad range of colors.

YCbCr

Y is the luminance component, Cb is blue chrominance, and Cr is red chrominance. This color space is used widely in digital video coding because the chroma components (CB and CR) can be stored with lower resolution than the luminance Y [23]. This is because the human eyes are more sensitive to intensity (luminance) than color. By lowering the resolution of the chroma components, the amount of data required to transmit YCbCr images is reduced with little or no loss of observed visual quality.

YCbCr are given by a weighted average of red, green, and blue according to the following equations [22]

$$Y = 0.257R + 0.504G + 0.098B + 16 \quad \text{Eq. 2.1}$$
$$Cb = -0.148R - 0.291G + 0.439B + 128 \quad \text{Eq. 2.2}$$
$$Cr = 0.439R - 0.368G - 0.071B + 128 \quad \text{Eq. 2.3}$$

2.2 H.264 Encoder Model

H.264, MPEG-2, MPEG-4 and H.263 video encoders are based on the coding model shown in figure 2-2 [24]. The encoders perform prediction, transformations, quantization, and entropy encoding to produce a compressed bitstream from a video. The following sections explain each of these processes in more detail [25].

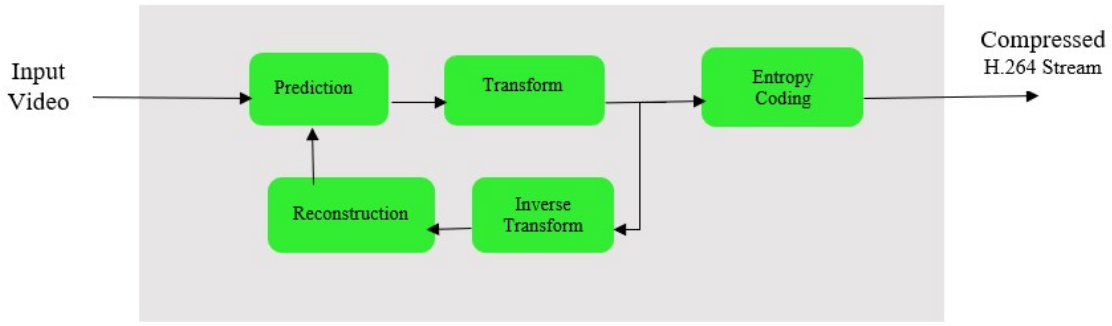


Figure 2-2 H.264 coding model [25]

2.2.1 Prediction

The prediction step exploits two types of redundancy in a video sequence, spatial redundancy, and temporal redundancy. Spatial redundancy is redundancy among neighboring pixels in the same video frame. The technique of reducing spatial redundancy is called intra prediction. Temporal redundancy is redundancy between pixels in the current frame and other frames (reference frames) that have already been coded.

The coding technique that reduces temporal redundancies is called inter prediction [25, 26].

The encoder processes a frame in macroblock units. Both predictions are performed for each macroblock. Based on block distortion measure, one prediction will be chosen for the macroblock. The predicted macroblock is subtracted from the current macroblock to form a residual block. This block is input to the next step which is a transform (Figure 2-3) [26].

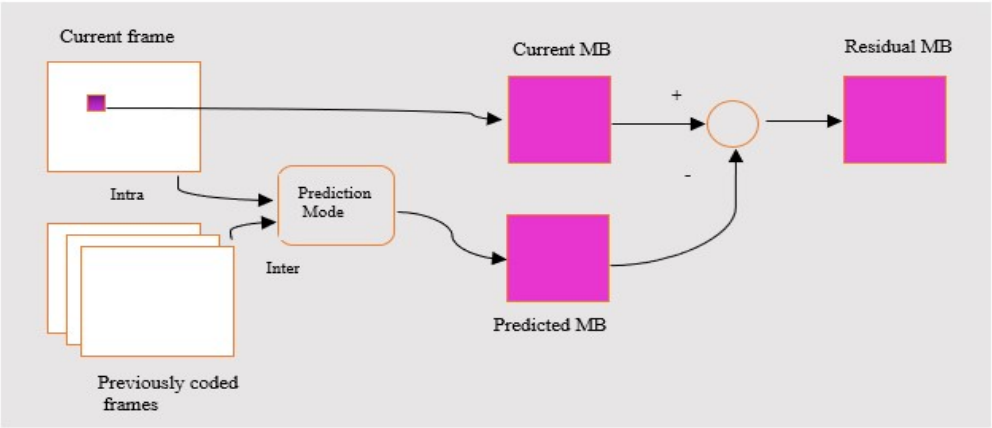


Figure 2-3 H.264 prediction model

2.2.1.1 Intra Prediction

In this prediction, each macroblock is encoded using already encoded pixels on the left and top of the block (Figure 2-4) [26].

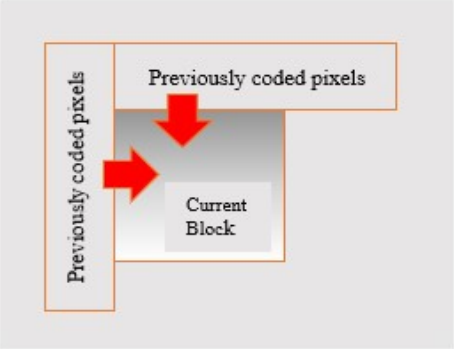


Figure 2-4 Intra prediction model

There are three choices of macroblock sizes, 16x16, 8x8, and 4x4 for luma components. There is one choice of block size, 8x8 for chroma components. There are 9 different intra prediction modes for each 4x4 and 8x8 luma macroblocks; 4 for a 16x16 luma macroblock and 4 for chroma macroblocks (MB), shown in figure 2-5. Each mode will be explained in more detail in following sections [26].

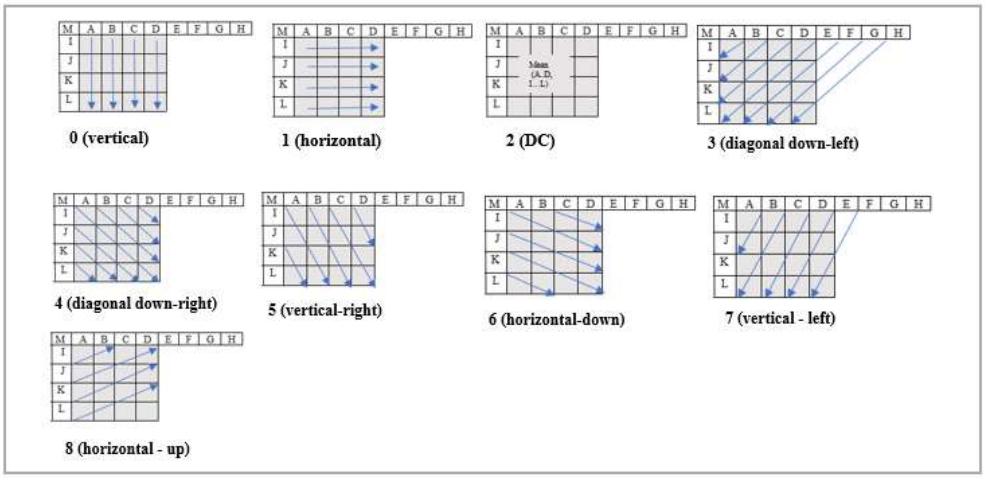


Figure 2-5 Intra prediction modes

Vertical Mode 0

All MB pixels will be replaced with the pixels on the top (T) of the MB. MB (j, i) = T (i) shown in

Figure 2-6 [26].

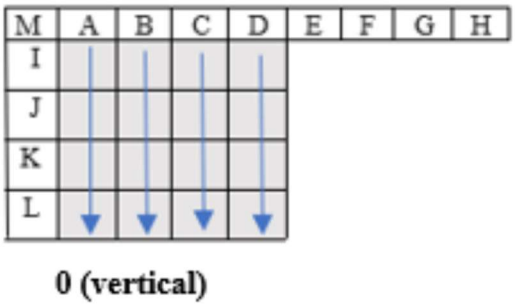


Figure 2-6 Vertical mode 0

Horizontal Mode 1

All MB pixels will be replaced with the pixels on the left(L) side of the MB. MB (j, i) = L (i) shown in figure 2-7 [26].

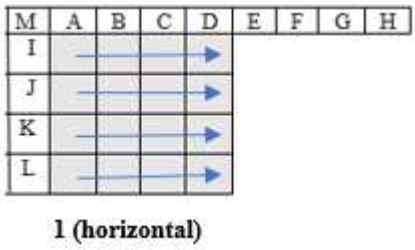


Figure 2-7 Horizontal mode1

DC Mode 2

All MB pixels will be replaced with the mean of the pixels on the left and top side of the MB[26]. MB (i, j) = round (mean ([L (1: N) T (1: N) 4])) shown in figure 2-8.

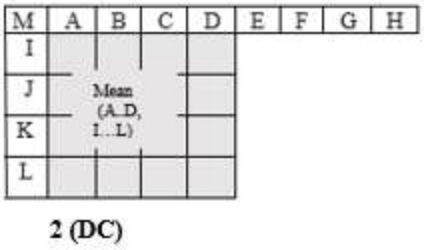


Figure 2-8 DC mode 2

Diagonal Down Left Mode 3

All MB pixels are replaced with the values calculated using equations in figure 2-9 [26].

$$a = (T(1) + 2*T(2) + T(3) + 2) / 4;\quad b = (T(2) + 2*T(3) + T(4) + 2) / 4;$$

$$c = (T(3) + 2*T(4) + T(5) + 2) / 4;\quad d = (T(4) + 2*T(5) + T(6) + 2) / 4;$$

$$e = (T(5) + 2*T(6) + T(7) + 2) / 4;\quad f = (T(6) + 2*T(7) + T(8) + 2) / 4;$$

$$g = (T(7) + 3*T(8) + 2) / 4;$$

MB(1,1)=a; MB(1,2)=b; MB(1,3)=c; MB(1,4)=d;

MB(2,1)=b; MB(2,2)=c; MB(2,3)=d; MB(2,4)=e;

MB(3,1)=c; MB(3,2)=d; MB(3,3)=e; MB(3,4)=f;

MB(4,1)=d; MB(4,2)=e; MB(4,3)=f; MB(4,4)=g;

Figure 2-9 Diagonal down left mode 3

Diagonal Down Right Mode 4

All MB pixels are replaced with the values calculated using equations in figure 2-10.

a = (L(4) + 2*L(3) + L(2) + 2) / 4; b = (L(3) + 2*L(2) + L(1) + 2) / 4;

c = (L(2) + 2*L(1) + M + 2) / 4; d = (L(1) + 2*M+ T(1) + 2) / 4;

e = (M + 2*T(1) + T(2) + 2) / 4; f = (T(1) + 2*T(2) + T(3) + 2) / 4;

g = (T(1) + 2*T(3) + T(4) + 2) / 4;

MB(1,1)=d; MB(1,2)=e; MB(1,3)=f; MB(1,4)=g;

MB(2,1)=c; MB(2,2)=d; MB(2,3)=e; MB(2,4)=f;

MB(3,1)=b; MB(3,2)=c; MB(3,3)=d; MB(3,4)=e;

MB(4,1)=a; MB(4,2)=b; MB(4,3)=c; MB(4,4)=d;

Figure 2-10 Diagonal down right mode 4

Vertical Right Mode 5

All MB pixels are replaced with the values calculated using equations in figure 2-11.

a = (M + T(1) + 1) / 2; b = (T(1) + T(2) + 1) / 2;

c = (T(2) + T(3) + 1) / 2; d = (T(3) + T(4) + 1) / 2;

e = (L(1) + 2*M + T(1) + 2) / 4; f = (M + 2*T(1) + T(2) + 2) / 4;

g = (T(1) + 2*T(2) + T(3) + 2) / 4; h = (T(2) + 2*T(3) + T(4) + 2) / 4;

i = (M + 2*L(1) + L(2) + 2) / 4; j = (L(1) + 2*L(2) + L(3) + 2) / 4;

MB(1,1)=a; MB(1,2)=b; MB(1,3)=c; MB(1,4)=d;

MB(2,1)=e; MB(2,2)=f; MB(2,3)=g; MB(2,4)=h;

MB(3,1)=i; MB(3,2)=a; MB(3,3)=b; MB(3,4)=c;

MB(4,1)=j; MB(4,2)=e; MB(4,3)=f; MB(4,4)=g;

Figure 2-11 Vertical right mode 5

Horizontal Down Mode 6

All MB pixels are replaced with the values calculated using equations in figure 2-12.

```
a = (M + L(1) + 1) / 2;              b = (L(1) + 2*M + T(1) + 2) / 4;
c = (M + 2*T(1) + T(2) + 2) / 4;   d = (T(1) + 2*T(2) + T(3) + 2) / 4;
e = (L(1) + L(2) + 1) / 2;           f = (M + 2*L(1) + L(2) + 2) / 4;
g = (L(2) + L(3) + 1) / 2;           h = (L(1) + 2*L(2) + L(3) + 2) / 4;
i = (L(3) + L(4) + 1) / 2;           j = (L(2) + 2*L(3) + L(4) + 2) / 4;
MB(1,1)=a; MB(1,2)=b; MB(1,3)=c; MB(1,4)=d;
MB(2,1)=e; MB(2,2)=f;  MB(2,3)=a; MB(2,4)=b;
MB(3,1)=g; MB(3,2)=h; MB(3,3)=e; MB(3,4)=f;
MB(4,1)=i;  MB(4,2)=j;  MB(4,3)=g;  MB(4,4)=h;
```

Figure 2-12 Horizontal down mode 6

Vertical Left Mode 7

All MB pixels are replaced with the values calculated using equations in:

```
a = (T(1) + T(2) + 1) / 2;      b = (T(2) + T(3) + 1) / 2;
c = (T(3) + T(4) + 1) / 2;      d = (T(4) + T(5) + 1) / 2;
e = (T(5) + T(6) + 1) / 2;      f = (T(1) + 2*T(2) + T(3) + 2) / 4;
g = (T(2) + 2*T(3) + T(4) + 2) / 4;   h = (T(3) + 2*T(4) + T(5) + 2) / 4;
i = (T(4) + 2*T(5) + T(6) + 2) / 4;   j = (T(5) + 2*T(6) + T(7) + 2) / 4;
MB(1,1)=a; MB(1,2)=b; MB(1,3)=c; MB(1,4)=d;
MB(2,1)=f; MB(2,2)=g;  MB(2,3)=h; MB(2,4)=i;
MB(3,1)=b; MB(3,2)=c; MB(3,3)=d; MB t(3,4)=e;
MB(4,1)=g; MB(4,2)=h; MB(4,3)=i;  MB(4,4)=j
```

Figure 2-13 Vertical left mode 7

Horizontal Up Mode 8

All MB pixels are replaced with the values calculated using equations in figure 2-14.

```
a = (L(1) + L(2) + 1) / 2;   b = (L(1) + 2*L(2) + L(3) + 2) / 4;
c = (L(2) + L(3) + 1) / 2;  d = (L(2) + 2*L(3) + L(4) + 2) / 4;
e = (L(3) + L(4) + 1) / 2;  f = (L(3) + 3*L(4)      + 2) / 4;
g = L(4);

MB(1,1)=a; MB(1,2)=b; MB(1,3)=c; MB(1,4)=d;
MB(2,1)=c;  MB(2,2)=d; MB(2,3)=e; MB(2,4)=f;
MB(3,1)=e;  MB(3,2)=f;  MB(3,3)=g; MB(3,4)=g;
MB(4,1)=g;  MB(4,2)=g; MB(4,3)=g; MB(4,4)=g;
```

Figure 2-14 Horizontal up mode 8

Prediction Mode Criteria

Prediction Mode criteria provides a way to determine the best prediction mode of each block, independently. The most popular criterion used as a block distortion measure (BDM) are [27, 28]:

Mean square error (MSE):

$$MSE(i,j) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (C_{ij} - R_{ij})^2 \qquad \text{Eq. 2.4}$$

Mean absolute difference (MAD):

$$MAD(i,j) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |C_{ij} - R_{ij}| \qquad \text{Eq. 2.5}$$

Peak to Signal Noise Ratio (PSNR):

$$PSNR = 10log_{10} \frac{(peak\ to\ peak\ value\ of\ data)^2}{MSE} \qquad \text{Eq. 2.6}$$

Sum of absolute difference (SAD)[29]:

$$SAD(i,j) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |C_{ij} - R_{ij}| \qquad \text{Eq. 2.7}$$

where $N \times N$ is the size of macroblock. C*ij* and R*ij* are the pixel values of pixel and predicted macroblock calculated by the prediction mode at position at *i,j,* respectively. SAD is commonly used in hardware implementations because of its simplicity [29], it does not require any multiplication or division, and PSNR is commonly used for finding video quality.

2.2.1.2 Inter Prediction

In inter prediction [22, 30], each MB is encoded using MBs in previously or future encoded frames. Several block sizes are allowed from 16x16 to 4x4 (figure 2-15). A macroblock can be partitioned into two 8x16 blocks, two 16x8, or four 8x8 blocks. Further, an 8x8 macroblock partition can be divided into two 4x8 blocks, two 8x4 blocks, or four 4x4 blocks which are called as sub-macroblock partitions.

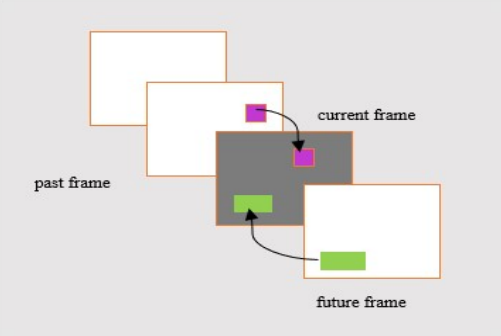


Figure 2-15 Inter prediction model

The process of finding matching pixel blocks is called Motion Estimation. Given a current frame, *k*, three different cases can be used as reference frame [22, 30]:

1- Previous frame (k-1)

2- Future frame(k+1)

3- Both previous and future frames

The first case is called backward motion estimation and the frame is referred to as the "P" frame (picture). The second case is called forward motion estimation. Finally, the last case is called bi-directional prediction and the frame is referred to as the "B" frame (picture) [22, 30].

The first frame in a video sequence is intra-coded to ensure that all MBs of the frame are intra coded. Successive frames are inter-coded (P-picture, B-picture); some MBs are intra-coded, and some are inter-coded (figure 2-16).

Figure 2-16 Input video sequence frame

The goal of motion estimation is to obtain motion vectors representing the estimated motion. There are different algorithms to compute the motion estimation vector which is out of scope of this thesis since these algorithms are computationally intense and it's not possible to compute them in real-time application.

2.2.2 Transform and Quantization

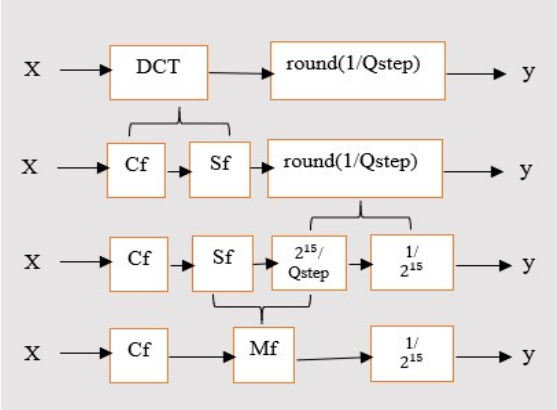


Figure 2-17 Development of the forward transform and quantization process

In an H.264 encoder, macroblock data are transformed and quantized prior to entropy coding. The transform is a scaled approximation to a 4x4 Discrete Cosine Transform (DCT) that can be computed using simple integer arithmetic. The use of integer coefficients eliminates rounding errors that cause drifting artifacts common with DCT-based video codecs. The process reorganizes into a core part, *Cf*, and a scaling part, *Sf*, as shown in figure 2-17 [30, 31].

Consider a 4*4 two-dimensional DCT of block X [32, 33]

$$Y = A.X.A^T \qquad \text{Eq. 2.8}$$

Where Y is the matrix multiplication and the matrix X is the 4x4 residual block and A is specified as Eq.2.9 [32, 33].

$$A_1 = C_f \cdot R_f \qquad \text{Eq. 2.9}$$

$$C_f = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \qquad \text{Eq. 2.10}$$

$$R_f = \begin{bmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ 1/\sqrt{10} & 1/\sqrt{10} & 1/\sqrt{10} & 1/\sqrt{10} \\ 1/2 & 1/2 & 1/2 & 1/2 \\ 1/\sqrt{10} & 1/\sqrt{10} & 1/\sqrt{10} & 1/\sqrt{10} \end{bmatrix} \quad \text{Eq. 2.11}$$

$$Y = \left[C_f \cdot R_f\right] \cdot X \cdot [C_f^T \cdot R_f^T] \quad \text{Eq. 2.12}$$

Rearrange the equation [32, 33]

$$Y = \left[C_f \cdot C_f^T\right] \cdot X \cdot [R_f \cdot R_f^T]$$

where

$$S_f = R_f \cdot R_f^T \quad \text{Eq. 2.13}$$

$$Y = \left[C_f \cdot X \cdot C_f^T\right] \cdot S_f$$

$$S_f = R_f \cdot R_f^T = \begin{bmatrix} 1/4 & 1/2\sqrt{10} & 1/4 & 1/2\sqrt{10} \\ 1/2\sqrt{10} & 1/10 & 1/2\sqrt{10} & 1/10 \\ 1/4 & 1/2\sqrt{10} & 1/4 & 1/2\sqrt{10} \\ 1/2\sqrt{10} & 1/10 & 1/2\sqrt{10} & 1/10 \end{bmatrix} \quad \text{Eq. 2.14}$$

Scale the quantization process by a constant (2^{15}) and compensate by dividing and rounding the result.

Combine S_f and the quantization process into MF [2, 3]

$$\text{MF} = \frac{S_f . 2^{15}}{Qstep}. \quad \text{Eq. 2.15}$$

$$Y = round(C_f \cdot X \cdot C_f^T . 2^{\frac{MF}{qbits}}) \qquad qbits = 15 + floor(\frac{QP}{6}) \quad \text{Eq. 2.16}$$

QP is the quantization step size which is calculated based on Table 2.1. It uses a set of 52 uniform scalar quantizers with a step increment of approximately 12.5% between each [2, 3].

Table 2-1 Quantization step sizes in an H.264 codec

QP	0	1	2	3	4	5	6	7	8	9	10	11	12	….
QStep	0.625	0.6875	0.8125	0.875	1	1.125	1.25	1.375	1.625	1.75	2	2.25	2.5	…..
QP	….	18	…	24	….	30	…..	36	…..	42	…..	48	…..	51
QStep		5		10		20		40		80		160		224

The *MF* value depends on *QP* and the position (*i, j*) of the element in the macroblock as shown in Table 2.2. The factor, *MF*, remains unchanged for *QP>5* which can be calculated by [2, 3]:

$$MF_{QP>5} = MF_{QP=Q \ \%6} \qquad \text{Eq. 2.17}$$

Then $Y = round(C_f \cdot X \cdot \ C_f^T . 2^{\frac{MF}{qbits}})$ can be further simplified in integer arithmetic as follow [2, 3]:

$$|Z_{ij}| = (|Y_{ij}| \cdot MF + f) \gg qbits \qquad \text{Eq. 2.18}$$

where *f* is a parameter used to avoid rounding errors and it depends on prediction type of the block and *QP [2, 3]*.

Table 2-2 The multiplication factor in H.264 [2, 3]

QP	Positions (0,0),(2,0),(0,2),(2,2)	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

only additions and shifts are needed to implement the integer transform and inverse integer transform as shown in figure 2-18.

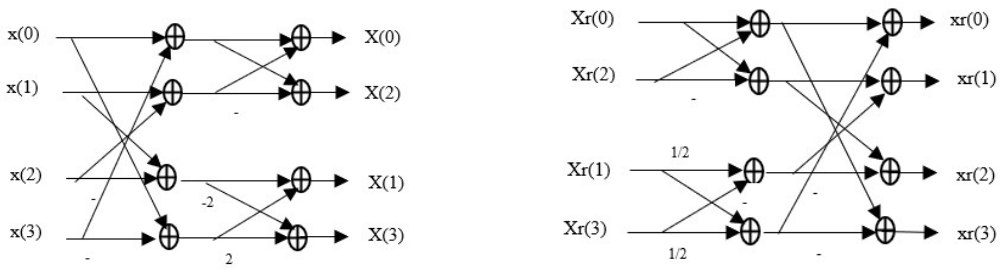


Figure 2-18 Implementation of the direct transform (left) and inverse transform (right) [2]

2.2.3 Inverse Transform and Quantization

Figure 2-19 illustrates the inverse transform and quantization process. The inverse discrete cosine (IDCT) process is reorganized into a core transform, C_i, and a scaling matrix, S_i. Multiply the quantization step size, Qstep, by constant 2^6 and compensate by dividing and rounding the result. The steps to obtain C_i and S_i are as Eq.2.19 – Eq.2.26 [26].

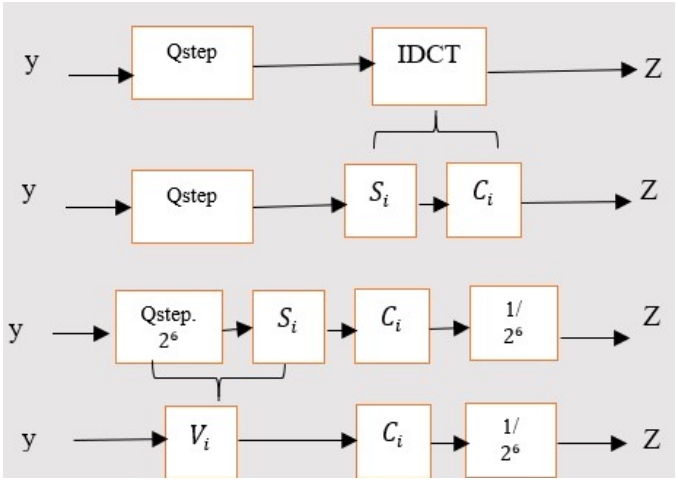


Figure 2-19 Development of the inverse transform and quantization process

4x4 two –dimensional IDCT of block y [26] :

$$Z = A.y.A^T \qquad \text{Eq. 2.19}$$

$$A = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix} \quad \text{Eq. 2.20}$$

Where $a = {}^{1}/_{2}$

$$b = \sqrt{\frac{1}{2}}\, cos\frac{\pi}{8} = 0.653$$

$$c = \sqrt{\frac{1}{2}}\, cos\frac{3\pi}{8} = 0.2706$$

Scaling each row of A and rounding to the nearest 0.5, which gives C_i [26]

$$C_i = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1/2 & -1/2 & -1 \\ 1 & -1 & -1 & 1 \\ 1/2 & -1 & 1 & -1/2 \end{bmatrix} \quad \text{Eq. 2.21}$$

Let $A_2 = C_i . R_i$

$$R_i = \begin{bmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ \sqrt{2}/5 & \sqrt{2}/5 & \sqrt{2}/5 & \sqrt{2}/5 \\ 1/2 & 1/2 & 1/2 & 1/2 \\ \sqrt{2}/5 & \sqrt{2}/5 & \sqrt{2}/5 & \sqrt{2}/5 \end{bmatrix} \quad \text{Eq. 2.22}$$

The two-dimensional inverse transform becomes:

$$Z = A_2 . y . A_2^T = [C_i^T \cdot R_i^T] \cdot y \cdot [C_i \cdot R_i] \quad \text{Eq. 2.23}$$

Rearranging the equation:

$$Z = [C_i^T] \cdot [y \cdot R_i^T \cdot R_i] \cdot [C_i] \quad \text{Eq. 2.24}$$

$$Z = [C_i^T] \cdot [y \cdot S_i] \cdot [C_i]$$

$$S_i = R_i^T \cdot R_i = \begin{bmatrix} 1/4 & 1/\sqrt{10} & 1/4 & 1/\sqrt{10} \\ 1/\sqrt{10} & 2/5 & 1/\sqrt{10} & 2/5 \\ 1/4 & 1/\sqrt{10} & 1/4 & 1/\sqrt{10} \\ 1/\sqrt{10} & 2/5 & 1/\sqrt{10} & 2/5 \end{bmatrix} \quad \text{Eq. 2.25}$$

Where $V_i = S_i \cdot Qstep . 2^6$

V_i values are given by H.264 standard show in Table 2.3.

Table 2-3 Values of V used in the H.264 standard [10]

P	V(r,0) V_i Positions (0,0),(0,2),(2,0),(2,2)	V(r,1) V_i Positions (1,1),(1,3),(3,1),(3,3)	V(r,2) Remaining V_i 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

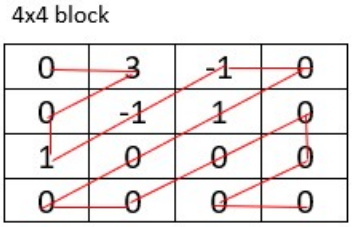
The complete inverse transformation and scaling process is given by [10]:

$$Z = round([C_i^T] \cdot \left[y \cdot V(QP\%6, n) \cdot 2^{floor\left(\frac{qp}{6}\right)}\right] \cdot [C_i] \cdot {}^{1}/_{2^6}) \quad \text{Eq. 2.26}$$

2.2.4 Entropy Coding

In the CAVLC unit, a 16x16 MB is divided into 16 4x4 blocks, and the 4x4 blocks are processed one after another in a defined order. Each 4x4 block is processed through two phases, the scanning phase and the coding phase. In the scanning phase, the residues are read in a backward zig-zag order shown in table 2.4 [34, 35]. Then, the run-level symbols are extracted. In the coding phase, the symbols are translated into codewords by the corresponding class of tables. There are four types of tables available: Coeff_Token, Total_Zeros, Run_Before, and Level Code [34, 35]. The parameter *TotalCoeffs* is the sum of all non-zero coefficients in a forward scan. The parameter *TotalZeros* is the sum of all zeros located before the last non-zero coefficient in a forward scan[35]. Table 2.5 illustrates the CAVLC procedure in more detail.

Table 2-4 Residual block scanned in backward zig-zag order [1, 35]



The reordered block is 0,3,0,1, -1,-1,0,1,0… Total Coeffs = 5 (indexed from highest frequency [4] to lowest frequency [0]) and the TotalZeros = 3. T1s = 3 (in fact, there are 4 trailing ones but only 3 can be encoded as a "special case") [1]

Table 2-5 CAVLC procedure [1]

Encoding:

Element	Value	Code
Coeff_token	TotalCoeffs=5,T1s=3	0000100
T1 sign (4)	+	0
T1 sign (3)	-	1
T1 sign (2)	-	1
Level (1)	+1 (use Level_VLC0)	1
Level (0)	+3 (use Level_VLC1)	0010
TotalZeros	3	111
Run_before(4)	ZaroLeft = 3; run_before = 1	10
Run_before(3)	ZaroLeft = 2; run_before = 0	1
Run_before(2)	ZaroLeft = 2; run_before = 0	1
Run_before(1)	ZaroLeft = 2; run_before = 1	01
Run_before(0)	ZaroLeft = 1; run_before = 1	No code required; last coefficient

2.3 H.264 Syntax

In the H.264 encoder, the coded video data are placed into a logical data packet called a Network Abstraction Layer (NAL) unit [36]. This structure/syntax is defined in the

H.264 standard. NAL units consist of VCL and non-VCL units. The VCL units contain the coded video samples. The non-VCL units contain header information such as frame rate, frame size, frame type, and profile. Figure 2-20 shows the H.264 output stream format [36].

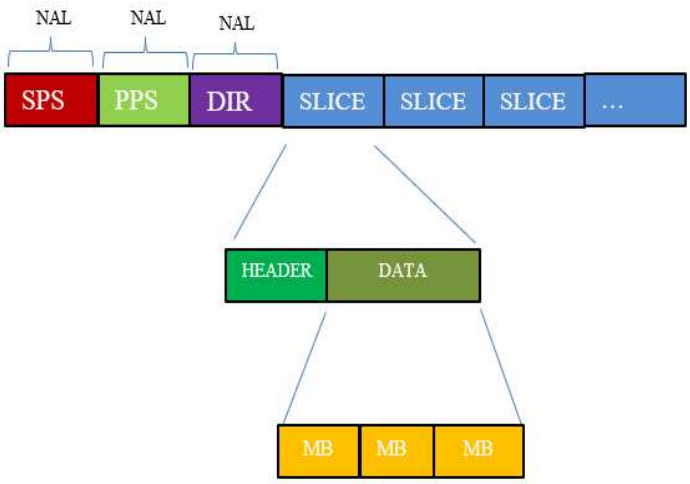


Figure 2-20 H.264 stream

The NAL units are explained in more detail as follow:

The **Sequence Parameter Set (SPS) is** a non-VCL unit that contains information required to configure the decoder such as profile, level, resolution, frame rate shown in table 2-6[36].

Table 2-6 Sequence parameter set syntax

Seq_parameter_set_rbsp(){	C	Descriptor
Profile_idc	0	u(8)
Constraint_set0_flag	0	u(1)
Constraint_set1_flag	0	u(1)
Constraint_set2_flag	0	u(1)
Reserved_zero_5bits/*equal to 0 */	0	u(5)
Level_idc	0	u(8)
Seq_parameter_set_id	0	ue(v)
Log2_max_frame_num_minus4	0	ue(v)
Pic_order_cnt_type	0	ue(v)
If(pic_order_cnt_type == 0)		
Log2_max_picture_order_cnt_lsb_minus4	0	ue(v)
Else if(pic_order_cnt_type == 1){		
Delta_pic_order_always_zero_flag	0	u(1)
Offset_for_non_ref_pic	0	se(v)
Offset_for_top_to_bottom_flied	0	se(v)
Num_ref_frames_in_pic_order_cnt_cycle	0	ue(v)
For(i=0;i<num_ref_frames_in_pic_order_cnt_cycle;i++)		
Offset_for_ref_frame[i]	0	se(v)
}		
Num_ref_frames	0	ue(v)
Gaps_in_frame_num_value_allowed_flag	0	u(1)
Pic_width_in_mabs_minus1	0	ue(v)
Pic_heights_in_map_units_minus1	0	ue(v)
Frame_mbs_only_flag	0	u(1)
If(!frame_mbs_only_flag)		
Mb_adaptive_frame_field_flag	0	u(1)
Direct_8x8_inference_flag	0	u(1)
Frame_cropping_flag	0	u(1)
If(frame_cropping_flag){		
Frame_crop_left_offset	0	ue(v)
Frame_crop_right_offset	0	ue(v)
Frame_crop_top_offset	0	ue(v)
Frame_crop_bottom_offset	0	ue(v)
}		
Vui_parameters_present_flag	0	u(1)
If(Vui_parameters_present_flag)		
vui_parameters()	0	
Rbsp_trailing_bits()	0	
}		

2.3.1 Parameter Set Syntax

The Picture Parameter Set (PPS) is a non-VCL unit that contains information on entropy coding techniques (CAVLC or CABAC), slice groups, motion prediction and deblocking filters shown in table 2.7 [36].

Table 2-7 Picture parameter set syntax

Pic_parameter_set_data(){	**Descriptor**	**No. of bits (Min/Max values)**
Pic_parameter_set_id	ue(v)	1,3 (0/2)
Seq_parameter_set_id	ue(v)	1(0/0)
Entropy_coding_mode_flag	u(1)	1
Bottom_field_pic_order_in_frame_present_flag	u(1)	1
Num_slice_groups_minus1	ue(v)	1(0/0)
Num_ref_idx_10_default_active_minus1	ue(v)	1,3,5,7,9,11(0/31)
Num_ref_idx_11_default_active_minus1	ue(v)	1,3,5,7,9,11(0/31)
Weighted_red_flag	u(1)	1
Weighted_bired_idc	u(2)	2
Pic_init_qp_minus26	se(v)	1(0/0)
Pic_init_qs_minus26	se(v)	1(0/0)
Chroma_qp_index_offset	se(v)	1,3,5,7,9 (-12/12)
Deblocking_filter_control_present_flag	u(1)	1
Constrained_intra_pred_flag	u(1)	1
Redundant_pic_cnt_present_flag	u(1)	1
Vui_parameters_present_flag	u(1)	1
}		
Trailing_bits()		1,2,3,4,5,6,7,8

The **Instantaneous Decoder Refresh (IDR)** is a VCL NALU and is a self-contained image slice. That is, an IDR can be decoded and displayed without referencing any other frames/slices. **SLICE** consists of header and data. The Slice header contains the information about the type of slice, the type of macroblocks in the slice, and the number of the slice frame. Slice data is a series of coded macroblocks. Slice header syntax shown in table 2.8 [36].

Table 2-8 Slice header syntax

Slice_header() {	C	Descriptor
First_mb_in_slice	2	ue(v)
Slice_type	2	ue(v)
Pic_parameter_set_id	2	ue(v)
Frame_num	2	u(v)
If(!frame_mbs_only_flag){		
Field_pic_flag	2	u(1)
If(field_pic_flag)		
Bottom_field_flag	2	u(1)
}		
If(nal_unit_type == 5)		
idr_pic_id	2	ue(v)
If(pic_order_cnt_type == 0){		
pic_order_cnt_lsb	2	ue(v)
If(pic_order_present_flag && !field_pic_flag)		
Delta_pic_order_cnt_bottom	2	se(v)
}		
If(pic_order_cnt_type == 1 && !delta_pic_order_always_zero_flag) {		
Delta_pic_order_cnt[0]	2	se(v)
If(pic_order_present_flag&& !field_pic_flag)		
Delta_pic_order_cnt[1]	2	se(v)
}		

A Macroblock consists of header and data as shown in table 2.9. The header contains the prediction type (intra/inter), Coded Block Pattern (COB), Quantization Parameter. Data is a set of luminance and chrominance components. The macroblock header syntax shown in figure 2-22 [36].

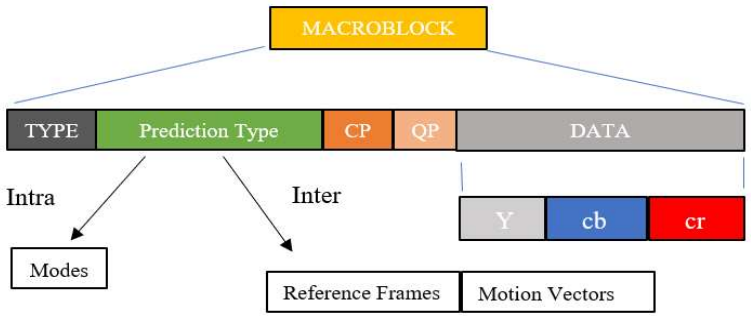


Figure 2-21 Macroblock

Table 2-9 Macroblock layer syntax

Macroblock_layer() {	C	Descriptor
mb_type	2	ue(v)|ae(v)
If(mb_type == I_PCM) {		
While(!byte_aligned())		
Pcm_alignment_zero_bit	2	f(1)
for(i=0;i<256*ChromaFormatFactor;i++)		
Pcm_byte[i]	2	u(8)
} else{		
if(MbPartPredMode(mb_type,0) != Intra_4x4 && MbPartPredMode(mb_type,0) != Intra_16x16 && NumMbPart(mb_type) == 4)		
Sub_mb_pred(mb_type)	2	
else		
mb_pred(mb_type)	2	
If(MbPartPredMode(mb_type,0) != Intra_16x16)		
Coded_block_pattern	2	me(v)|ae(v)
If(CodedBlockPatternLuma > 0 || CodedBlockPatternChroma> 0 || MbPartPredMode(mb_type,0) == Intra_16x16){		
mb_qp_delta	2	se(v)|ae(v)
Residual()	3|4	
}		
}		
}		

The next chapter presents a proposed H.264 Encoder model and the criteria that are considered to design the encoder.

CHAPTER THREE

PROPOSED H.264 ENCODER MODEL AND DESIGN CRITERIA

This chapter presents the proposed H.264 encoder model and criteria which are considered before starting of the encoder design. These criteria are used to compare the performance of our encoder's design with other existing encoders.

3.1 Encoder Design Criteria

There are some design criteria which should be specified before designing an encoder. Since it depends on encoder application, the design model can vary. Some of these parameters are:

1- The Maximum Clock Frequency.

2- Hardware utilization

3- Target video resolution and frame rate

4- Real-time operation

5- Using External memory in hardware design

6- Compression ratio

There are several enterprise-level H.264 encoders available in the market, but they often come with limitations. Many of these encoders are hardware-specific, demanding significant hardware resources, and can be quite expensive. Here are a few examples:

1. VISENGI Inc: They offer H.264 Encoder IP for both H264E-I (intra-prediction only) and H264E-P (both intra and inter prediction). While they support various FPGA

devices, the resource requirements are relatively high, with approximately 54K LUTs and 32 DSPs for Xilinx devices and 35K ALMs and 32 DSPs for Altera devices.

2. AMD LogiCORE™ IP: AMD's H.264/H.265 Video Codec Unit (VCU) core is embedded as hard IP inside Zynq UltraScale+ MPSoC EV devices. This makes it device-specific and comes at a significant cost. Also, it consumes a lot of hardware resources, utilizing 504K logic cells and 1728 DSP slices.
3. SOC System-On-Chip Technologies: They provide an H.264 encoder IP core compatible with multiple FPGA families, including ALTERA Arria 10, ZYNQ 7045, and XILINX KINTEX Ultrascale. However, this IP also demands a substantial amount of hardware resources, requiring about 110K LUTs and 235 DSPs.

While these options are available, they often have constraints related to hardware specificity, resource utilization, and cost. To overcome these limitations, we focus on the design and implementation of a parametrized H.264 encoder with several key objectives in mind. First, we aim to minimize hardware utilization, so multiple encoder instances can be deployed on the same FPGA device to simultaneously encode multiple input channels. This is necessary for scenarios where we have multiple cameras connected to our device. Secondly, the encoder is not tied to specific hardware, and it is designed for high-frequency operation, ensuring low latency video encoding. Moreover, we aim to have the capability to handle high-resolution input video streams of up to 2Gbit/sec. While many encoders prioritize achieving high compression ratios, this encoder focuses on hardware simplicity to minimize resource consumption and some compression up to 2.7. To achieve real-time operation, it relies on intra-frame prediction, as inter-frame

prediction demands substantial computational resources that makes real-time processing challenging. The encoder is designed with a pipelined structure and parallel hardware components to achieve higher execution frequencies. Lastly, we avoid the use of off-chip memory due to reading from and writing to external memory can introduce latency.

Table 3.1 provides a comparison of the performance of various H.264 hardware designs based on the criteria mentioned above.

Table 3-1 Performance comparison

Design Criteria	[4]	[5]	[6]	[7]	[8]	[9]	This work
Profile	High	High, SVC	High	Baseline	Baseline	Baseline	Baseline
Frequency (MHZ)	130	120(high), 166 (SVC)	162	81(SD); 180 (HD)	N/A	28 (CIF);108 (HD720)	100
Hardware utilization	37178 ALUTs & 128 DSP	2079 KGates	3745 KGates	922.8 KGates	452.8 KGates	470 KGates	As low as possible
Target video resolution	0.5Gbit/s	0.5Gbit/s	0.5Gbit/s	82Mbit/s-0.5Gbit/s	6Mbit/s,82Mbit/s	24Mbit/s to 221Mbit/s	2Gbit/s
Target	Real-time	Scalable Extension SVC; High profile	Low power; Video size scalable	Hardware design for H.264 codec	Low power; Power aware; Portable devices	Dynamic QualityScalable; Poweraware video applications	Real-time
External memory usage	yes	yes	yes	yes	N/A	N/A	NO
Compression ratio	N/A	N/A	N/A	N/A	N/A	N/A	TBD
Extra notes	No entropy coding	UMC90nm CMOS 1P9M	CMOS 65 Technology	UMC 180, 1P6M CMOS	TSMC 180, 1P6M CMOS	CMOS 130	

Reference [37] (column [4] in table 3-1) executed inter-prediction for the high-profile encoder which is a time-consuming algorithm; the average time to determine the best

motion vector for several MB is 250 clock cycles. They didn't integrate the entropy coding block. Higher compression ratios can be achieved using the entropy coding block. [38-42] (columns [5,6,7,8,9] in table 3-1) implemented H.264 encoders based on 4- stage pipeline architecture, as shown in figure 3-1, with some modifications.

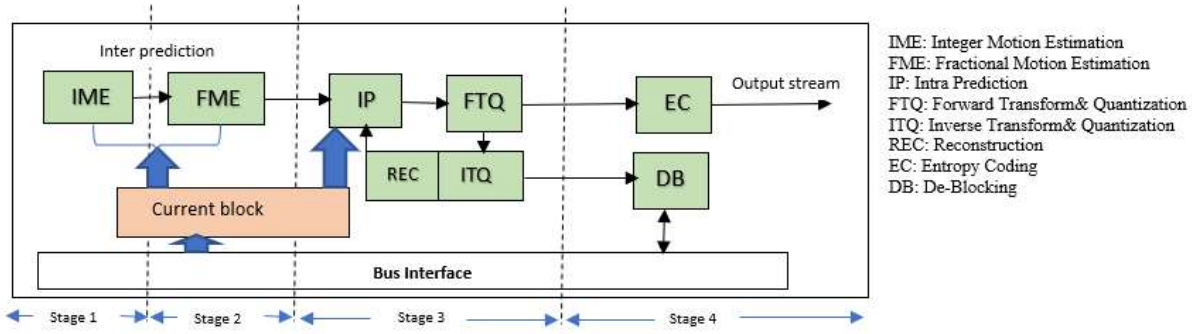


Figure 3-1 4- stage pipeline architecture of H.264 encoder

An H.264 encoder for high profile was proposed in [38] (column [5] in table 3-1). Figure 3-2 shows its 2-stage frame pipelining architecture and its B-frame parallel scheme. Two consecutive B-frames can be concurrently encoded and share the overlap search window data. With this method, the external memory bandwidth of loading search window for GOP IBBP can be reduced by 50%, and 25% for the HB. In the next stages, Intra, REC, FGS scan and EC blocks are duplicated to process two MBs in the same pipeline stage while de-blocking (DB) engine is shared. Many techniques were used to reduce external memory bandwidth and internal memory access. However, the area cost of the design is quite larger than classical architecture in figure 3-1.

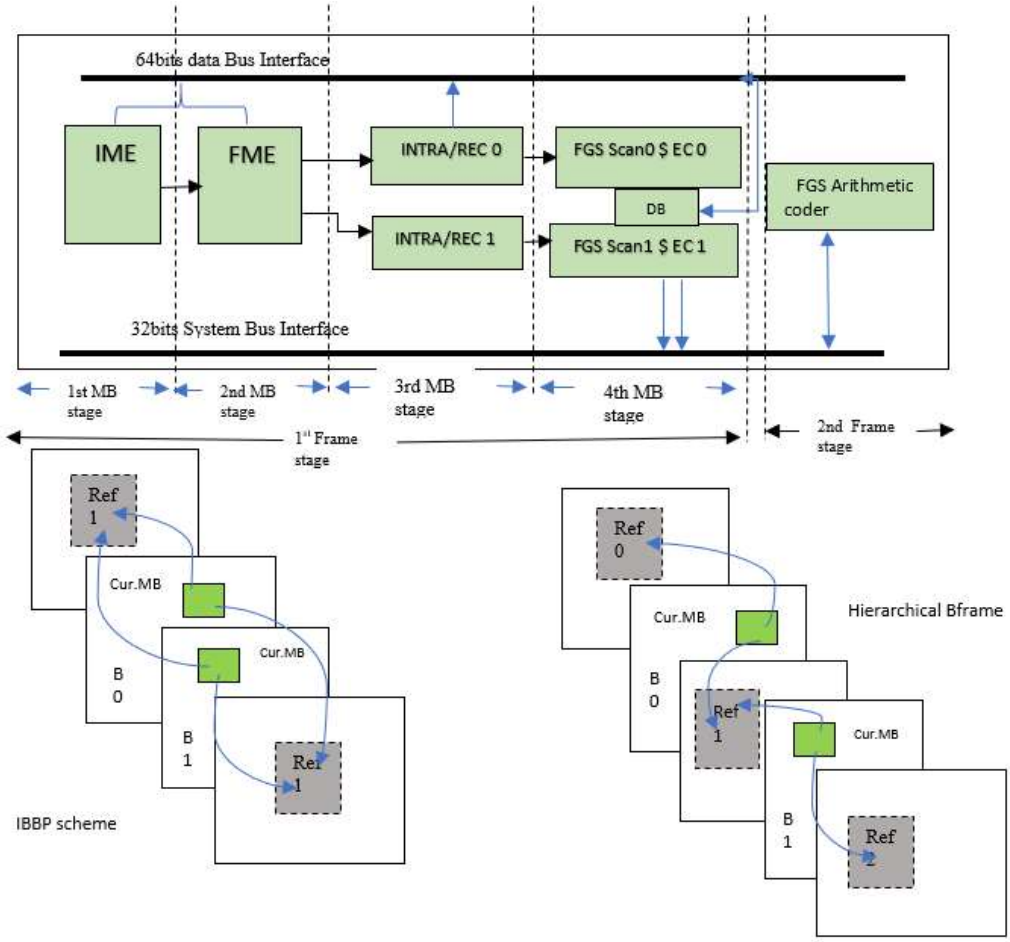


Figure 3-2 Block diagram of two- stage frame pipeline H.264 encoder[38] ([5])

Reference [39] (column [6] in table) introduced a H.264 encoder in high profile. It is a 2-stage pipeline encoder at frame level which includes two parallel MB-pipeline processing modules. This method increases the processing performance, but it also increases the area cost and therefore the power consumption.

In this work [40] (column [7] in table 3-1), the authors proposed a H.264 encoder for baseline profile. The encoder uses the 4-stage encoder architecture shown in figure 3-3. The reference frames are stored in an external memory. Many on-chip memories are used

to reduce the external bandwidth requirement, thus reducing the timing costs. The area cost of the design is much smaller than [39] (column [6] in table 3-1) but still substantial. In their study, as referenced in [41] (column [8] in table 3-1), they presented a 3-stage architecture shown in Figure 3-5. The design uses some data reuse techniques to reduce memory access requirement thus implementing an efficient high-throughput low-power scheme. However, they need to design a complex controlling system including the system control, the power control, and the PE control. The area cost of the design is smaller than [40] (column [7] in table 3-1) but it supports lower video resolutions.

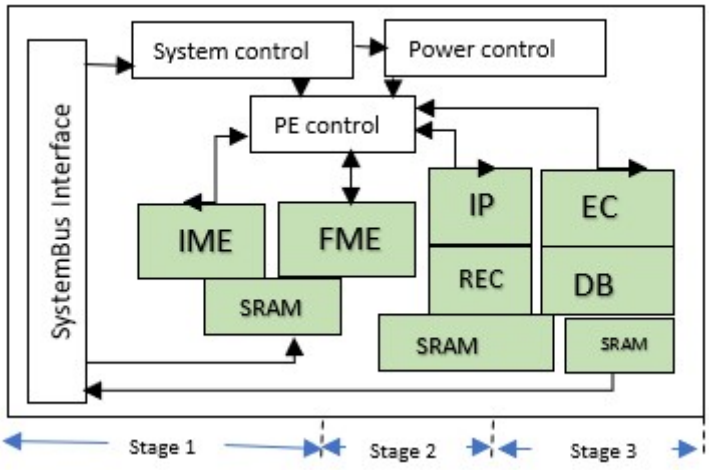


Figure 3-3 Proposed three- stage system architecture for the H.264 encoder [8]

The encoder in [42] (column [9] in table 3-1) utilizes less hardware with power-aware ability. The design follows 4-stage pipeline architecture shown in Figure 3.3. Some design techniques are used to simplify the encoding flow and reduce the data dependency. However, the encoder just supports low video resolution.

The architecture of the proposed H.264 encoder uses 4-stage pipeline design in figure 3-1 with some modification, as shown in figure 3-4. It is a 2-stage scheme. I eliminated

IME and FME since they are the most time-consuming blocks in the design with a high area cost. The local memory only will be used to store the necessary pixels, thus reduce the timing cost. Some saving techniques will be used to share resources among processing blocks therefore lower the area cost.

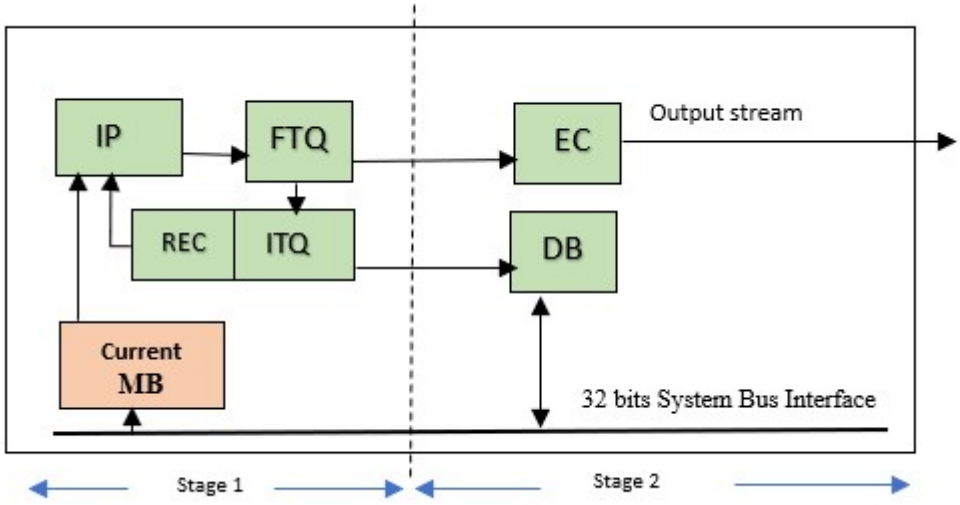


Figure 3-4 Proposed two- stage architecture for the H.264 encoder

The proposed encoder, depicted in Figure 3-4, is designed to be sufficiently small to allow multiple instances to run on a single FPGA. Consequently, it can encode multiple video channels concurrently and transmit the encoded video over Ethernet to a PC or store it locally. The following chapters provides a detailed explanation of the hardware architecture for each component within the proposed H.264 design.

CHAPTER FOUR

4x4 INTRA PREDICTION HARDWARE ARCHITECTURE AND FPGA IMPLEMENTATION

As mentioned before in chapter 2, intra-predicted macroblock just refers to the neighboring pixels of the macroblock in the same frame. There are two types of intra-prediction methods for luma components in a frame. One is 16x16 which predicts the entire macro block at once and another one is 4x4 which partitions a macroblock into sixteen 4x4 block to perform the predictions for each partition. This thesis implements only the 4x4 intra prediction type in hardware due to 4x4 intra prediction allows for more precise prediction of smaller blocks within a frame. This can lead to better compression efficiency when there are detailed or rapidly changing regions within a frame. The chroma components of a frame utilize an 8x8 intra prediction type.

This chapter is outlined as follows. section 4.1 explains the FPGA implementation of intra 4x4 luma and 8x8 chroma prediction. Section 4.2 presents FPGA implementation of the intra mode selection algorithm. Finally, section 4.3 provides the hardware usage and timing of the intra prediction implementation.

4.1 Intra 4x4 Luma and 8x8 Chroma Prediction

The intra 4x4 luma prediction scans the 16x16 luma MB and 8x8 chroma MB in the order shown in figure 4-1 [43], each location is a 4x4 block of pixels.

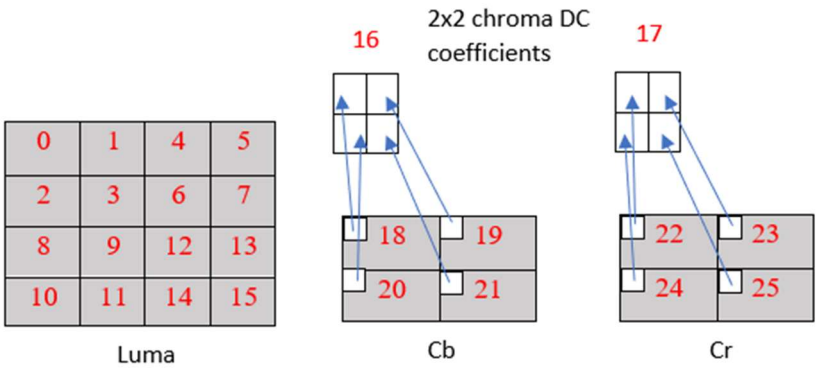


Figure 4-1 Scanning order of 4x4 blocks within MB

There are nine 4x4 luma prediction modes and four 8x8 chroma prediction modes. As shown in figure 4.1 [10] when the current 4x4 MB is number 12, its surrounding 4x4 MBs are 3,6,7,9. For vertical prediction the predicted block is constructed by above block 6's samples A, B, C and D as illustrated in figure 4-2. for horizontal prediction, block 9's samples I, J, K, L are used. the rest of prediction modes utilize the block 6,7,9's samples in different ways based on mentioned equations in chapter 2.

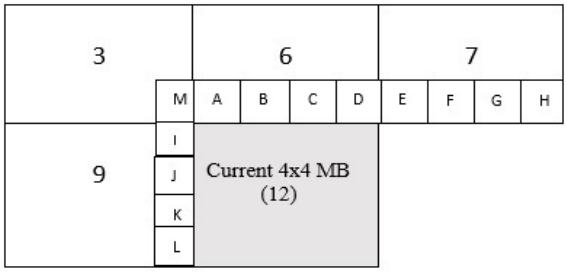


Figure 4-2 A 4x4 luma MB and neighbouring pixels and blocks

The number of possible intra 4x4 prediction modes for each 4x4 MB varies according to the MB position. For example, for above 4x4 MB upper reference pixels (A to H) are not available, so the available modes for these MBs are, DC, diagonal down left, horizontal up and vertical left. DC mode is always used regardless of the availability of the

neighboring pixels. Table 4.1 shows [36] the possible modes for each 4x4 luma MB based on availability of the neighboring pixels.

Table 4-1 Available 4x4 luma prediction modes

Availability of neighboring pixels	**Available 4x4 luma prediction modes**
Not available	DC
Upper available, left unavailable	DC, Vertical, Diagonal Down-Left, Vertical Left
Upper unavailable, left available	DC, Horizonal, Horizonal Up
Both upper and left available	All modes

In the proposed hardware design of the 4x4 Intra prediction, there are two settings 4 modes and 9 modes. Four modes, DC, horizontal, vertical, and Diagonal down right are implemented to reduce the hardware utilization. Nine modes. all available 4x4 intra prediction are implemented. Figure 4-3 presents the position of the pixels within a 4x4 MB. The pixel-by-pixel prediction equations used are shown in figure 4-4, pred[x,y] is the prediction for the pixel in the position [x,y].

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]
[3,0]	[3,1]	[3,2]	[3,3]

Figure 4-3 Position of pixels within a 4x4 MB

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[3,2] = C

Pred[0,3] = Pred[1,3] = Pred[2,3] = Pred[3,3] = D

(a) *4x4 vertical mode*

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) 4x4 horizontal mode

(If left and upper neighboring pixels are available)

Pred[x,y] = (A + B + C + D + I + J + K + L + 4) >> 3

(Else If only left neighboring pixels are available)

Pred[x,y] = (I + J + K + L + 2) >> 2

(Else If only upper neighboring pixels are available)

Pred[x,y] = (A + B + C + D + 2) >> 2

(Else If both neighboring pixels are not available)

Pred[x,y] = 128

(c) *4x4 DC mode*

Pred[0,0] = (A + 2*B + C + 2) >> 2

Pred[0,1] = (B + 2*C + D + 2) >> 2

Pred[0,2] = (C + 2*D + E + 2) >> 2

Pred[0,3] = (D + 2*E + F + 2) >> 2

Pred[1,0] = (B + 2*C + D + 2) >> 2

Pred[1,1] = (C + 2*D + E + 2) >> 2

Pred[1,2] = (D + 2*E + F + 2) >> 2

Pred[1,3] = (E + 2*F + G + 2) >> 2

Pred[2,0] = (C + 2*D + E + 2) >> 2

Pred[2,1] = (D + 2*E + F + 2) >> 2

Pred[2,2] = (E + 2*F + G + 2) >> 2

Pred[2,3] = (F + 2*G + H + 2) >> 2

Pred[3,0] = (D + 2*E + F + 2) >> 2

Pred[3,1] = (E + 2*F + G + 2) >> 2

Pred[3,2] = (F + 2*G + H + 2) >> 2

Pred[3,3] = (G + 3*H + 2) >> 2

(*d*) *4x4 diagonal down left mode*

Pred[0,0] = (A + 2*M + I + 2) >> 2

Pred[0,1] = (M + 2*A + B + 2) >> 2

Pred[0,2] = (A + 2*B + C + 2) >> 2

Pred[0,3] = (B + 2*C + D + 2) >> 2

Pred[1,0] = (M + 2*I + J + 2) >> 2

Pred[1,1] = (A + 2*M + I + 2) >> 2

Pred[1,2] = (M + 2*A + B + 2) >> 2

Pred[1,3] = (A + 2*B + C + 2) >> 2

Pred[2,0] = (I + 2*J + K + 2) >> 2

Pred[2,1] = (M + 2*I + J + 2) >> 2

Pred[2,2] = (A + 2*M + I + 2) >> 2

Pred[2,3] = (M + 2*A + B + 2) >> 2

Pred[3,0] = (J + 2*K + L + 2) >> 2

Pred[3,1] = (I + 2*J + K + 2) >> 2

Pred[3,2] = (M + 2*I + J + 2) >> 2

Pred[3,3] = (A + 2*M + I + 2) >> 2

(*e*) *4x4 diagonal down right mode*

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 + 2*M + A + 2) >> 2

Pred[1, 1] = (M + 2*A + B + 2) >> 2

Pred[1, 2] = (A + 2*B + C + 2) >> 2

Pred[1, 3] = (B + 2*C + D + 2) >> 2

Pred[2, 0] = (M + 2*I + 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 + 2*J + K + 2) >> 2

Pred[3, 1] = (I + 2*M + A + 2) >> 2

Pred[3, 2] = (M + 2*A + B + 2) >> 2

Pred[3, 3] = (A + 2*B + C + 2) >> 2

(f) 4x4 vertical right mode

Pred[0, 0] = (M + I + 1) >> 1

Pred[0, 1] = (I + 2*M + A + 2) >> 2

Pred[0, 2] = (B + 2*A + M + 2) >> 2

Pred[0, 3] = (C + 2*B + A + 2) >> 2

Pred[1, 0] = (I + J + 1) >> 1

Pred[1, 1] = (M + 2*I + J + 2) >> 2

Pred[1, 2] = (M + I + 1) >> 1

Pred[1, 3] = (I + 2*M + A + 2) >> 2

Pred[2, 0] = (J + K + 1) >> 1

Pred[2, 1] = (I + 2*J + K + 2) >> 2

Pred[2, 2] = (I + J + 1) >> 1

Pred[2, 3] = (M + 2*I + J + 2) >> 2

Pred[3, 0] = (K + L + 1) >> 1

Pred[3, 1] = (J + 2*K + L + 2) >> 2

Pred[3, 2] = (J + K + 1) >> 1

Pred[3, 3] = (I + 2*J + K + 2) >> 2

(g) 4x4 horizontal down mode

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 + 2*B + C + 2) >> 2

Pred[1, 1] = (B + 2*C + D + 2) >> 2

Pred[1, 2] = (C + 2*D + E + 2) >> 2

Pred[1, 3] = (D + 2*E + F + 2) >> 2

Pred[2, 0] = (B + C + 1) >> 2

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 + 2*C + D + 2) >> 2

Pred[3, 1] = (C + 2*D + E + 2) >> 2

Pred[3, 2] = (D + 2*E + F + 2) >> 2

Pred[3, 3] = (E + 2*F + G + 2) >> 2

(h) 4x4 vertical left mode

Pred[0, 0] = (I + J + 1) >> 1

Pred[0, 1] = (I + 2*J + K + 2) >> 2

Pred[0, 2] = (J + K+ 1) >> 2

Pred[0, 3] = (J + 2*K + L + 2) >> 2

Pred[1, 0] = (J + K+ 1) >> 1

Pred[1, 1] = (J + 2*K + L + 2) >> 2

Pred[1, 2] = (K + L + 1) >> 1

Pred[1, 3] = (K + 3*L + 2) >> 2

Pred[2, 0] = (K + L + 1) >> 1

Pred[2, 1] = (K + 3*L + 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) 4x4 horizontal up mode

Figure 4-4 Pixel-by-pixel prediction equations (a) 4x4 vertical mode, (b) 4x4 horizontal mode, (c) 4x4 DC mode, (d) 4x4 diagonal down left mode , (e) 4x4 diagonal down right mode, (f) 4x4 vertical right mode, (g) 4x4 horizontal down mode, (h) 4x4 vertical left mode, (i) 4x4 horizontal up mode

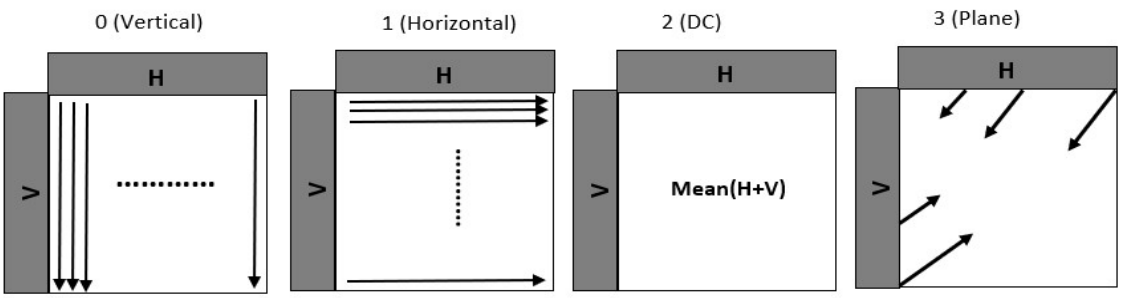


Figure 4-5 8x8 chroma prediction modes

The 8x8 intra chroma prediction modes are horizontal, vertical, DC, plane as shown in figure 4-5. The number of possible modes for each chroma component varies according to the MB position. For example, if the upper reference pixels are not available, the presented modes for these MBs are DC, horizontal. DC mode is always used regardless of the availability of the neighboring pixels. Table 4.2 shows the possible modes for each 8x8 chroma MB based on availability of the neighboring pixels.

Table 4.2 – Available 8x8 chroma prediction modes

Availability of neighboring pixels	**Available 4x4 luma prediction modes**
Not available	DC
Upper available, left unavailable	DC, Vertical
Upper unavailable, left available	DC, Horizonal
Both upper and left available	All modes

In the proposed hardware design of the 8x8 chroma prediction just 3 modes, DC, horizontal, and vertical are implemented. The pixel-by-pixel prediction equations used are shown in figure 4-6 pred[x,y] is the prediction for the pixel in the position[x,y].

Pred[y,0] = P[-1,0]

Pred[y,1] = P[-1,1]

Pred[y,2] = P[-1,2]

Pred[y,3] = P[-1,3]

Pred[y,4] = P[-1,4]

Pred[y,5] = P[-1,5]

Pred[y,6] = P[-1,6]

Pred[y,7] = P[-1,7]

(0 <= y <= 7)

(a)

Pred[0,x] = P[0,-1]

Pred[1,x] = P[1, -1]

Pred[2,x] = P[2, -1]

Pred[3,x] = P[3, -1]

Pred[4,x] = P[4, -1]

Pred[5,x] = P[5, -1]

Pred[6,x] = P[6, -1]

Pred[7,x] = P[7, -1]

(0 <= x <= 7)

(b)

(If P[x,-1] with x = 0…3 and P[-1,y] with y = 0…3 are available)

$$Pred[y,x] = (\sum P[x,-1] + \sum P[-1,y] + 4) \gg 3$$

(Else If P[x,-1] with x = 0…3 is available and P[-1,y] with y = 0…3 isn't available)

$$Pred[y,x] = (\sum P[x,-1] + 2) \gg 2$$

(Else If P[x,-1] with x = 0…3 isn't available and P[-1,y] with y = 0…3 is available)

$$Pred[y,x] = (\sum P[-1,y] + 2) \gg 2$$

(Else If both neighboring pixels are not available)

$$Pred[y,x] = 128$$

(0 <= x <= 3) , (0 <= y <= 3)

(c-1)

(If P[x,-1] with x = 4…7 is available)

$$Pred[y,x] = (\sum P[x,-1] + 2) \gg 2$$

(Else If P[-1,y] with y = 0…3 is available)

$$Pred[y,x] = (\sum P[-1,y] + 2) \gg 2$$

(Else)

$$Pred[y,x] = 128$$

(4 <= x <= 7) , (0 <= y <= 3)

(c-2)

(If P[-1,y] with y = 4…7 is available)

$$Pred[y,x] = (\sum P[-1,y] + 2) \gg 2$$

(Else If P[x,-1] with x = 0…3 is available)

$$Pred[y,x] = (\sum P[x,-1] + 2) \gg 2$$

(Else)

$$Pred[y,x] = 128$$

(0 <= x <= 3) , (4 <= y <= 7)

(c-3)

(If P[x,-1] with x = 4…7 and P[-1,y] with y = 4…7 are available)

$$Pred[y,x] = (\sum P[x,-1] + \sum P[-1,y] + 4) \gg 3$$

(Else If P[x,-1] with x = 4…7 is available and P[-1,y] with y = 4…7 isn't available)

$$Pred[y,x] = (\sum P[x,-1] + 2) \gg 2$$

(Else If P[x,-1] with x = 4…7 isn't available and P[-1,y] with y = 4…7 is available)

$$Pred[y,x] = (\sum P[-1,y] + 2) \gg 2$$

(Else If both neighboring pixels are not available)

$$Pred[y,x] = 128$$

(4 <= x <= 7) , (4 <= y <= 7)

(c-4)

Figure 4-6 (a) 8x8 Vertical mode (b) 8x8 Horizontal mode (c) 8x8 DC mode

4.2 Proposed Hardware for 4x4 Luma and 8x8 Chroma Prediction Modes

There are common parts in the equations of the 4x4 luma prediction modes. We reorganize the equation to exploit these common parts and reduce the required computation time and hardware usage. Figure 4.7 shows the FPGA block diagram of intra prediction, it consists of input video flow controller and predictor block, controller

unit, SAD block, and comparator. The blocks coded using VHDL (VHSIC Hardware Description Language). Input video flow control block takes 16 lines of video data and store them in RAM (Random Access Memory) of FPGA internal memory. The reason 16 lines of video data are stored is that a 16x16 MB is the main processing unit for the H.264 video coding and in order to start the encoding process one MB must be captured. The input video flow control block reads one 16x16 MB and passes it to the 4x4 prediction calculation blocks. The 16x16 MB's location information will also be transmitted to the 4x4 prediction blocks, allowing us to determine whether it resides along the upper or left border of the video frame. Two RAMs are incorporated into the design, with the second 16x16 MB being stored during the processing of the first 16x16 MB.

The Controller unit manages the 4x4 intra prediction process by taking a 16x16 MB and dividing it into 4x4 blocks. This block transmits data to the prediction unit, along with *hflag* and *vflag* signals that determine the available prediction modes and the neighboring pixels required for prediction calculations. Next, the Predictor block acquires the 4x4 blocks of data and computes the residual data for the available prediction modes by utilizing the original pixels and their neighboring top and/or left pixels. Initially, this block calculates the common parts and stores them in registers. Afterward, it concurrently measures the difference between the source pixels and the pixels located above and/or to the left of the 4x4 block for all available prediction modes. If both the top and left pixels are available, there's a requirement to compute either all 4 modes or 9 modes, depending on the available hardware settings. Generating predicted pixels for these modes takes 31 clock cycles, as detailed in table 4.2. Considering that each 4x4 MB contains 16 pixels,

and we process 4 pixels simultaneously, the intra prediction calculation requires fewer clock cycles. We save the top and left pixels in separate RAMs. Following that, the SAD block receives the sum of absolute differences (SAD) for the available predicted modes and measures the totalSAD for each mode. Then, the Comparator block compares all modes and selecting the one with the minimum totalSAD. This chosen mode and 4x4 predicted pixel block are the final outputs.

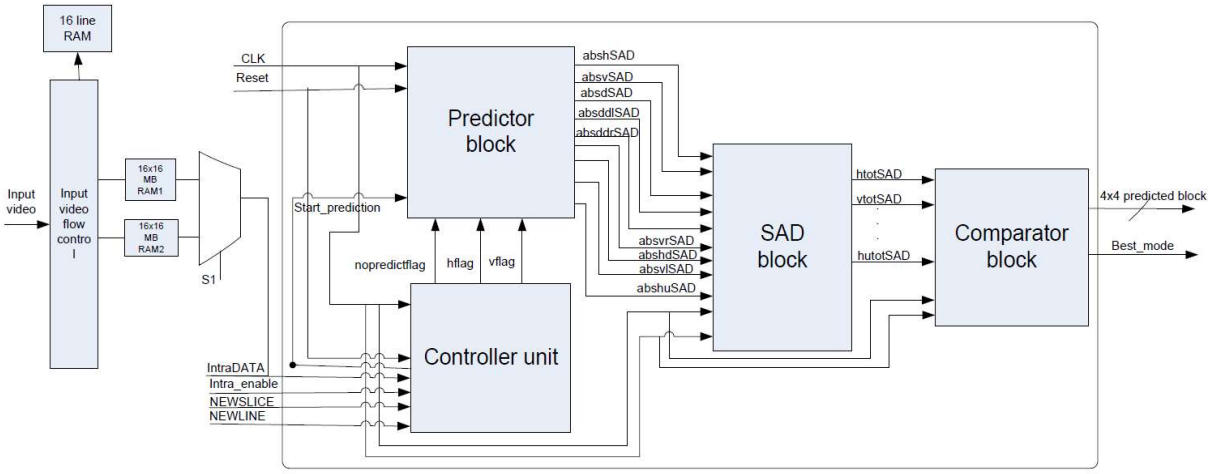


Figure 4-7 4x4 intra prediction hardware architecture

Table 4.3 Required clock cycles for available 4x4 luma prediction modes

Preprocessing and available prediction modes	**Clock cycles/4x4 luma block**
Preprocessing	13
Parallel available mode calculation	18

Table 4.4 Required clock cycles for available 8x8 chroma prediction modes

Preprocessing and available prediction modes	**Clock cycles/MB**
Preprocessing	32
Parallel available mode calculation	18

The hardware architecture proposed for 8x8 chroma prediction is similar the one introduced for 4x4 luma prediction. We've incorporated horizontal, vertical, and DC modes, depending on the availability of neighboring pixels. However, we did not implement the plane mode due to its computational complexity. It takes 50 clock cycles to generate the predicted pixels for the available modes as given in table 4.3.

CHAPTER FIVE

4x4 INTEGER TRANSFORM AND QUANTIZATION HARDWARE ARCHITECTURE AND FPGA IMPLEMENTATION

After the prediction block, Transform and quantization are applied. These two blocks reduce the spatial redundancy of the prediction error signal. The previous encoding standards used an 8x8 2D-DCT (Discrete Cosine Transform). However, the transform block in H.264 is a scaled approximation of the 4x4 DCT, and it can be computed using simple integer arithmetic. Quantization is then applied after this transformation to reduce the dynamic range of the transform coefficients. The output of quantization is a 4x4 block with mostly zero coefficients and a few non-zero coefficients or all zero coefficients. This chapter is outlined as follows. section 5.1 explains the FPGA implementation of 4x4 integer transform. Section 5.2 presents FPGA implementation of the 4x4 quantization block. Finally, section 5.3 provides the hardware usage and timing of the transform and quantization implementation.

5.1 FPGA Implementation of 4x4 Integer Transform

This section explains the FPGA hardware architecture for the 4x4 integer transform, which was implemented using VHDL. The 2D-DCT transformation equation 2.18 from chapter 2 was restructured into two components: the 1D-vertical transform and the 1D-horizontal transform, as shown in figure 5-1 [32]. The 4x4 input block initially processed in the vertical block, followed by the application of the horizontal block to obtain the

results. The Controller unit change the sequence of 1-D Vertical Transform output and 1-D Horizontal Transform input.

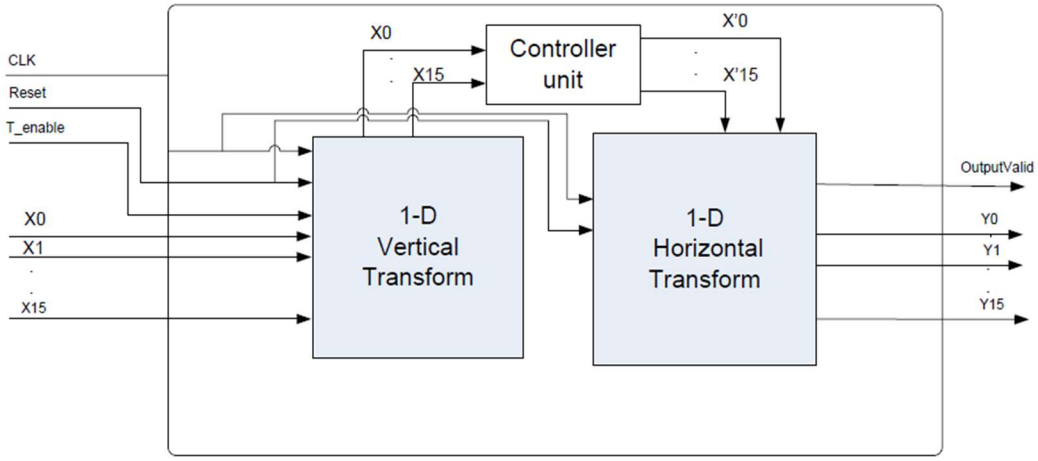


Figure 5-1 FPGA block diagram of 4x4 integer transform hardware architecture.

Figure 5-2 illustrates the fast implementation of the 1-D integer transform [33]. This implementation utilizes only additions and shifts without involving multiplications to reduce the complexity of the architecture. In VHDL, the coefficient 2 in the figure can be achieved by performing a left shift by one.

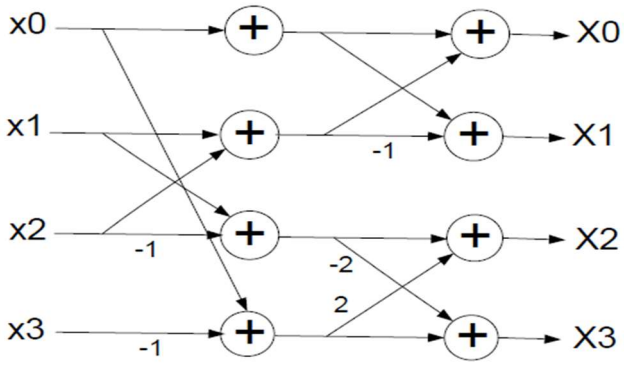


Figure 5-2 Fast implementation of 1-D integer transform

5.2 FPGA Implementation of the 4x4 Quantization Block

The quantization block comes after transformation block. Figure 5-3 shows the H.264 quantization block diagram as outlined in equation 5.1 with the parameters MF = 13107, qbits = 15 + floor(QP/6) .

$$Y_{ij} = sign(W_{ij})(|W_{ij}|MF_{ij} + f) \gg qbits \qquad \text{Eq. 5.1}$$

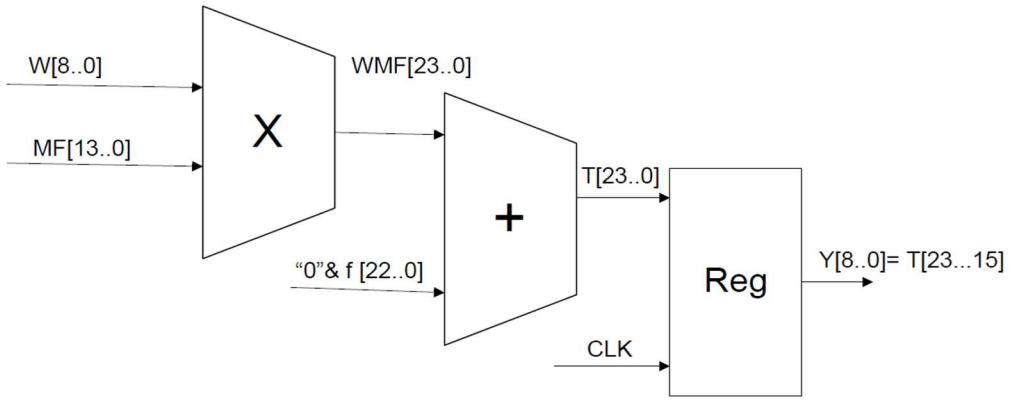


Figure 5-3 Block diagram of H.264 quantization with $\mathbf{MF/2^{qbits}}$ = 13107/$\mathbf{2^{15+floor(QP/6)}}$

As shown in the figure, there is a pipeline consisting of three blocks: multiplier, adder, and shift register. Prior to the initiation of the quantization process, the sign of each transform coefficient is saved in a register and subsequently concatenated with the output signal at the final block. The multiplier block multiplies the absolute value of the transform coefficients by multiplication factor *MF*. Following that, the adder block adds the result from the multiplier with the rounding factor *f*. Finally, the last block performs a right shift on the adder block's output, with the *qbits* size, which in this case is set to 15 bits.

The multiplication block, originally designed as a 14-bit by 9-bit multiplier, can be replaced with adders and shifters to reduce delay and complexity. However, given the large numbers involved in this quantization process, this alternative approach required a substantial number of adders and shifters, which isn't the most hardware-efficient implementation. Therefor In their work, Zhang et al [44] introduced a modification to the quantization process to lower high computational precision and complexity while still achieving acceptable accuracy shown as in equation 5.2.

$$qbits' = 5 + floor(QP/6) \qquad \text{Eq. 5.2}$$

Table 5-1 shows the modified MF for 0<QP<5 [44]

Table 5-1 Modified $\mathbf{MF_{ij}'/2^{qbits'}}$

QP(i,j)	(0,0),(2,0),(2,2),(0,2)	(0,0),(2,0),(2,2),(0,2)	Other positions
0	$13/2^5$	$5/2^5$	$1/2^2$
1	$3/2^3$	$9/2^6$	$7/2^5$
2	$5/2^4$	$1/2^3$	$13/2^6$
3	$9/2^5$	$7/2^6$	$3/2^4$
4	$1/2^2$	$7/2^6$	$5/2^5$
5	$7/2^5$	$3/2^5$	$9/2^6$

The block diagram of improved quantization for $\frac{MF'}{2^{qbits'}} = 13/2^{15+floor(QP/6)}$ is shown as figure 5-4 [44] .

In this hardware architecture, there's a pipeline consisting of three adders and one shift register. Before initializing the quantization process, the sign of transform coefficients is saved in a register and later concatenated at the final block to form the output signal. To replace the multiplier block, we've implemented two adders: the first adder adds two shift-adjusted transform coefficients, and the second adder sums the output of the first

adder with another shift-adjusted transform coefficient. The third adder adds the output of second adder with the rounding factor, denoted as *f*. Finally, the last block applied a right shift on the adder block's output, with a qbits size of 5 bits.

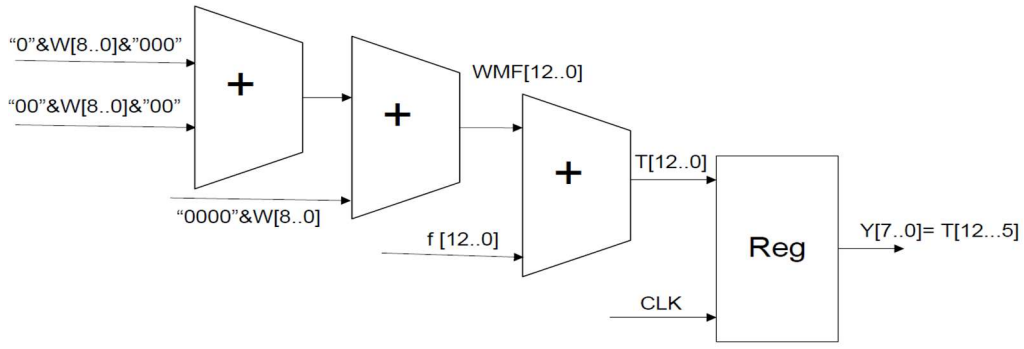


Figure 5-4 Block diagram of improved quantization with $\frac{\mathbf{MF'}}{\mathbf{2^{qbits'}}} = \mathbf{13/2^{15+floor(QP/6)}}$

The enhanced quantization approach utilizes less hardware resources and power consumption while achieving shorter processing delays. In our quantization hardware architecture, we used the block diagram in figure 5-4 as the foundation for block Q. By utilizing four instances of block Q in parallel, we can quantize an entire 4x4 pixel block simultaneously, as illustrated in figure 5.5. As shown in figure 5-5, the entire transform and quantization process operates as a two-step pipeline. In the first step, the predicted residual enters the transform block, and it takes 7 clock cycles to produce the transform coefficients. In the second step, the 16 coefficients generated in the first step are divided by 4, with each set of 4 coefficients feeding into a respective Q block. The process of generating the quantized output requires just 4 clock cycles. Therefore, the transformation and quantization process can be accomplished within 11 clock cycles, leading to a substantial reduction in delay along the H.264 stream path.

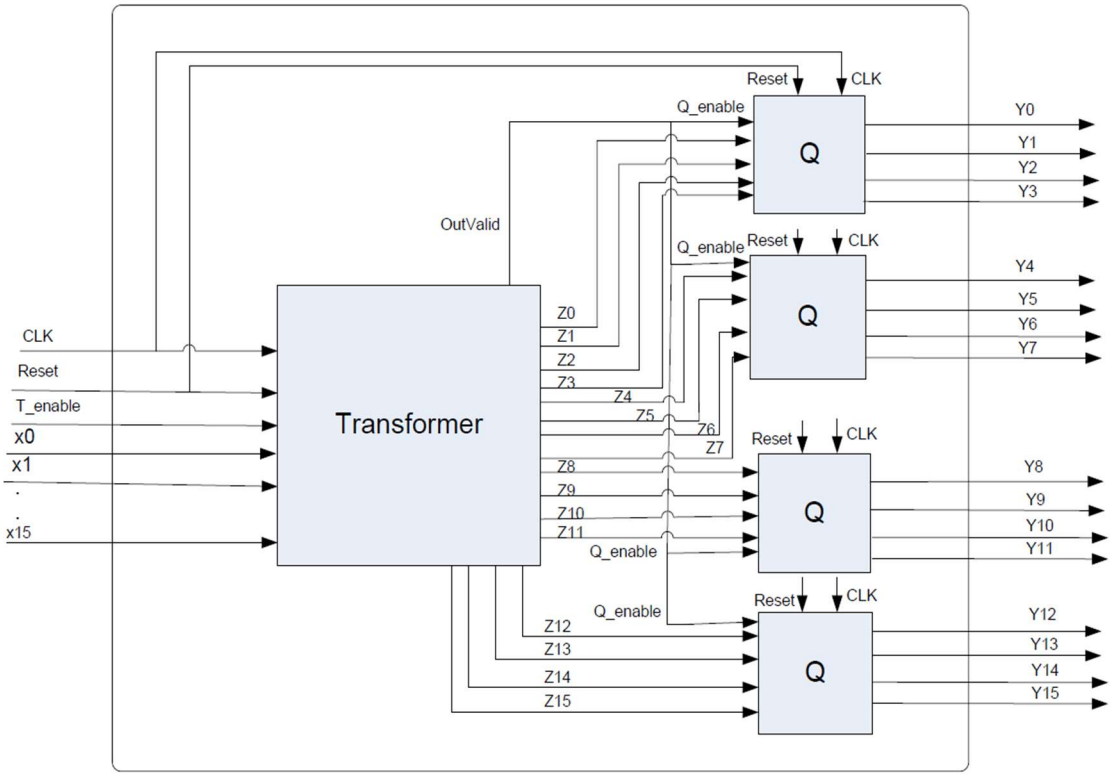


Figure 5-5 The transform and quantization FPGA block diagram

CHAPTER SIX

PARALLEL CAVLC HARDWARE ARCHITECTURE AND FPGA IMPLEMENTATION

Context-adaptive variable-length coding (CAVLC) is the final component of H.264, following the intra prediction, transformation, and quantization blocks. This lossless technique is employed to produce the H.264 output bitstream, as detailed in Chapter 2. It encodes quantized coefficients in both 4x4 and 2x2 blocks in zig-zag order through the utilization of several subblocks, including T1sCalculator, TotalCoefCounter, TotalZeroCounte, Run_beforeCalculator, and LevelCalculator. The CAVLC algorithm delivers better encoding outcomes, but this advantage comes at the expense of increased computational complexity. As a result, achieving real-time implementation of the algorithm can be more challenging.

This chapter is outlined as follows. section 6.1 explains the FPGA implementation of T1sCalculator, TotalCoefCounter, TotalZeroCounter, Section 6.2 covers the FPGA implementation of Run_beforeCalculator, LevelCalculator. Finally, section 6.3 provides the hardware usage and timing details of the CAVLC implementation.

6.1 FPGA Implementation of CAVLC Block

To improve the speed of CAVLC, we reduced the processing cycles for both the 4x4 and 2x2 blocks by increasing parallelism. We introduced a two-stage architecture that allows the scanning and coding phases to work simultaneously. Building upon Zhong et al.'s design [45], our architecture can scan and code four coefficients concurrently, as

shown in figure 6.1. The CAVLC block is implemented in VHDL, and further specifics of this architecture are given in the following sections.

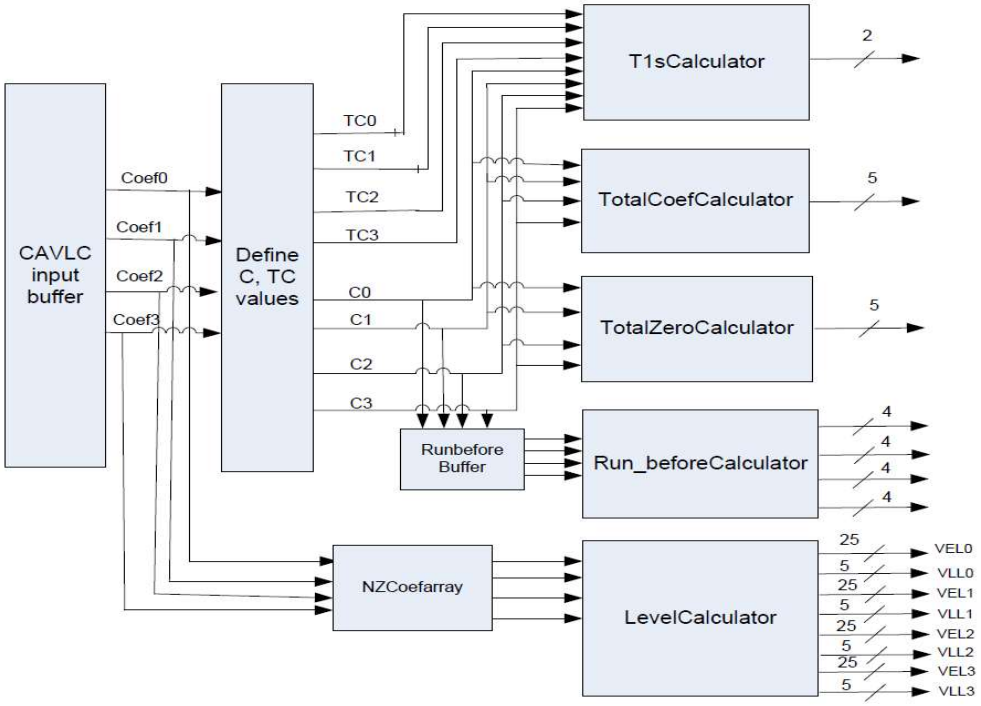


Figure 6-1 FPGA block diagram of CAVLC hardware architecture

6.1.1 Reverse Zigzag Reading of Coefficients and C, TC Calculation

As illustrated in figure 6-1, the CAVLC input buffer block simultaneously reads four coefficients from a 4x4 block in zig-zag order. These coefficients are then transmitted as Coef0, Coef1, Coef2, and Coef3 to the subsequent block, referred to as Define C, TC values. Define C. The Define C, TC values block assigns values to C0-C3 and TC0-TC3 as follows:

For C0-C3:

-If Coef is greater than 1, then C is set to 1.

- Otherwise, C is set to 0.

For TC0-TC3:

- If Coef is equal to +1 or -1, then TC is set to 1.

- Otherwise, TC is set to 0.

Using the C and TC values as a basis, CAVLC computes several counters to save information about the 4x4 pixel block:

1. *T1sCounter* records the number of coefficients with ±1 values.
2. *TotalCoefCounter* keeps track of the total number of non-zero coefficients within the 4x4 block.
3. *TotalZeroCounter* stores the total number of zeros preceding the last non-zero coefficient.

This process takes 4 clock cycles to read all coefficients of the 4x4 block and calculate the C and TC values.

6.1.2 Hardware Implementation of Encoding totalcoef

The TotalCoefCalculator block takes inputs from TotalCoefCounter and T1sCounter (referred to as trailingones) to encode the variable length coeff_token using VLC tables. There are four VLC tables for generating coef_token, and the selection of the appropriate table is determined by the values of *nC*, as detailed in table 6.1. Having multiple VLC tables enhances the coding performance of the CAVLC algorithm. The parameter *nC* is

calculated based on the number of non-zero coefficients in the left-hand and upper previously blocks, denoted as *nA* and *nB*, respectively. Specifically:

- If both *nA* and *nB* are available, then *nC* is set to the rounded average of (*nA* + *nB*) / 2.
- If only *nA* is available, *nC* = *nA*.
- If only *nB* is available, *nC* = *nB*.
- If neither *nA* nor *nB* is available, *nC* = 0.

For 2x2 dc chroma blocks, *nC* is consistently set to -1. These tables are implemented in VHDL as lookup tables.

Table 6-1 VLC table for coeff_token

nC	**VLC Table**
0,1	Table 1
2,3	Table 2
4,5,6,7	Table 3
8 or above	Table 4

6.1.3 Hardware Implementation of Encoding T1s and Totalzero

The *TotalZeroCalculator* block computes the total number of zeros before the last non-zero coefficient using the signals *C0-C3* in combination with a counter. For example, when *C0* equals 1 (indicating that *Coef0* is a non-zero coefficient) and *C1* equals 0 (showing that *Coef1* is a zero coefficient), the counter is incremented by one.

CAVLC uses two total_zeros tables—one for the luma component and another for the chroma components (refer to the appendix) to encode a variable-length totalzero. These tables utilize the totalcoeffs and total_zeros counters, which were calculated in previous steps for either the 4x4 or 2x2 block, to generate the ztoken. In our VHDL implementation, we considered a fixed length of 3 bits for ztoken and utilized a ztoken_len to define its length. For example, according to the luma total_zeros table, if totalzeros equals 0 and totalcoeffs equals 1, then in our code implementation, ztoken is represented as b"001," and ztoken_len is defined as b"0001". The total_zeros tables is incorporated into our code implementation as lookup tables, as shown in figure 6-2.
The *T1sCalculator* block utilizes the *TC0-TC3* and *C0-C3* signals to encode trailing ones and produce an output referred to as *T1sEncode*. This encoding is variable-length, with a maximum value of 3. Specifically, when TC equals 1 for a corresponding non-zero coefficient that is either +1 or -1, the following encoding rules apply:

- For +1, we add 0 into T1sEncode.
- For -1, we add 1 into T1sEncode.

For instance, if the trailing ones are +1, +1, -1, the resulting *T1sEncode* would be 001.

In our VHDL implementation, we simultaneously check *TC0-TC3* and *C0-C3* in a single clock cycle, allowing us to check all 4x4 coefficients within 4 clock cycles. Figure 6-3 provides an example of the *T1sEncode* calculation.

```
ztoken <=
       b"001" when totalzeros=0 and totalcoeffs=1 and ztable='0' else
       b"011" when totalzeros=1 and totalcoeffs=1 and ztable='0' else
       b"010" when totalzeros=2 and totalcoeffs=1 and ztable='0' else
       b"011" when totalzeros=3 and totalcoeffs=1 and ztable='0' else
       b"010" when totalzeros=4 and totalcoeffs=1 and ztable='0' else
       b"011" when totalzeros=5 and totalcoeffs=1 and ztable='0' else
       b"010" when totalzeros=6 and totalcoeffs=1 and ztable='0' else
       b"011" when totalzeros=7 and totalcoeffs=1 and ztable='0' else
       b"010" when totalzeros=8 and totalcoeffs=1 and ztable='0' else
       b"011" when totalzeros=9 and totalcoeffs=1 and ztable='0' else
       b"010" when totalzeros=10 and totalcoeffs=1 and ztable='0' else
       b"011" when totalzeros=11 and totalcoeffs=1 and ztable='0' else
       b"010" when totalzeros=12 and totalcoeffs=1 and ztable='0' else

ztoken_len <=
       b"0001" when totalzeros=0 and totalcoeffs=1 and ztable='0' else
       b"0011" when totalzeros=1 and totalcoeffs=1 and ztable='0' else
       b"0011" when totalzeros=2 and totalcoeffs=1 and ztable='0' else
       b"0100" when totalzeros=3 and totalcoeffs=1 and ztable='0' else
       b"0100" when totalzeros=4 and totalcoeffs=1 and ztable='0' else
       b"0101" when totalzeros=5 and totalcoeffs=1 and ztable='0' else
       b"0101" when totalzeros=6 and totalcoeffs=1 and ztable='0' else
       b"0110" when totalzeros=7 and totalcoeffs=1 and ztable='0' else
       b"0110" when totalzeros=8 and totalcoeffs=1 and ztable='0' else
       b"0111" when totalzeros=9 and totalcoeffs=1 and ztable='0' else
       b"0111" when totalzeros=10 and totalcoeffs=1 and ztable='0' else
       b"1000" when totalzeros=11 and totalcoeffs=1 and ztable='0' else
       b"1000" when totalzeros=12 and totalcoeffs=1 and ztable='0' else
       b"1001" when totalzeros=13 and totalcoeffs=1 and ztable='0' else
       b"1001" when totalzeros=14 and totalcoeffs=1 and ztable='0' else
       b"1001" when totalzeros=15 and totalcoeffs=1 and ztable='0' else
       b"0011" when totalzeros=0 and totalcoeffs=2 and ztable='0' else
```

Figure 6-2 total_zeros table look-up table VHDL implementation

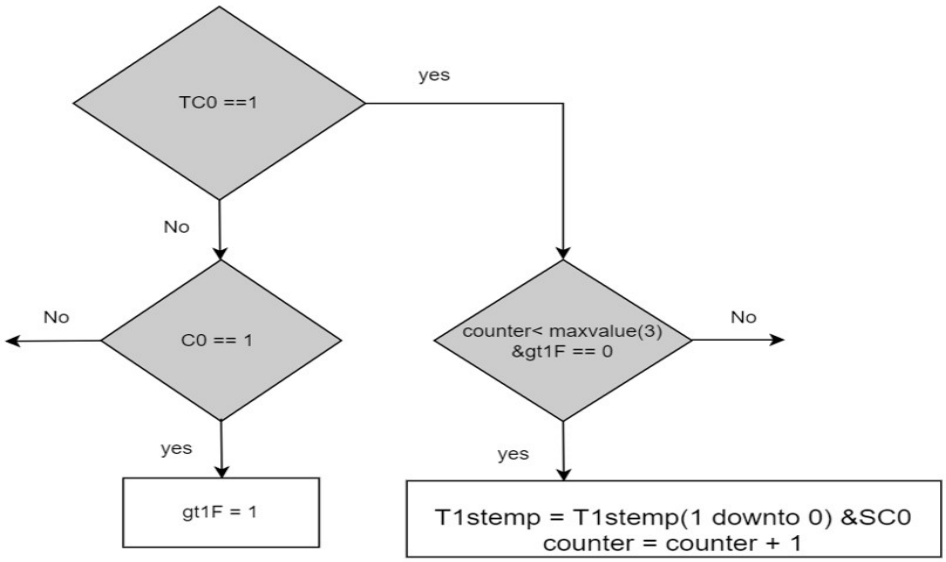


Figure 6-3 Block diagram of T1sEncode calculation

6.1.4 Hardware Implementation of Encoding Runbefore

Runbefore is another variable-length encoding method that encodes the number of zeros using two parameters: *run_before* and *zerosLeft*. Run_before is the number of zeros preceding each non-zero coefficient, while *zerosLeft* denotes the total number of zeros located between the current non-zero coefficient and the last non-zero coefficient. Since the coefficients are mostly zeros after transformation and quantization blocks, this technique significantly enhances the compression capability of H.264.

As shown in figure 6.1, there is a RunbeforeBuffer block responsible for storing runbefore values, according to the diagram outlined in figure 6-4. Subsequently, the *Run_beforeCalculator* block reads these runbefore values from the buffer and calculates *zerosleft* and *runb* parameters as 6.1 and 6.2 equations.

$$rbzerosleft = TotalZeroCount - RunbeforeBuffer(index-1) \quad \text{Eq. 6.1}$$

$$runb = RunbeforeBuffer(index) - RunbeforeBuffer(index-1) \quad \text{Eq. 6.2}$$

Following the calculation of these two parameters, they are used to generate the rb parameter through the utilization of a lookup table, as illustrated in figure 6-5. The rb parameter is then utilized in producing the runbefore codeword (*vrbe, vrbl*). Figure 6-6 provides a detailed FPGA implementation of the Run_before Calculator, consisting of 8 blocks: *Runb&Rbzerosleft Calculator*, *Run encode 0*, *Run encode 1*, *Run encode 2*, *Run encode 3*, *rbe controller unit*, *rbe Packer*, and *Zero/rbe Packer*. This configuration simultaneously generates 4 vrbe values, which are then organized and packed together based on their corresponding vrbl lengths. The final block, Zero/rbe Packer, combines the totalzeros value with the rbe parameter.

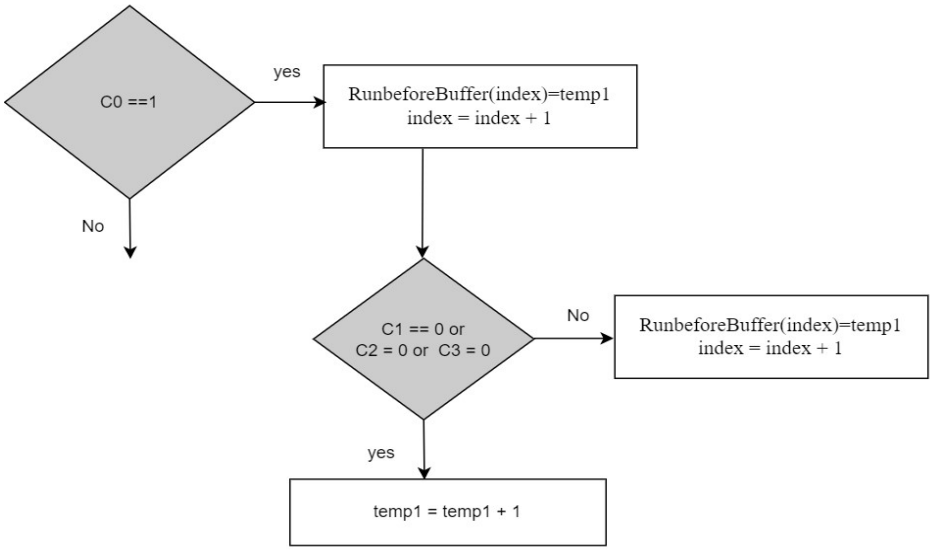


Figure 6-4 Block diagram of storing & calculating runbefore values

```
rb0  <=
        b"111" when runb0=0 else
        b"000" when runb0=1 and rbzerosleft0=1 else
        b"001" when runb0=1 and rbzerosleft0=2 else
        b"010" when runb0=1 and rbzerosleft0=3 else
        b"010" when runb0=1 and rbzerosleft0=4 else
        b"010" when runb0=1 and rbzerosleft0=5 else
        b"000" when runb0=1 and rbzerosleft0=6 else
        b"110" when runb0=1 else
        b"000" when runb0=2 and rbzerosleft0=2 else
        b"001" when runb0=2 and rbzerosleft0=3 else
        b"001" when runb0=2 and rbzerosleft0=4 else
        b"011" when runb0=2 and rbzerosleft0=5 else
        b"001" when runb0=2 and rbzerosleft0=6 else
        b"101" when runb0=2 else
        b"000" when runb0=3 and rbzerosleft0=3 else
        b"001" when runb0=3 and rbzerosleft0=4 else
        b"010" when runb0=3 and rbzerosleft0=5 else
        b"011" when runb0=3 and rbzerosleft0=6 else
        b"100" when runb0=3 else
        b"000" when runb0=4 and rbzerosleft0=4 else
        b"001" when runb0=4 and rbzerosleft0=5 else
        b"010" when runb0=4 and rbzerosleft0=6 else
        b"011" when runb0=4 else
        b"000" when runb0=5 and rbzerosleft0=5 else
        b"101" when runb0=5 and rbzerosleft0=6 else
        b"010" when runb0=5 else
        b"100" when runb0=6 and rbzerosleft0=6 else
        b"001";
```

Figure 6-5 run_before table look-up table VHDL implementation

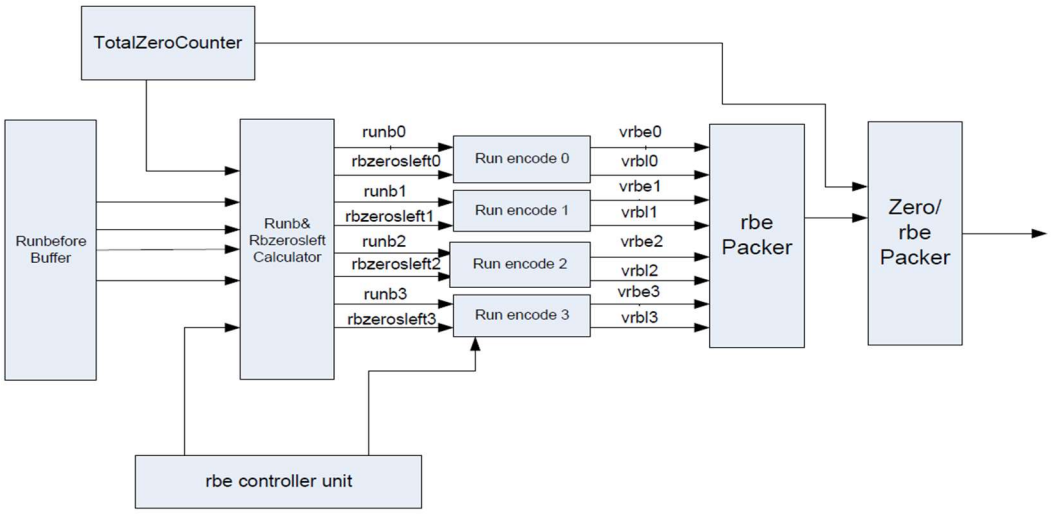


Figure 6-6 Runbefore calculator FPGA implementation in more details

6.1.5 Hardware Implementation of Encoding Level

In the CAVLC encoding process, a maximum of three coefficients with values of ±1 is accounted for T1s. The remaining non-zero coefficients are encoded using the variable-length level encoding technique. This encoding starts in reverse order, beginning from the highest frequency and proceeding towards the DC coefficients. The codeword generation of each level consists of both a prefix and a suffix. The length of the suffix for the current level depends on the magnitude of the previously coded level, which makes the parallelism difficult. In our proposed architecture for level encoding, we concurrently calculate four levels, using a simplified equation for suffix generation. The generation of the suffix takes just one clock cycle.

In CAVLC, the non-zero coefficient values, along with their respective signs, are stored in the NZCoefarray, as shown in figure 6-1. During this process, C0-C3 values are

assigned either 0 or 1 based on the coefficients' values. To initiate the level encoding, we need to know both the totalcoeffs and trailingones.

We have n = (totalcoeffs – trailingones) numbers of levels to encode. A 4-bit load register is utilized to determine how many level calculators' blocks are required.

For example, if n equals 7, during the first clock cycles, we can calculate 4 levels in parallel, and during the second clock cycle, we calculate 3 levels. In the worst-case

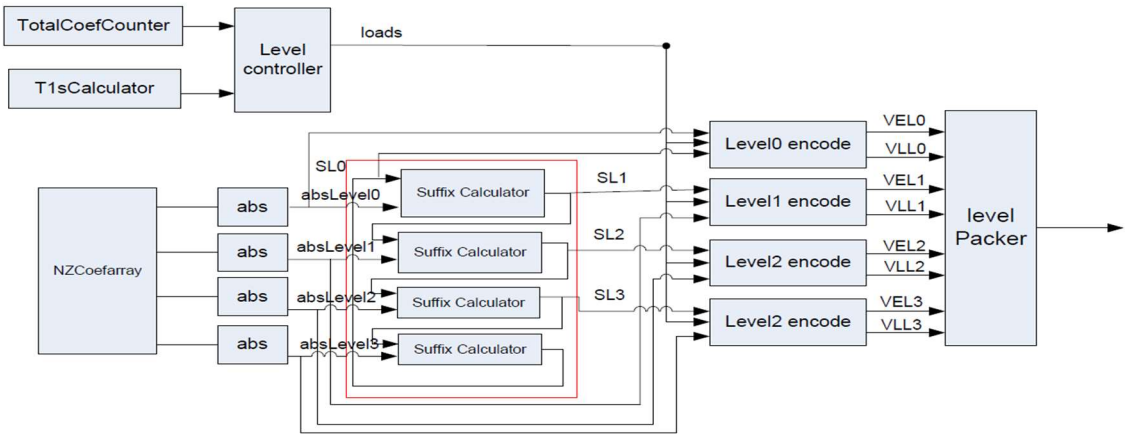


Figure 6-7 Level encoding FPGA implementation in more details

scenario, we have a 4x4 block of non-zero coefficients with n equal to 16, which requires 4 clock cycles to complete the entire block level encoding. Figure 6-7 shows FPGA implementation of the level encoding in more details. It contains 10 blocks: the level controller, four suffix calculators, four level encode blocks, and a level packer. Within this configuration, it can concurrently generate four VELs, and these VELs are packed together based on their corresponding VLL lengths.

6.1.6 Packing VLC Variable-length Output

The H.264 output bitstream should be in the sequence given in table 6-2.

Table 6-2 H.264 bitstream

Coeff_token	T1s	Level	TotalZeros	Run_before

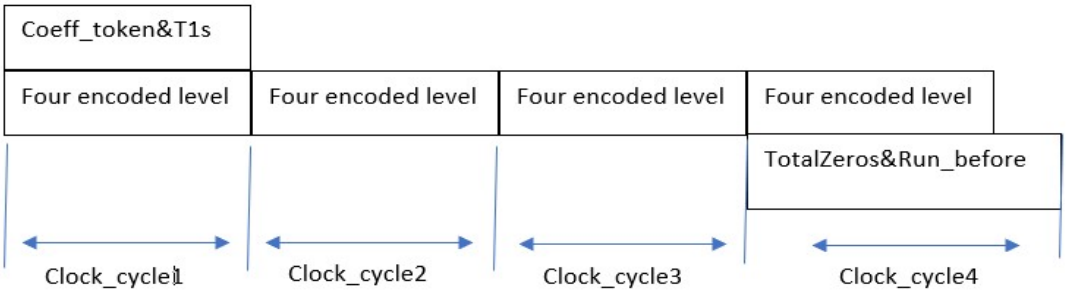


Figure 6-8 Timing diagram of CAVLC packing process

As previously mentioned, CAVLC encodes several parameters such as coeff_token, T1s, Level, Totalzeros, and Run_before using variable-length encoding. However, in hardware implementation, the output register has a fixed size, in our design is 32 bits. To adjust this fixed size, we need to pack the CAVLC output efficiently. Figure 6-8 provides a detailed timing diagram of the packing process, which takes 4 clock cycles. In the first clock cycle, we pack the coeff_token and T1s together with four encoded levels. During the second and third clock cycles, we incorporate four encoded levels each. Finally, in the last clock cycle, we combined the Totalzeros and Run_before with the four encoded levels.

CHAPTER SEVEN

FPGA IMPLEMENTATION AND RESULTS

This chapter discusses different settings and configurations of the proposed hardware architecture of H.264 encoder. Also, the VHDL implementation timing and test results will be illustrated.

It is outlined as follows: section 7.1 explains the proposed hardware architecture of H.264, Section 7.2 presents the implementation test results.

7.1 The Proposed Hardware Architecture of H.264

Figure 5 shows the proposed hardware architecture for H.264, it contains a total of 13 blocks and components: three SRAMS, four Sparse controllers, a 4x4 block creator, Intra prediction for both 4x4 and 8x8 blocks, Mode selector, Transformation, Quantization, CAVLC, and Header. In our design, we efficiently stored 16x16 pixel MBs of the luma component as well as 8x8/16x16 pixel MBs of the chroma components within the SRAMs. This design operates without the need for external memory. It offers two different settings: one with compression and the other without compression. In the without compression setting, we bypassed the Intra prediction, integer transform, quantization, and CAVLC blocks. The encoder output just has an H.264 stream format. This configuration of our architecture minimizes hardware usage, making it suitable for scenarios where there is a very low hardware resources available. Our architecture is highly parametrized, allowing it to adapt various pixel sizes in the without compression setting. In the with compression setting, it supports pixel sizes of 8, 10, and 12 bits, as

well as multiple input formats, including YUV 4:2:0, 4:4:4, and 4:2:2. To have a 16x16 pixel macroblock (MB), such as with pixel sizes of 10 bits or 8 bits, we require 320 bytes and 256 bytes, respectively. To achieve this, we introduced sparse controllers both at the input and output stages of the luma and chroma blocks. This arrangement extracts the necessary variable bytes for each 16x16 MB, as illustrated in figure 7-1.

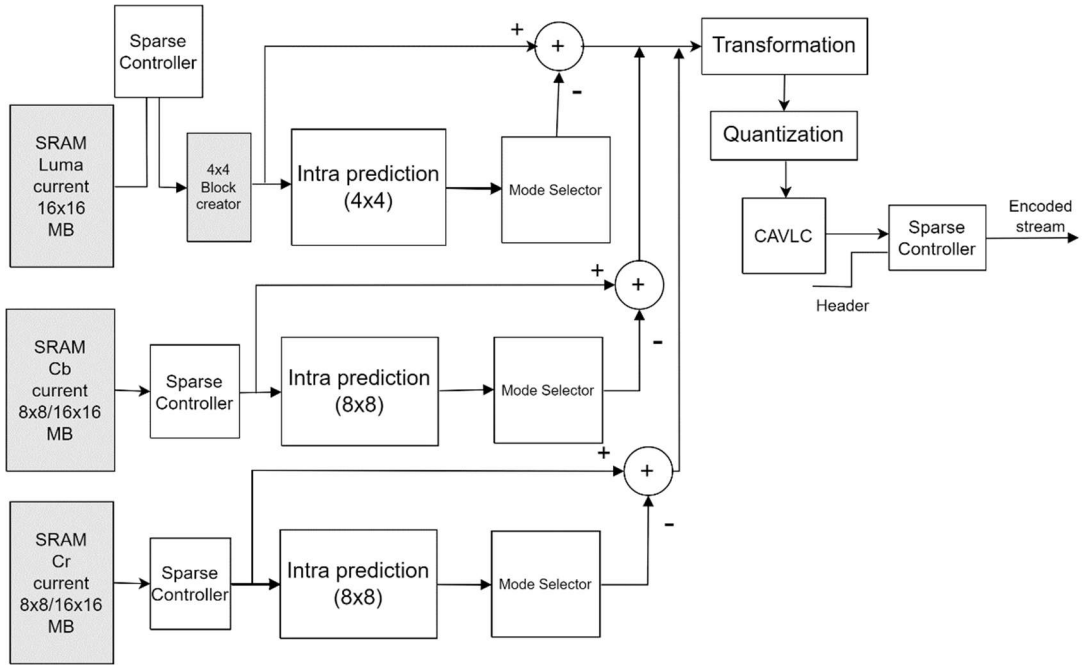


Figure 7-1 Top level block diagram of proposed H.264 encoder

To reduce the design complexity and the bottleneck in intra prediction, we skipped the loop of inverse transform and inverse quantization. However, to mitigate the error caused by this omission, we adopt the approach proposed by Wang et al. We modify the error cost function by introducing an error term, a/b(51-QP), where a and b are set to 80 and 1.07, respectively, as suggested by Wang et al. These values have proven to achieve desired outcomes for 4x4 intra prediction in our experimentation.

7.2 Implementation Test Results

Initially, each block was individually implemented in MATLAB. This approach was used to validate the algorithms. MATLAB offers a convenient environment for working with video inputs and outputs while simplifying the coding process compared to VHDL. Once we gained a full understanding of how the algorithm works, we transitioned to VHDL for implementation. In VHDL, we explored various hardware optimization techniques, such as resource sharing across different components and parallel processing, aiming to minimize hardware utilization and reduce latency. We verified the implementation through RTL and timing simulations using the Vivado simulator. Subsequently, the VHDL RTL was synthesized to target multiple FPGA devices including Zynq Z-7030 FPGA, Artix-7 XC7A100T FPGA, and Spartan-7 XC7S100 FPGA. The input video used for testing contains two formats: Foreman YUV 4:2:0 in CIF resolution and Elephants Dream YUV in Full HD 1080p. The hardware cost in terms of K logic gates, compression ratio, and frequency for various configurations of implementing 3-mode intra prediction are summarized in Table 7-1. Additionally, Table 7-2 presents the results when 9 modes of intra prediction are implemented. By bypassing compression blocks, the corresponding outcomes are listed in Table 7-3.

Table 7-1 Three-mode settings – with compression

Profile	Xilinx product family	Frequency	Hardware usage	Compression ration	Throughput
Intra High	Zynq 7000	159 MHZ	31 KGates	2	2 Gbit/s
Intra High	Artix 7	142 MHZ	32 KGates	2	1.8 Gbit/s
Intra High	Spartan 7	135 MHZ	33 KGates	2	1.7 Gbit/s

Table 7-2 Nine-mode settings – with compression

Profile	Xilinx product family	Frequency	Hardware usage	Compression ration	Throughput
Intra High	Zynq 7000	158 MHZ	50 KGates	2.7	2 Gbit/s
Intra High	Artix 7	131 MHZ	52 KGates	2.7	1.6 Gbit/s
Intra High	Spartan 7	115 MHZ	51.5 KGates	2.7	1.4 Gbit/s

Table 7-3 Without compression settings

Profile	Xilinx product family	Frequency	Hardware usage	Compression ration	Throughput
Intra High	Zynq 7000	183 MHZ	18 KGates	---	2.3 Gbit/s
Intra High	Artix 7	165 MHZ	19 KGates	---	2 Gbit/s
Intra High	Spartan 7	148 MHZ	20 KGates	---	1.8 Gbit/s

Table 7-4 Comparison with other work's results

Design Criteria	[12]	[13]	[14]	[15]	[16]	[17]	Achieved result
Profile	High	High, SVC	High	Baseline	Baseline	Baseline	Intra High
Frequency (MHZ)	130	120(high), 166 (SVC)	162	81(SD); 180 (HD)	N/A	28 (CIF);108 (HD720)	159
Hardware Utilization	37178 ALUTs & 128 DSP	2079 KGates	3745 KGates	922.8 KGates	452.8 KGates	470 KGates	31 KGates
Target video resolution	0.5Gbit/s	0.5Gbit/s	0.5Gbit/s	82Mbit/s-0.5Gbit/s	6Mbit/s,82Mbit/s	24Mbit/s to 221Mbit/s	2Gbit/s
Target	Real-time	Scalable Extension SVC; High profile	Low power; Video size scalable	Hardware design for H.264 codec	Low power; Power aware; Portable devices	Dynamic QualityScalable; Poweraware video applications	Real-time
External memory usage	yes	yes	yes	yes	N/A	N/A	NO
Compression ratio	N/A	N/A	N/A	N/A	N/A	N/A	~2
Extra notes	No entropy coding	UMC90nm CMOS 1P9M	CMOS 65 Technology	UMC 180, 1P6M CMOS	TSMC 180, 1P6M CMOS	CMOS 130	

Table 7-4 presents a list of previous work, while our design stands out due to its substantially reduced hardware area achieved by skipping inter prediction and implementing other modules with an efficient hardware architecture approach. Our design excels in higher frequency and achieves a higher throughput.

CHAPTER EIGHT

CONCLUSION

This thesis presents an H.264/AVC intra frame video encoder hardware architecture which is implemented on FPGA devices using VHDL language. The architecture design includes intra prediction, transform and quantization and CAVLC blocks. The performance of the encoder was evaluated using Vivado simulation and subsequently synthesized for various FPGA devices to measure both its speed and power consumption. Computational complexity of H.264 makes its real-time, parallel hardware implementation challenging. We used different techniques to overcome these challenges as follow:

- We designed a scalable architecture with two different settings, 3 modes intra prediction and 9 modes intra prediction, based on the available hardware on FPGA.
- We process 4 pixels at the same time to increase the speed of the encoding.
- The modified transformer architecture is faster and needs less hardware to be implemented.
- The quantization equation changed in a way to use lower bits to represent MF, QP therefore lower the number of required adders and shifters.
- The dependency between different levels in CAVLC algorithm is eliminated to calculate four levels in parallel. CAVLC output is variable-length, we pack them in 32-bit format to make a fix length output in shortest time possible.

8.1 Suggestions for Future Work

There are some suggested improvements for the future hardware implementation:

1- Adding a simple inter prediction search algorithm to improve the compression ratio because there are more similarities between successive frames than pixels within a frame.

2- 16x16 Luma intra prediction modes can be added to compare 4x4 intra predication and 16x16 and select the prediction with minimum error.

3- Apply a deblocking filter to reduce the blocking artifact.

4- Implement the design in Application-specific integrated circuit (ASIC) to compare the performance with FPGA design.

5- Different mode selection algorithms can be implemented to compare the logic usage and performance.

APPENDICES

A. CAVLC CODING TABLES

Table 7-1 Coeff token mapping to totalCoeff (coeff token) and TrailingOnes (coeff token)

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

Table 7-2 - Continued

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 0	0000 0101 0	1011 01	-
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	-

Table 7-3 - Continued

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 7-4 Codeword table for level_prefix

Level_prefix	**bit string**
0	1
1	01
2	001
3	0001
4	0000 1
5	0000 01
6	0000 001
7	0000 0001
8	0000 0000 1
9	0000 0000 01
10	0000 0000 001
11	0000 0000 0001
12	0000 0000 0000 1
13	0000 0000 0000 01
14	0000 0000 0000 001
15	0000 0000 0000 0001

Table 7-3 Total_zero tables for 4x4 blocks with TotalCoeff (coeff_token) 1 to 7

total_zeros	TotalCoeff (coeff_token)						
	1	2	3	4	5	6	7
0	1	111	0101	0001 1	0101	0000 01	0000 01
1	011	110	111	111	0100	0000 1	0000 1
2	010	101	110	0101	0011	111	101
3	0011	100	101	0100	111	110	100
4	0010	011	0100	110	110	101	011
5	00011	0101	0011	101	101	100	11
6	0001 0	0100	100	100	100	011	010
7	0000 11	0011	011	0011	011	010	0001
8	0000 10	0010	0010	011	0010	0001	001
9	0000 011	0001 1	0001 1	0010	0000 1	001	0000 00
10	0000 010	0001 0	0001 0	0001 0	0001	0000 00	
11	0000 0011	0000 11	0000 01	0000 1	0000 0		
12	0000 0010	0000 10	0000 1	0000 0			
13	0000 0001 1	0000 01	0000 00				
14	0000 0001 0	0000 00					
15	0000 0000 1						

Table 7-5 Total_zero tables for 4x4 blocks with TotalCoeff (coeff_token) 8 to 15

total_zeros	TotalCoeff (coeff_token)							
	8	9	10	11	12	13	14	15
0	0000 01	0000 01	0000 1	0000	0000	000	00	0
1	0001	0000 00	0000 0	0001	0001	001	01	1
2	0000 1	0001	001	001	01	1	1	
3	011	11	11	010	1	01		
4	11	10	10	1	001			
5	10	001	01	011				
6	010	01	0001					
7	001	0000 1						
8	0000 00							

Table 7-6 Tables for run_before

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	001	010	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

REFERENCES

1. *https://www.vcodex.com/h264avc-context-adaptive-variable-length-coding/*.
2. Tran, X.-T. and V.-H. Tran, *An efficient architecture of forward transforms and quantization for H. 264/AVC codecs.* REV Journal on Electronics and Communications, 2011. **1**(2).
3. Loukil, H., et al., *FPGA design of an intra 16× 16 module for H. 264/AVC video encoder.* Circuits and Systems, 2010. **1**(01): p. 18.
4. Messaoudi, K., et al., *Connection of H. 264/AVC hardware IPs using a specific Networks-on-Chip.* Microprocessors and Microsystems, 2015. **39**(8): p. 609-620.
5. Alvarez, M., et al. *HD-VideoBench. A benchmark for evaluating high definition digital video applications*. in *2007 IEEE 10th International Symposium on Workload Characterization*. 2007. IEEE.
6. Ohm, J., *Transmission and storage of multimedia data*, in *Multimedia Signal Coding and Transmission*. 2015, Springer. p. 491-520.
7. Mstafa, R.J. and K.M. Elleithy, *Compressed and raw video steganography techniques: a comprehensive survey and analysis.* Multimedia Tools and Applications, 2017. **76**(20): p. 21749-21786.
8. Oh, D.M., *lNTERNATIONAL TELECOMMUNICATION UNION.* 2010.
9. Chiariglione, L., *Moving picture experts group (mpeg).* Scholarpedia, 2009. **4**(2): p. 6600.
10. Wiegand, T., et al., *Overview of the H. 264/AVC video coding standard.* IEEE Transactions on circuits and systems for video technology, 2003. **13**(7): p. 560-576.
11. Hanzo, L., P. Cherriman, and J. Streit, *Video compression and communications: from basics to H. 261, H. 263, H. 264, MPEG4 for DVB and HSDPA-style adaptive turbo-transceivers*. 2007: John Wiley & Sons.
12. Brandenburg, K. and G. Stoll, *ISO/MPEG-1 audio: A generic standard for coding of high-quality digital audio.* Journal of the Audio Engineering Society, 1994. **42**(10): p. 780-792.
13. Rec, I., *H. 262.* ISO/IEC, 1995: p. 13818-2.
14. Bormann, C., et al., *RTP payload format for the 1998 version of ITU-T Rec. H. 263 video (H. 263+)*. 1998.
15. Pereira, F.C., et al., *The MPEG-4 book*. 2002: Prentice Hall Professional.

16. Puri, A., X. Chen, and A. Luthra, *Video coding using the H. 264/MPEG-4 AVC compression standard.* Signal processing: Image communication, 2004. **19**(9): p. 793-849.

17. *https://en.wikipedia.org/wiki/Advanced_Video_Coding*.

18. *https://www.itu.int/rec/T-REC-H.265*.

19. Han, J., et al., *A technical overview of AV1.* Proceedings of the IEEE, 2021. **109**(9): p. 1435-1462.

20. Laude, T., et al., *A comprehensive video codec comparison.* APSIPA Transactions on Signal and Information Processing, 2019. **8**.

21. Bross, B., et al., *Overview of the versatile video coding (VVC) standard and its applications.* IEEE Transactions on Circuits and Systems for Video Technology, 2021. **31**(10): p. 3736-3764.

22. Ghanbari, M., *Standard codecs: Image compression to advanced video coding*. 2003: Iet.

23. Kuehni, R.G., *Color space and its divisions.* Color Research & Application: Endorsed by Inter-Society Color Council, The Colour Group (Great Britain), Canadian Society for Color, Color Science Association of Japan, Dutch Society for the Study of Color, The Swedish Colour Centre Foundation, Colour Society of Australia, Centre Français de la Couleur, 2001. **26**(3): p. 209-222.

24. Sallam, A.I., O.S. Faragallah, and E.-S.M. El-Rabaie, *Comparative study of video compression techniques.* Menoufia Journal of Electronic Engineering Research, 2018. **27**(1): p. 1-32.

25. Wang, Y., *Analysis application for h. 264 video encoding*. 2010.

26. Tamhankar, A. and K. Rao. *An overview of H. 264/MPEG-4 part 10*. in *Proceedings EC-VIP-MC 2003. 4th EURASIP Conference focused on Video/Image Processing and Multimedia Communications (IEEE Cat. No. 03EX667)*. 2003. IEEE.

27. Park, J.S. and H.J. Song, *Selective intra prediction mode decision for H. 264/AVC encoders.* World Academy of Science, Engineering and Technology, 2006. **13**: p. 51-55.

28. Su, R., G. Liu, and T. Zhang, *Fast mode decision algorithm for intra prediction in H. 264/AVC with integer transform and adaptive threshold.* Signal, Image and Video Processing, 2007. **1**(1): p. 11-27.

29. Kim, C.-s., Q. Li, and C.-C.J. Kuo. *Fast intra-prediction model selection for H. 264 codec*. in *Multimedia Systems and Applications VI*. 2003. SPIE.

30. Kwon, S.-k., A. Tamhankar, and K. Rao, *Overview of H. 264/MPEG-4 part 10.* Journal of Visual Communication and Image Representation, 2006. **17**(2): p. 186-216.

31. Li, Z.-N., M.S. Drew, and J. Liu, *New video coding standards: H. 264 and H. 265*, in *Fundamentals of Multimedia*. 2014, Springer. p. 395-434.

32. Nadeem, M., S. Wong, and G. Kuzmanov. *An efficient realization of forward integer transform in H. 264/AVC intra-frame encoder*. in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. 2010. IEEE.

33. Malvar, H.S., et al., *Low-complexity transform and quantization in H. 264/AVC.* IEEE Transactions on circuits and systems for video technology, 2003. **13**(7): p. 598-603.

34. Silva, T., et al. *FPGA based design of CAVLC and exp-golomb coders for H. 264/AVC baseline entropy coding*. in *2007 3rd Southern Conference on Programmable Logic*. 2007. IEEE.

35. Tsai, C.-Y., T.-C. Chen, and L.-G. Chen. *Low power entropy coding hardware design for H. 264/AVC baseline profile encoder*. in *2006 IEEE International Conference on Multimedia and Expo*. 2006. IEEE.

36. *https://www.itu.int/rec/T-REC-H.264*.

37. Atitallah, A.B., H. Loukil, and N. Masmoudi, *Fpga design for h. 264/avc encoder.* International Journal of Computer Science, Engineering and Applications, 2011. **1**(5): p. 119.

38. Chen, Y.-H., et al. *An H. 264/AVC scalable extension and high profile HDTV 1080p encoder chip*. in *2008 IEEE Symposium on VLSI Circuits*. 2008. IEEE.

39. Iwata, K., et al., *A 256 mw 40 mbps full-hd h. 264 high-profile codec featuring a dual-macroblock pipeline architecture in 65 nm cmos.* IEEE Journal of Solid-State Circuits, 2009. **44**(4): p. 1184-1191.

40. Chen, T.-C., et al., *Analysis and architecture design of an HDTV720p 30 frames/s H. 264/AVC encoder.* IEEE Transactions on Circuits and Systems for Video Technology, 2006. **16**(6): p. 673-688.

41. Chen, Y.-H., et al., *Algorithm and architecture design of power-oriented H. 264/AVC baseline profile encoder for portable devices.* IEEE Transactions on Circuits and Systems for Video Technology, 2009. **19**(8): p. 1118-1128.

42. Chang, H.-C., et al., *A dynamic quality-adjustable H. 264 video encoder for power-aware video applications.* IEEE Transactions on Circuits and Systems for Video Technology, 2009. **19**(12): p. 1739-1754.

43. Pradhan, A.K., L.K. Kanoje, and B.R. Swain, *FPGA based High Performance CAVLC Implementation for H. 264 Video Coding.* International Journal of Computer Applications, 2013. **69**(10).

44. Zhang, Y., G. Jiang, and M. Yu, *Low-complexity quantization for H. 264/AVC.* Journal of Real-Time Image Processing, 2009. **4**(1): p. 3-12.

45. Zhong, H., et al., *A 4-way parallel CAVLC design for H. 264/AVC 4Kx2K 60fps encoder.* IEICE Electronics Express, 2011. **8**(22): p. 1863-1869.