Asic design for MPEG-4 AVC decoder

Zhang, Chaochao

2006

Zhang, C. (2006). Asic design for MPEG-4 AVC decoder. Master' s thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/39136>

<https://doi.org/10.32657/10356/39136>

4435 697

ASIC Design for MPEG-4 AVC Decoder

Zhang Chaochao

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Master of Engineering

2006

Abstract

The new multimedia standard H.264/MPEG-4 AVC achieves higher compression while preserving video quality. It has several key new features including the integer block transform, enhanced motion prediction capability, in-loop deblocking filter and enhanced entropy coding methods.

In this report, algorithms of H.264 decoder and the five functional blocks are summarized and explained. The advantages of H.264 over the old algorithm like MPEG-2 are analyzed. For example, the integer block transform has simpler computation and has no mismatch problem over the traditional transforms.

The project aims to design hardware for some functional blocks of the H.264 baseline profile decoder. The requirements of these functional blocks are set to real time processing for HDTV application of video format 1280x720 at 60 Hz. The application specific integrated circuit (ASIC) designs for five functional blocks are described. The five functional blocks are Inverse Quantization and Transform block (Inverse Q & T), Reorder block, Context Adaptive Variable Length Coding block (CAVLC), Intra Prediction block and Deblocking Filter block. The hardware has been synthesized using the AMS (AustriaMicroSystems) 0.35 μm at 3.3 Volt technology library.

For the Inverse Quantization and Transform functional block, an integrated architecture for the quantization and transform has been designed. My design,

compared to [17], has the chroma DC transform integrated into the transform architecture. The gate level synthesis results show that a hardware saving of 36% can be achieved for this integrated architecture over one using separate components.

In the Context Adaptive Variable Length Coding block, the algorithm for the level computation component is re-organized to reduce the operation bit width. As a result, the critical path delay for level computation component is reduced from 5.25 ns in [24] to 4.51 ns.

For the deblocking filter block, the block edges filtering sequence has been modified from the algorithm in the standard document [2]. However the actual filtering outputs will not change. In this way, a large amount of memories can be saved. The simulation results show a hardware saving of 66% over the architecture in [18] if the on-chip memory is modeled by registers for area comparison.

Acknowledgement

I would like to take this opportunity to sincerely thank my project supervisor Associate Professor Ho Duan Juat, for her support and patience in this project. Particularly, she helped me to set the project direction. In this project, I have learnt the necessary technical skills to do digital IC design. I have also gained a good understanding of what research should be about. Besides, in these two years, I have learnt the importance of determination and consistency to accomplish a task.

I also like to thank Associate Professor Wong Moon Chung, Eddie, for helping me deal with some issues on my project.

Finally, I would like to thank all the technicians in the Centre of Integrated Circuits & Systems Lab for helping me install the software. Especially Mr. D. R. Neubronner, in the Integrated Systems Research Lab helped me troubleshoot the Synopsys software tools.

Abbreviations

ASIC	– Application Specific Integrated Circuit
AVC	– Advanced video coding
MPEG	– Moving Picture Experts Group
HDTV	– High Definition Television
SDTV	– Standard Definition Television
VCEG	– Video Coding Experts Group
JVT	– Joint Video Team of MPEG and VCEG
CAVLC	– Context Adaptive Variable Length Coding
CABAC	– Context-based Adaptive Binary Arithmetic Coding
MB	– Macroblock, 16x16 pixels
VHDL	– Very High Speed Integrated Circuits Hardware Description Language
CAD	– Computer Aided Design
Luma	– Short form for Luminance
Chroma	– Short form for Chrominance
HVS	– Human Visual System
MAD	– Mean Absolute Difference
MSE	– Mean Square Error
ME	– Motion Estimation
MC	– Motion Compensation
CIF	– Common Intermediate Format
DCT	– Discrete Cosine Transform
ISO	– International Standards Organization
IEC	– International Electrotechnical Commission
ITU-T	– International Telecommunication Union Telecommunication Standardization Sector
ISDN	– Integrated Services Digital Network
Q and T	– Quantization and Transform
DC coefficients	– DC coefficient in this report means the top left coefficient of a 4x4 matrix after transform. The physical meaning is the average pixel value of the 4x4 pixel matrix before integer transform. So it is named DC coefficients.
AC coefficients	– All the other elements except the DC coefficient are AC coefficients.
NAL	– Network Abstraction Layer
VLC	– Variable Length Coding
LIFO	– Last In First Out

Content Table

Chapter 1 Introduction1

 1.1 Motivation1

 1.2 Objectives2

 1.3 Major contributions of the thesis.....3

 1.4 Design methodology3

 1.5 Report organization5

Chapter 2 Background.....6

 2.1 Video concepts.....6

 2.2 Video coding theory.....8

 2.3 MPEG standard overview.....12

Chapter 3 The H.264 algorithm15

 3.1 Overview of the H.264 algorithm.....15

 3.2 Transform and quantization.....18

 3.2.1 Introduction.....18

 3.2.2 4x4 integer transform.....20

 3.2.3 Quantization for luma AC transformed coefficients.22

 3.2.4 Quantization for chroma AC transformed coefficients.....25

 3.2.5 Hadamard transform and quantization for luma DC coefficients.....26

 3.2.6 Chroma DC transform and quantization.....27

 3.3 Reorder28

 3.4 Entropy coding.....29

 3.4.1 Introduction.....29

 3.4.2 CAVLC decoding process.....30

 3.5 Intra prediction35

 3.5.1 Luma 4x4 intra prediction.....35

 3.5.2 Luma 16x16 intra prediction40

 3.5.3 Chroma 8x8 intra prediction.....42

 3.6 Deblocking filter.....43

 3.6.1 Introduction.....43

 3.6.2 Filter process for edge element with bS<4.....46

 3.6.3 Filter process for edge element with bS=4.....47

Chapter 4 Overall hardware architecture48

 4.1 Overall requirement and architecture design48

 4.2 Design requirements for the functional blocks.....50

 4.3 Literature review of some functional blocks.....51

Chapter 5 Designs of the five functional blocks54

5.1 Inverse quantization and transform54

 5.1.1 Inverse Hadamard transform 55

 5.1.2 Inverse 4x4 integer transform.....57

 5.1.3 Inverse luma AC quantization.....58

 5.1.4 Chroma DC transform58

 5.1.5 Inverse luma (and chroma) DC quantization.....59

 5.1.6 Functional simulation for inverse Q and T block59

5.2 Reorder62

 5.2.1 Design for level and run_before LIFO registers.....63

 5.2.2 Reorder state diagram and operations.....64

 5.2.3 Functional simulation for reorder block.....68

5.3 Context Adaptive Variable Length Coding.....69

 5.3.1 Introduction.....69

 5.3.2 Architecture of CAVLC.....69

 5.3.3 CAVLC block state diagram72

 5.3.4 Functional simulation for CAVLC block.....79

5.4 Intra prediction80

 5.4.1 Luma 4x4 intra prediction.....80

 5.4.2 Luma 16x16 and chroma 8x8 intra prediction87

 5.4.3 Simplified state diagram for intra prediction.....94

 5.4.4 Functional simulation for intra prediction block96

5.5 Deblocking filter98

 5.5.1 Architecture of the filter block.....98

 5.5.2 Filter block state diagram.....102

 5.5.3 Functional simulation for filter block.....102

Chapter 6 Synthesis results.....106

 6.1 Inverse quantization and transform108

 6.2 Reorder111

 6.3 Context Adaptive Variable Length Coding.....112

 6.4 Intra prediction114

 6.5 Deblocking filter115

Chapter 7 Conclusions and recommendations117

 7.1 Conclusions.....117

 7.2 Recommendations120

Bibliography.....121

List of figures

Figure 1. Design flow diagram.....	4
Figure 2. Spatial and temporal sampling.....	6
Figure 3. 4:2:0 spatial sampling format.	7
Figure 4. Video compression encoder	8
Figure 5. A common picture.....	9
Figure 6. Frame (a) is previous frame and (b) is current frame.	10
Figure 7. 4x4 DCT transform (a) Spatial domain (b) frequency domain	11
Figure 8. H.264 encoder architecture	15
Figure 9. H.264 decoder architecture	15
Figure 10. Standard syntax hierarchy.....	16
Figure 11. 16x16 MB intra prediction residual data processing.	18
Figure 12. Inter prediction or luma non-16x16 intra prediction residual data flow chart.....	19
Figure 13. Luma 16x16 intra prediction residual data flow chart.....	19
Figure 14. Chroma residual data flow chart.....	20
Figure 15. zig-zag scanning pattern for frame mode.	29
Figure 16. run_before illustration.....	34
Figure 17. luma4x4BlkIdx of a macroblock and co-ordinate system..	35
Figure 18. Neighboring pixel locations for a 4x4 luma block.....	35
Figure 19. The luma 4x4 intra prediction directions..	36
Figure 20. Neighboring pixels and the predicted data for luma 16x16.....	41
Figure 21. Deblocking filtering for a macroblock	43
Figure 22. Pixel labeling across the edge.	45
Figure 23. Hardware architecture of H.264 decoder..	48
Figure 24. Macroblock level pipeline processing.....	49
Figure 25. Macroblock level pipeline for H.264 decoder.....	50
Figure 26. 8x8 DCT architecture [16].....	51
Figure 27. Architecture of CAVLC [24]	52
Figure 28. Architecture of deblocking filter [18].....	53
Figure 29. Inverse quantization and transform architecture.....	55
Figure 30. Hadamard transform architecture..	55

Figure 31. Transposed register.....	57
Figure 32. Modified Integer transform or Hadamard transform.....	57
Figure 33. Inverse AC quantization.	58
Figure 34. Functional simulation for inverse Q and T block.....	60
Figure 35. Hardware architecture of reorder.	62
Figure 36. Level block circuit	63
Figure 37. Reorder state diagram.	65
Figure 38. Functional simulation for reorder block.....	68
Figure 39. Architecture of CAVLC.....	70
Figure 40. State diagram for CAVLC architecture.....	73
Figure 41. Functional simulation for CAVLC block.....	79
Figure 42. Registers for neighboring pixels.....	80
Figure 43. Computational element 0.....	81
Figure 44. Computational element 1.....	81
Figure 45. Computational element 2.....	82
Figure 46. Computational element 3.....	82
Figure 47. Circuit for luma 4x4 DC prediction output.....	84
Figure 48. State diagram for luma 4x4 intra prediction.....	86
Figure 49. Registers to hold the input neighboring pixels.....	87
Figure 50. The circuit for calculating H and V.....	89
Figure 51. Hardware to get the m41, (b or c at different clocks).....	90
Figure 52. Hardware for the predicted output.....	92
Figure 53. The circuit for chroma 8x8 DC and luma 16x16 DC prediction.....	93
Figure 54. State diagram of the intra prediction.....	95
Figure 55. Timing diagram of luma 16x16 Plane prediction.....	96
Figure 56. Functional simulation for intra prediction block.....	96
Figure 57. Proposed architecture of deblocking filter.....	99
Figure 58. Filtering for luma component.....	100
Figure 59. State diagram of filtering the luma data.....	103
Figure 60. State diagram of filtering the chroma data.....	104
Figure 61. Functional simulation for filter block.....	105

List of tables

Table 1. Quantization step.....23

Table 2. Values of MF_{ij} for $QP=0\sim5$24

Table 3. Values of MR_{ij} for $QP=0\sim5$25

Table 4. Level codeword table of $vlcnum=0$ 32

Table 5. Level codeword table of $vlcnum=4$ 32

Table 6. 9 luma 4x4 intra prediction modes.....36

Table 7. Four prediction modes for Intra luma 16x16 intra prediction.....40

Table 8. Operations for $diag_down_right$ prediction mode.....85

Table 9. Comparison of the transform architecture.108

Table 10. Separate components without integration for quantization and transform109

Table 11. Integrated blocks and integrated circuit for quantization and transform109

Table 12. Reorder synthesis results.....111

Table 13. CAVLC synthesis results.....112

Table 14. Intra prediction synthesis results.....114

Table 15. Deblocking filter synthesis results.....115

Table 16. Features comparison of inverse Q and T block.....117

Table 17. Features comparison of level computation component of CAVLC block118

Table 18. Features comparison of deblocking filter block118

Table 19. Summarized synthesis results.....118

Mathematical functions

The following few functions are defined and used in this report.

$$\text{Abs}(x) = \begin{cases} x & ; \ x \geq 0 \\ -x & ; \ x < 0 \end{cases}$$

$$\text{Sign}(x) = \begin{cases} 1 & ; \ x \geq 0 \\ -1 & ; \ x < 0 \end{cases}$$

Floor(x) is the greatest integer less than or equal to x.

$$\text{Round}(x) = \text{Sign}(x) * \text{Floor}(\text{Abs}(x) + 0.5)$$

$$\text{Clip1}(x) = \text{Clip3}(0, 255, x)$$

$$\text{Clip3}(x, y, z) = \begin{cases} x & ; \ z < x \\ y & ; \ z > y \\ z & ; \ \text{otherwise} \end{cases} \quad \text{Clip3}(x, y, z) \text{ is defined when } y > x.$$

Chapter 1 Introduction

1.1 Motivation

In early 1998, the Video Coding Experts Group (VCEG) issued a call for proposals on a project called H.26L, which was aimed at providing a higher coding efficiency for a broad variety of applications. The first draft for that new standard was adopted in October 1999. In December 2001, the Moving Picture Experts Group (MPEG) joined with VCEG to form a Joint Video Team (JVT) [1]. The new standard, now known as MPEG-4 Part 10, “Advanced Video Coding”, is also known as H.264 [2]. Its main goal is to enhance the compression performance and provide a “network-friendly” video representation.

The H.264 standard is intended to cover a large range of applications including [3]:

- Cable TV on optical networks
- Direct broadcast satellite video services
- Interactive storage media
- Multimedia messaging service
- Real-time conversational services

The algorithm of H.264 has some new features like motion estimation with variable block sizes, enhanced motion-prediction capability, integer transform, in-loop deblocking filter and enhanced entropy-coding methods. Due to all these new features, H.264 provides a compression gain factor of 1.5-2.0 over the previous standards such as H.263+ and MPEG-4 Part 2 [1]. Compared with MPEG-4, H.263, and MPEG-2, H.264/AVC can achieve 39%, 49%, and 64% of bit rate reduction, respectively [4].

However, the high compression efficiency is achieved at the cost of algorithm complexity. Functions such as motion estimation, integer transform, arithmetic coding and in-loop deblocking filter can be highly computationally intensive [5]. For example, even after a tremendous effort in speed optimization of the filtering algorithms, the deblocking filter can easily account for one-third of the computational complexity of a decoder [6].

The required computation load for the H.264 standard is more than four times higher than the old standards [7]. The complexity of H.264 is too high to be applied in those consumer electronic industries, especially for those with low computational capability [8]. Therefore, hardware acceleration is a must for real-time applications, especially for Context-Adaptive Variable Length Coding (CAVLC) and in-loop deblocking filter, which are very computationally intensive parts.

This project was set up to explore the architecture of some functional blocks in the H.264 decoder based on the application specific integrated circuit (ASIC) technology.

1.2 Objectives

The standard H.264 includes three profiles: baseline, main and extended profile. Each profile includes a subset of the coding features in the standard. The complexity of the decoder could be reduced by designing it specifically for one profile of certain capabilities [9].

The objective of this project is to implement the ASIC hardware architecture for some functional blocks of the H.264 baseline profile decoder. The timing requirements of the functional blocks are mainly set to decode HDTV, with video format 1280x720 at 60 Hz. The important consideration in the design for the functional blocks is to ensure the real time decoding of the HDTV video. Beside that, the H.264 algorithms are exploited to achieve hardware savings.

The baseline profile supports:

- 4:2:0 video format (explained later)
- I-slice and P-slice (no B-slice)
- Frame macroblock and frame pictures (No field picture)
- Entropy-coding of CAVLC (No CABAC (Context-Based Adaptive Arithmetic Coding))

The functional blocks that have been designed include the CAVLC (Context Adaptive Variable Length Coding), Reorder, Inverse Quantization and Transform, Intra

Prediction and In-loop Deblocking Filter. However, there is one functional block, the inter prediction that has not been designed in this project.

1.3 Main contributions of the thesis

Five specific components of the H.264 decoder for baseline profile have been designed. They are the Inverse Quantization and Transform, Reorder, CAVLC, Intra Prediction and Deblocking Filter blocks.

First, the algorithms of the five functional blocks are summarized and explained clearly in the report. The H.264 standard document is very difficult to read. It is necessary to gain some general background of video compression before reading H.264 standard document directly. It takes a long time to understand the algorithm in detail.

Second, the algorithms for some functional blocks are analyzed carefully and the efficient hardware architectures are generated respectively.

An integrated architecture for the Inverse Quantization and Transform block has been proposed. In this architecture, the chroma DC transform is integrated with the transform part nicely. The synthesis results show that the integrated architecture for Inverse Quantization and Transform gains a hardware saving of 36% over the architecture using separate components.

For the Deblocking Filter, a new method of filtering sequence is suggested. The gain in the hardware saving is about 67% over the architecture in [18] if the on-chip memory is replaced by registers for comparison.

For CAVLC, the level computation component is improved by reducing its delay from 5.25 ns in [24] to 4.51 ns. This is important because the level computation component reduces the critical path delay of CAVLC block.

1.4 Design methodology

Figure 1 shows a typical design flow for digital circuits. The steps performed in this project are also indicated in Figure 1. The specific CAD tools like ModelSim and Design Compiler are employed in this project.

In digital system designs, the first step is to translate the algorithms into its equivalent digital hardware circuit. In this project, the digital circuit is described using VHDL code at the Register Transfer Level (RTL) abstraction. VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language (HDL). It is a high level description language to model hardware.

Figure 1. Design flow diagram

After VHDL coding, functional simulation is carried out to check whether the hardware description is correct. That means whether the circuit is able to get the desired output from a set of inputs. Of course, it also checks for syntax errors. In this step, the simulator assumes that the hardware has unit delay. In this project, ModelSim SE Version 5.5e is employed to do functional simulation.

When functional simulation is passed, logic synthesis is performed to do timing simulation at gate level. UNIX system based Synopsys Design Compiler is used in this project to read in the VHDL code to do logic synthesis. During logic synthesis, the design is mapped to the given technology library and optimized to meet the design constraints. The necessary timing constraints are set in order to meet the timing requirements of the circuit. In this design, the target library is AMS (AustriaMicroSystems) 0.35 μm at 3.3 Volt.

After the gate level timing simulation is passed, the netlist is generated for doing Place and Route. These steps after logic synthesis in Figure 1 have not been carried out in the project. So no further discussion is provided here. They are shown in Figure 1 for integrity of a design flow.

1.5 Report organization

This report is organized in the following way. After the introduction, Chapter 2 gives a background of the H.264 algorithm, which introduces the basics of video compression, video coding theory and the MPEG standard overview. Then in Chapter 3, the detailed algorithms for the five functional blocks of the H.264 are discussed. This is followed by Chapter 4, where the design requirements, architectural design and a literature review of the functional blocks of the H.264 decoder are presented. In Chapter 5, the hardware designs of the five functional blocks are presented. After that, the functional simulations of the five functional blocks are presented. Chapter 6 provides the synthesis results. Finally, in Chapter 7 the conclusions and recommendations are provided.

Chapter 2 Background

2.1 Video concepts

Video compression is very important in transmission and storage because the huge amount of raw data involved usually greatly exceeds the capability of the transmission and storage hardware. This section introduces some issues and basic concepts related to video compression.

A natural visual scene is spatially and temporally continuous. In order to represent it in digital form, the real scene is sampled spatially and temporally as shown in Figure 2.

Spatial sampling means that an image frame is represented using a rectangular grid of samples (or pixels). A parameter used to measure the density of the pixels in an image is called spatial resolution. If the spatial sampling rate or spatial resolution is low, the video frame itself will appear blurred.

Temporal sampling means that a video clip of certain time unit is represented by certain number of static images. For example, a common temporal sampling rate is 25 frames/second. If the temporal sampling rate is low, the video clip will not appear continuously between two adjacent frames.

Figure 2. Spatial and temporal sampling

In the spatial sampling, the video capture is performed by sampling the image horizontally line by line. Frames are formed by progressive sampling, which means in the spatial domain, it samples every line of pixels from top to bottom progressively. However, fields are formed by interlaced sampling, which means in the spatial domain,

it samples either the odd-numbered or even-numbered lines every time. So the temporal sampling rate of the fields should be double of the sampling rate of frames, while the actual information conveyed is the same. However, interlaced sampling method gives a smoother motion visual effect.

For every pixel in the spatial domain, it can be represented by RGB (red, green, blue) color space. That means every pixel is denoted by three components. However, in the actual video processing of H.264, another color space YCbCr or YUV format is employed. Y is the luminance (represented by luma) component and Cb, Cr are the color (chroma) components. The human visual system (HVS) is less sensitive to color components (Cb and Cr) than to brightness (Y). In other words, our eyes are more sensitive to the black and white components than to the color components. So in video processing, higher resolution is reserved for the luma component than the chroma components. This is why, in H.264, the YCbCr sampling format 4:2:0 is used as shown in Figure 3. It is noticed that, the spatial sampling rate for Cb or Cr component is only $\frac{1}{4}$ that for Y component. In this way, extra processing on Cb and Cr is reduced. In RGB format, this property cannot be utilized.

Figure 3. 4:2:0 spatial sampling format

In video processing, generally the video format defines the spatial sampling resolution. For example, the video format Common Intermediate Format (CIF) is of 352 x 288 luma resolution in horizontal direction and vertical direction respectively.

With the above introduction to video concepts, a network modem is taken as an example to illustrate the bit rate required to transmit the video signal. It can operate at

56,600 bits per second. Assume that a CIF video with the spatial resolution 352 by 288 is to be displayed at 30 frames per second. If each of the three primary colors RGB (red, green, blue) is represented by 1 pixel with 8 bits, then in order to transmit this raw data without compression continuously, the transmission line requires a bit rate of $288 \times 352 \times 8 \times 3 \times 30 = 72,990,720$ bps [10]. Without data compression, it is impossible for the modem to handle the raw video in real time. Video compression is also very critical for storage. A Digital Versatile Disk (DVD) can only store a few seconds of raw video at television-quality resolution and frame rate [5].

2.2 Video coding theory

Figure 4. Video compression encoder

The common functional blocks for a typical video compression encoder are shown in Figure 4. The decoding will be just the reverse process. The basic processing unit is a macroblock (MB). A macroblock is a block of 16x16 pixels. It may be divided into even smaller units for different functional blocks. In the following paragraphs of this section, the functional blocks will be addressed briefly.

The basic idea for the video compression is to eliminate the redundancies in the video itself and to exploit the HVS.

Intra prediction

As for the redundancies in the video itself, it refers to both spatial and temporal redundancies. A video contains a series of picture frames. In a picture frame itself, the contents of the adjacent areas are generally similar. The area with arrows in Figure 5 gives an example. So if the pixel data of a block of the picture is known, it is not needed to encode the second block that is adjacent. Instead, it may encode the

difference between them. In the extreme case, if the second block is exactly the same as the first one, the difference would be zero, and it requires very few bits to code the second block. This is called spatial redundancy and is exploited by a process called intra prediction. However, intra prediction is usually more than simply finding out the differences between the two adjacent blocks. Detailed H.264 intra prediction algorithm can be found in section 3.5.

Figure 5. A common picture

Inter prediction

The temporal redundancy refers to the similarities and correlations between two nearby picture frames. It could be eliminated by inter prediction. For a typical video stream, there are about 25 picture frames to be displayed in one second. In fact, the two temporally adjacent frames usually do not differ too much in their content. Figure 6 shows two temporally adjacent frames. The two indicated blocks appear to be very similar.

So when encoding a block of a frame and when the encoder uses inter prediction method, it will search similar blocks in the recent past and future frames. Upon certain matching criterion, it will choose a matched block from the selected reference frame, whose pixel values will generally be similar to the current encoding blocks. The process of searching a matched block is called motion estimation (ME). The location of the matched block with respect to the encoding block is stored in a parameter called motion vector. There are several types of matching criterion, like the mean square

error (MSE) and mean absolute difference (MAD) [10]. For example, a matched block may be found if MSE or MAD between the current encoding block and the block in a reference frame is smaller than a threshold value. The threshold value depends on the actual algorithm.

After ME, it will get the motion vectors and the selected reference frame, and perform modifications like interpolation on the matched block in the reference frame. This is called motion compensation (MC). Then the motion compensated block is subtracted from the current encoding blocks to get the residuals. As shown in Figure 6, the frame (b) is very similar to the frame (a) except for the area near the eyes. When encoding the indicated block in the frame (b), ME process will search and find the corresponding block in the frame (a). These two blocks, by a glance, will be very similar and after MC, the difference should be close to zero.

Figure 6. Frame (a) is previous frame and (b) is current frame

Discrete Cosine Transform

HVS perceives the outside world in a rather complicated way. In HVS, visual information is not perceived equally; some information may be more important than others. As mentioned earlier, in video format YCbCr, luma information is more sensitive for HVS than chroma information.

Besides, HVS also helps the compression during transform. The Discrete Cosine Transform (DCT) converts a block of area (a matrix of pixels, like the size of 8x8) from the spatial domain to the frequency domain. In the spatial domain, samples are highly spatially correlated, which means the neighboring samples have high

dependency. The aim of the transform is to reduce correlation of data, ideally leaving a small number of big transformed coefficients located in the low frequency area and a large number of small transformed coefficients located in the non-low frequency areas. In Figure 7, (a) shows the spatial domain 4x4 matrix; (b) shows the frequency domain 4x4 matrix after DCT transform. In the frequency domain, the top left element (0,0) is called the DC coefficient. All the other elements are called AC coefficients. HVS is much more sensitive to low frequency components than high frequency components. That means in Figure 7 (b), the elements located near (0,0) are more important than those located near (3,3). So in the following functional block quantization, the DCT transformed coefficients near (0,0), which are of low frequency will be treated more carefully.

Figure 7. 4x4 DCT transform

Quantization

Quantization is applied to the transformed coefficients. This is similar to the quantization in IC design, to convert the analog signal to digital signal. In this way, the data range is reduced and fewer bits are required to represent the specific data value. Compression is achieved in this step. Meanwhile, losses are introduced as well since the inverse quantization can only get back the approximate transformed coefficients.

Big quantizers give a higher ratio of compression while causing bigger distortion. Since the HVS is more sensitive to the low frequency components of the transformed coefficients, smaller quantizers for the low frequency coefficients and bigger quantizers for the high frequency coefficients are assigned. This results in many zeros

in the quantized coefficients, especially near location (3,3) in Figure 7. This way of quantization produces better compression while keeping a good visual effect than having a constant quantizer for all the transformed coefficients.

Reorder

After quantization, the reorder process produces a one-dimensional quantized coefficients array with certain zig-zag scan order. The scan sequence is designed in a way such that the resulting array typically contains one or more clusters of non-zero coefficients near the start, followed by strings of zero coefficients. In this manner, the array could be group into (run_before, level) pairs easily. The run_before stands for the number of zeros preceding a non-zero coefficient and level is the non-zero coefficient [5]. This type of grouping is mainly used for entropy coding.

Entropy coding

Entropy coding is a statistical coding algorithm that will save more bits. It basically converts video information into a compressed bitstream. The video information includes quantized coefficients (run_before-level pairs), motion vectors and sequence header information. The detailed algorithm will be explained in H.264 entropy coding in section 3.4.

Filter

Beside all the functional blocks described above, there is another functional block called the filter. This is not in Figure 4. In the decoder, the filter mainly serves to produce better visual effects before display. For example, quantization leads to block edge discontinuities and distortions in the pictures. The filter block eliminates this kind of problems effectively and produces better visual effects.

2.3 MPEG standard overview

In response to the needs of the emerging markets for video compression, several national and worldwide standards have been implemented and developed. These include the Moving Picture Experts Group and the Video Coding Experts Group. The first one is a working group of the ISO (International Standards Organization) and the

International Electrotechnical Commission (IEC). The second one (VCEG) is a study group of ITU-T (International Telecommunication Union Telecommunication Standardisation Sector).

The Moving Picture Experts Group is responsible for MPEG-1 and MPEG-4 standards, while Video Coding Experts Group is responsible for H.261 and H.263 video teleconferencing coding standards. Standard MPEG-2 (or H.262) is a joint work between ITU and ISO. The following paragraphs briefly summarizes the history of these standards.

MPEG-1: MPEG-1 was completed in 1991. It is also known as ISO/IEC specification 11172, "Coding of Moving Picture and Associated audio". The target application is digital storage media at bit rates up to about 1.5 Mbps. It was developed for multimedia CD-ROM applications. It supports a wide range of application profiles and the user can specify a set of input parameters including flexible picture size and frame rate. Important features provided by MPEG-1 include frame-based random access of video [10].

MPEG-2: It was started in 1990 and completed in 1994. It is an extension of MPEG-1 that allows for greater input-format flexibility, higher data rate for SDTV or HDTV application, and better error resilience. The alternative name is ISO/IEC specification 13818, "Generic Coding of Moving Pictures and Associated Audio". The MPEG-2 was mainly targeted for storage and broadcasting of television quality video and audio applications at bit rates between 2 and 30 Mbps. Besides, it is backward compatible to the MPEG-1 standard. It had a great impact on the consumer electronics industry since the DVD and DTV have adopted it as a core technology.

MPEG-4: The standard was started in July 1993. The International Standard (IS) of MPEG-4 version 1 was completed in February of 1999, by name of ISO/IEC Standard 14496 Part 2 (MPEG-4 Visual). The goal of the MPEG-4 standard is to provide the core technology that allows efficient content-based storage, transmission, and manipulation of video, graphics, audio, and other data within a multimedia environment.

In the above explanation, the standard *MPEG-3* is missing. It was initially intended for high-definition video, however this was incorporated into the scope of *MPEG-2*. And by this time, *MPEG-4* had been already under development.

MPEG-7 and **MPEG-21** were concerned with multimedia content representation and a generic multimedia framework respectively [5]. *MPEG-7* is concerned with access to multimedia data rather than the mechanisms for coding and compression. *MPEG-21* defines key elements of this delivery framework, including content description and identification, content handling, intellectual property management, terminal and network interoperability and content representation [11].

H.261: The H.261 video coding standard was approved during 1988 to 1993 by ITU-T. It was designed for video teleconferencing over constant bit rate channels, and primarily aimed at ISDN channels of bit rate $P \times 64$ kbps ($1 \leq P \leq 30$).

H.263: The technical content was completed in late 1995 and the standard was approved in early 1996. It is specifically designed for very low bit rate applications such as practical video telecommunication, supporting a range of channels from low bit rates (20-30 kbps) to high bit rates (several Mbps).

H.263+ (1998), H.263++(2001): Extensions to H.263 to support a wider range of transmission scenarios and to improve the compression performance.

H.264: From 1998, VCEG was working on the new standard H.26L (L stands for long term). With the development of hardware processor and video coding researches, it was clear there would be a step change in video coding performance. So after evaluating several technologies in 2001, H.26L test model CODEC was the best choice to meet the MPEG's requirements. Then the two groups VCEG and MPEG set up the collaborative Joint Video Team (JVT) to finalize the H.26L proposal and convert it into an international standard, entitled "Advanced Video Coding" (AVC). It was published as ITU-T H.264 and ISO/IEC MPEG-4 Part 10 in 2003 [5].

Chapter 3 The H.264 algorithm

In Chapter 2, the general video compression codec is discussed. In this chapter, some obvious advantages of H.264 algorithm are described briefly. At the same time, the detailed algorithms of the five specific functional blocks in decoding process are presented. Some of them are also explained in encoding process in order to be easily understood.

3.1 Overview of the H.264 Algorithm

Figure 8. H.264 encoder architecture [5]

Figure 9. H.264 decoder architecture [5]

The H.264 standard does not explicitly define an encoder or decoder. Rather, the standard defines the syntax of an encoded video bitstream together with the method to

decode this bitstream. However, a compliant decoder or encoder is likely to include the functional blocks as shown in Figure 8 and 9.

In the encoder, for inter prediction, motion estimation (ME) is performed to select the matched block. Then motion compensation (MC) is performed to get the residual data. For intra prediction, nine 4x4 and four 16x16 intra prediction modes are evaluated and one of them is selected. Then the intra predicted residual data will be ready. After comparing the inter and intra prediction results, one of them P will be selected according to certain criterion. These residual coefficients R will go through transform and quantization to obtain X. The transform in H.264 uses 4x4 integer transform instead of DCT. The quantized coefficients, X will be the input to the reorder and Entropy encode blocks as explained in Chapter 2.

Entropy coding encodes all the syntax elements, including parameter information and data coefficients using uniquely identified binary codes. Figure 10 shows the standard syntax or an encoded bitstream hierarchy.

Figure 10. Standard syntax hierarchy

For the encoder, besides the above data path mentioned, there is also a feed back loop. As shown, the feed back loop goes through the inverse quantization and transform. The reconstructed pictures serve as the reference pictures to do the inter and intra predictions. The reason why H.264 uses reconstructed pictures instead of the existing input pictures is that in the decoding process, the reference pictures for inter and intra prediction are the reconstructed pictures. This is to keep the encoding and decoding processes in step with the same set of reference pictures. Without this feed back loop, the reference pictures in the encoding process are the input pictures, while the reference pictures in the decoding process are the reconstructed pictures. The compression loss will cause the two types of reference pictures to be different. This will cause drift errors in the decoded pictures.

Besides, in the encoding process, if it is inter prediction, there will be deblocking filter process. This is necessary as explained later in this page.

The decoding process is almost the inversed process of the encoding process. However, it does not have the ME and the intra mode selection, because during decoding, these information can be decoded from the input bitstream.

The H.264 algorithm has the following improved features:

Inter prediction: It is more flexible in the selection of inter prediction block sizes and shapes in H.264 standard than any previous standards, with a minimum luma block size as small as 4x4. It allows multiple reference frames for prediction. It also supports fractional pel inter-polation.

4x4 integer transform: The previous standards rely on the 8x8 DCT that provides the transform but produces inverse transform mismatch problems. H.264 uses 4x4 integer transform, which is multiplier free and involves addition, shift operations in 16-bit arithmetic only [7].

Enhanced entropy coding: Both CAVLC and CABAC use context-based adaptivity to improve performance relative to prior standard designs. Besides the context-based adaptivity, CABAC also employs arithmetic coding to provide even better coding efficiency [12].

In-loop deblocking filter: Block based video compression produces blocking artifacts. Two sources of blocking artifacts: first, the source is due to the coarse quantization of the transform coefficients which can cause visually disturbing discontinuities at the block boundaries. The second source is the discontinuities on the edges of the inter predicted blocks. In the previous standards, the deblocking filter is applied as a post-filtering operation before the video frame is displayed. However, this technique is not optimal because the reference frame used for the inter prediction has not been filtered. The blocking artifacts in the reference frame result in a less accurate inter prediction, thus coding efficiency is reduced. In H.264, the deblocking is performed inside the coding loop, so all the reference frames are filtered [13].

3.2 Transform and quantization

3.2.1 Introduction

Luma component

In the encoder, the input residual data before transform and quantization block can be either the:

- (1) 4x4 inter prediction residual or luma non-16x16 MB intra prediction residual
- (2) luma 16x16 MB intra prediction residual data

If it is case (1), it means that the 4x4 residual data is either inter prediction or 4x4 intra prediction residual data. So the residual data will be 4x4 forward integer transformed (section 3.2.2) and then forward quantized (section 3.2.3).

If it is case (2), the luma 16x16 MB intra prediction residual data has 16 4x4 residual blocks as shown in Figure 11. They will all be 4x4 forward integer transformed (section 3.2.2) to produce 16 4x4 transformed blocks, which is in the frequency domain. Then the DC coefficients (the top left coefficient of a 4x4 matrix after transform) are then picked out from each 4x4 transformed block to form a 4x4 DC block. The DC coefficients are generally big. In Figure 11, the block labeled as -1 is the 4x4 DC block.

Figure 11. 16x16 MB intra prediction residual data processing

For this 4x4 DC block, Hadamard transform will be applied followed by its quantization (section 3.2.5). For the rest of the AC blocks (those transformed blocks without the DC coefficients) labeled from 0 to 15, forward quantization (section 3.2.3) will be applied.

Chroma components

For the chroma components, the intra prediction is similar to the luma 16x16 MB intra prediction. Regardless of prediction modes, for residual data, the 4x4 forward integer transform (section 3.2.2) is carried out first. Then the DC coefficients are picked off as shown in Figure 11 to form the two 2x2 DC blocks labeled as 16 and 17. Then chroma DC transform and quantization (section 3.2.6) are performed on these two DC blocks. For the AC blocks labeled from 18 to 25, chroma quantization (section 3.2.4) is performed.

Flow chart for the transform and quantization

Figure 12 and Figure 13 show the two data flow charts of transform and quantization for luma component and Figure 14 shows only one data flow chart for chroma components. Those blocks before the “Reorder, entropy coding and NAL” are in the encoder and those blocks after “Reorder, entropy coding and NAL” are in the decoder. NAL stands for Network Abstraction Layer, which is for transmission and storage.

1) Inter prediction or luma non-16x16 intra prediction residual data

Figure 12. Inter prediction or luma non-16x16 intra prediction residual data flow chart

2) Luma 16x16 intra prediction residual

Figure 13. Luma 16x16 intra prediction residual data flow chart

3) Chroma prediction residual

Figure 14. Chroma residual data flow chart

Accordingly, the decoded 4x4 residual data block for a macroblock will be generated with the sequence from index 0 to 15 for the luma component, then 18 to 25 for the chroma component as labeled in Figure 11.

3.2.2 4x4 integer transform

In this section, the 4x4 block integer transform is analyzed. The 4x4 integer transform is actually derived from the 4x4 DCT (Discrete Cosine Transform).

Forward 4x4 integer transform

The forward 4x4 DCT of an input array X is given by:

$$Y = A * X * A^T = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix} \begin{bmatrix} X \end{bmatrix} \begin{bmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{bmatrix} \quad \text{where } A = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix}$$

$$a = \frac{1}{2} \quad b = \sqrt{\frac{1}{2}} \cos(\pi/8) \quad c = \sqrt{\frac{1}{2}} \cos(3\pi/8) \quad \text{-- Equation 1}$$

Matrix X is the input data. Matrix A is the coefficient matrix and matrix Y represents the output transform data. Equation 1 is equivalent to the matrix operation in Equation 2 on the next page.

In Equation 2, E is a matrix of scaling factors and the symbol $\otimes$, indicates that each element of $(C * X * C^T)$ is multiplied by the scaling factor in the same position in matrix E (scalar multiplication rather than matrix multiplication).

$$Y=(C*X*C^T)\otimes E = \begin{pmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{bmatrix} & \begin{bmatrix} X \end{bmatrix} & \begin{bmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{bmatrix} \end{pmatrix} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix}$$

-- Equation 2

To simplify the implementation of the transform, d is approximated to be 0.5. To ensure that the transform remains orthogonal, b also needs to be modified so that:

$$a = 1/2 \quad d = 1/2 \quad b = \sqrt{2/5}$$

To avoid multiplication by $d = 1/2$ in the “core” transform $C*X*C^T$, which would result in loss of accuracy using integer arithmetic, the 2nd and 4th rows of matrix C and the 2nd and 4th columns of matrix C^T are scaled by the post-scaling matrix E_f using matrix operation. Then the transform becomes:

$$Y=(C_f*X*C_f^T)\otimes E_f = \begin{pmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} & \begin{bmatrix} X \end{bmatrix} & \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \end{pmatrix} \otimes \begin{bmatrix} a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \\ a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \end{bmatrix}$$

-- Equation 3

This transform is an approximation to the 4x4 DCT. Because of the change to factors d and b , the output of the new transform will not be identical to the 4x4 DCT. In fact the integer transform is only $C_f*X*C_f^T$. The scaling factor E_f in Equation 3 is not an integer any more, so it is incorporated into the quantization process.

The 4x4 forward integer transform in the implementation is given by Equation 4:

$$W=(C_f*X*C_f^T) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \begin{bmatrix} X \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix}$$

-- Equation 4

Inverse 4x4 integer transform

Similarly, Equation 5 for the inverse integer transform can be deduced. In Equation 5, Y is pre-scaled by multiplying each coefficient by the appropriate weighting factor from matrix E_i . The factors $\pm 1/2$ in the matrices C_i and C_i^T can be implemented with a

right-shift without a significant loss of accuracy because the coefficients Y are pre-scaled. Again the pre-scaling factor E_i is incorporated into the quantization process.

$$X' = C_i^T (Y \otimes E_i) C_i = \begin{bmatrix} 1 & 1 & 1 & 1/2 \\ 1 & 1/2 & -1 & -1 \\ 1 & -1/2 & -1 & 1 \\ 1 & -1 & 1 & -1/2 \end{bmatrix} \left(\begin{bmatrix} Y \end{bmatrix} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1/2 & -1/2 & -1 \\ 1 & -1 & -1 & 1 \\ 1/2 & -1 & 1 & -1/2 \end{bmatrix}$$

-- Equation 5

The actual inverse integer transform is given below:

$$X' = C_i^T H C_i = \begin{bmatrix} 1 & 1 & 1 & 1/2 \\ 1 & 1/2 & -1 & -1 \\ 1 & -1/2 & -1 & 1 \\ 1 & -1 & 1 & -1/2 \end{bmatrix} \begin{bmatrix} H \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1/2 & -1/2 & -1 \\ 1 & -1 & -1 & 1 \\ 1/2 & -1 & 1 & -1/2 \end{bmatrix}$$

-- Equation 6

A summary of the advantages of the 4x4 integer transform is provided below:

Shorter word length transforms: For input pixels with 9-bit dynamic range (because they are residuals from 8-bit pixel data), the output after the integer transform is limited to 16 bits. Previous designs generally require 32-bit processing.

Exact match for the inverse transform: For the DCT, since there are multiplications in floating point, it is impractical to obtain an exact match to the ideal inverse transform. As a result, each decoder would produce a slightly different decoded video. In H.264, since all the operations are shifted either left or right only, the mismatch problem is solved [1].

Multiplier-free: In H.264 forward integer transform, multiplication by 2 can be performed by a left shift. And in the reverse integer transform, division by 2 can be performed by a right shift. The small error introduced is compensated by the large dynamic range of the input data, which will be explained in more detail in the quantization section [9].

3.2.3 Quantization for luma AC transformed coefficients

As explained in section 3.2.2, in order to avoid the floating-point operation, it incorporates the post- and pre-scaling matrices E_f and E_i into the quantization process.

Forward quantization

The basic operation of forward quantization is W_{ij}/Q_{step} . (W_{ij} is the output of the forward integer transform.) However, it has to incorporate post-scaling matrix E_f of Equation 3 into the forward quantization. Let PF_{ij} be the elements (a^2 , $ab/2$, $b^2/4$) of the post-scaling matrix E_f . Then the actual forward quantization is specified below:

$$Z_{ij} = \text{round}(W_{ij} * PF_{ij}/Q_{step}) \quad \text{-- Equation 7}$$

W_{ij} is the data element of matrix W after the forward integer transform. Z_{ij} is the element of matrix Z after forward quantization. Q_{step} is a fixed value from Table 1 indexed by QP .

A total of 52 values of Q_{step} are defined and are indexed by a quantization parameter, which is QP .

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

Table 1. Quantization step

Q_{step} is doubled for every increment of 6 in QP , which also means that Q_{step} increases by 12.5% for every increment of 1 in QP . The wide range of Q_{step} enables the encoder to accurately and flexibly control the trade-off between bit rate and quality. Q_{step} and a^2 , $ab/2$, $b^2/4$ are in floating point format. So Equation 7 involves floating point manipulation. In order to have a simpler implementation in ASIC design, we let:

$$PF_{ij}/Q_{step} = MF_{ij} / 2^{q_{bits}} \quad , \quad \text{where } q_{bits} = 15 + \text{floor}(QP/6) \quad \text{-- Equation 8}$$

Term $\text{floor}(QP/6)$ is the defined mathematical function at the beginning of the report, which means that if $(QP/6)$ is a floating point value, it will round the value of $(QP/6)$ to the integer closer to zero. PF_{ij} is a constant element of E_f .

Q_{step} is doubled for every increment of 6 of QP , and $2^{q_{bits}}$ will be doubled as well for every increment of 6 of QP because q_{bits} is increased by 1. This means that $MF_{ij}=PF_{ij}*2^{q_{bits}}/Q_{step}$ will repeat its value for every increment of 6 of QP . For

example, $MF_{ij}(QP=6) = MF_{ij}(QP=0)$. Thus it only needs to calculate MF_{ij} value for $QP=0\sim5$.

For example, if $QP=1$, then $Qstep=0.6875$. At position (0,0), $PF_{00} = a^2 = 0.25$. (value of a is 0.5 in Equation 1); $qbits = 15 + \text{floor}(1/6) = 15$. This implies that $MF_{00} = (0.25 \cdot 2^{15}) / 0.6875 = 11915.636 \cong 11916$.

The values of MF_{ij} are shown in Table 2. Some values are modified a bit for better perceptual quality.

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

Table 2. Values of MF_{ij} for $QP=0\sim5$.

Thus, Equation 7 becomes:

$$Z_{ij} = \text{round} (W_{ij} * MF_{ij} / 2^{qbits}) \quad \text{-- Equation 9}$$

In this way, the quantization has only integer multiplications and right shift operations instead of floating-point multiplications and divisions. In actual software or ASIC implementation, the following will be performed,

$$\begin{aligned} |Z_{ij}| &= (|W_{ij}| * MF_{ij} + f) \gg qbits \\ \text{sign}(Z_{ij}) &= \text{sign}(W_{ij}) \end{aligned} \quad \text{-- Equation 10}$$

Symbol $\gg$ stands for binary right shift. The factor f is added to compensate for the rounding loss of right shift. The factor $f=2^{qbits}/3$ is for intra block quantization and $f=2^{qbits}/6$ is for inter block quantization.

Inverse quantization

The basic operation of inverse quantization is $Z_{ij} * Qstep$. However, as mentioned, the pre-scaling matrix E_i of Equation 5 should be incorporated into inverse quantization. Thus the actual inverse quantization is specified as in Equation 11:

$$H_{ij} = Z_{ij} * Qstep * PF_{ij}' * 64. \quad \text{-- Equation 11}$$

Qstep is from Table 1. PF_{ij} is the element like (a^2, ab, b^2) from the pre-scaling matrix E_i . Scaling factor 64 is multiplied in order to minimize rounding errors in the inverse quantization. Besides, it also increases the input dynamic range to the inverse integer transform. Of course, this requires the output of the inverse integer transform to be divided by 64.

Let $MR_{ij} = Qstep * PF_{ij} * 64$. By a similar method as in the forward quantization, the inverse quantization is specified in Equation 12 with only integer multiplication and right shift operation.

$$H_{ij} = Z_{ij} * MR_{ij} * 2^{\text{floor}(QP/6)} \quad \text{-- Equation 12}$$

The values for MR_{ij} are shown in Table 3. It shows that MR_{ij} is much smaller than MF_{ij} . This results in the forward quantization having a bigger circuit in ASIC implementation than inverse quantization.

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

Table 3. Values of MR_{ij} for $QP=0\sim5$.

3.2.4 Quantization for chroma AC transformed coefficients

The chroma quantization process is similar to the luma quantization process. The difference is the value of quantization parameter used to address the quantization step. For the given input QP, the actual quantization parameter QP_c for chroma component is related to QP as shown below.

QP	<30	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
QP_c	=QP	29	30	31	32	32	33	34	34	35	35	36	36	37	37	37	38	38	38	39	39	39	39

QP_c is smaller than QP when $QP > 30$. This is to avoid visible color quantization artifacts [7]. After getting QP_c , it is used to index the MF_{ij} value in Table 2 for forward quantization and the MR_{ij} value in Table 3 for inverse quantization.

Besides, in the actual quantization equations, the QP should be replaced by QPc as well. The rest is exactly the same as those in luma quantization for either forward or inverse quantization.

3.2.5 Hadamard transform and quantization for luma DC coefficients

In section 3.2.1, case (2), for luma 16x16 intra prediction, after applying the forward 4x4 integer transform, there are 16 DC coefficients to form a 4x4 matrix labeled as -1 in Figure 11. It is expected that all these 16 coefficients will be big, since they are the average values of their respective 4x4 block samples. Much of the energy is concentrated in these DC coefficients, so the extra Hadamard transform is applied to de-correlate these 4x4 DC coefficients.

Forward Hadamard transform and forward quantization

The forward 4x4 Hadamard transform is shown below:

$$Y_{DC} = \begin{pmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} \\ \\ \\ \end{bmatrix} W_{DC} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \end{pmatrix} / 2 \quad \text{-- Equation 13}$$

W_{DC} is the input of the DC coefficients labeled as -1 in Figure 11. Y_{DC} is the output. Then the forward quantization will be performed on the output Y_{DC} as follows:

$$\begin{aligned} |Z_{DCij}| &= (|Y_{DCij}| * MF_{00} + 2f) >> (qbits + 1) \\ \text{sign}(Z_{DCij}) &= \text{sign}(Y_{DCij}) \end{aligned} \quad \text{-- Equation 14}$$

The parameter f and $qbits$ is the same as that in luma quantization. MF_{00} is also from Table 2 and depends on QP.

Inverse Hadamard transform and inverse quantization

At the decoder, the quantized data Z_{DCij} will go through inverse Hadamard transform followed by inverse quantization as shown in Figure 13. Note that the order is not inverse quantization followed by inverse Hadamard transform as expected. It is inverse Hadamard transform first, then followed by the inverse quantization.

The inverse Hadamard transform is the same as the forward Hadamard transform except that inverse Hadamard transform does not have division of 2.

Let WQ_{DCij} be the output after inverse Hadamard transform. Then the inverse quantization is given below:

$$\begin{aligned} \text{If } QP < 12, W'_{DCij} &= [WQ_{DCij} * MR_{00} + 2^{1-\text{floor}(QP/6)}] \gg (2 - \text{floor}(QP/6)) \\ \text{If } QP \geq 12, W'_{DCij} &= WQ_{DCij} * MR_{00} * 2^{\text{floor}(QP/6)-2} \end{aligned} \quad \text{-- Equation 15}$$

The output data will be inserted into the inverse quantized AC data to do the inverse integer transform.

3.2.6 Chroma DC transform and quantization

Forward chroma DC transform and forward quantization

Chroma DC transform and quantization is similar to the luma DC transform and quantization. The input is 2x2 DC coefficients instead of 4x4 DC coefficients. Equation 16 shows the forward chroma DC transform.

$$Y_{C_{DC}} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} W_{C_{DC}} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{-- Equation 16}$$

Then forward quantization is performed on $Y_{C_{DC}}$ as shown in Equation 17. It is exactly the same as that in section 3.2.5 for luma DC quantization except that the QP used to index MF_{00} is changed to QP_C .

$$\begin{aligned} |Z_{C_{DCij}}| &= (|Y_{C_{DCij}}| * MF_{00} + 2^f) \gg (qbits + 1) \\ \text{sign}(Z_{C_{DCij}}) &= \text{sign}(Y_{C_{DCij}}) \end{aligned} \quad \text{-- Equation 17}$$

Inverse chroma transform and inverse quantization

At the decoder, inverse quantization is followed by inverse chroma DC transform as expected. The inverse quantization is shown in Equation 18:

$$\begin{aligned} \text{If } QP_C \geq 6, Y_{C'_{DCij}} &= Z_{C_{DCij}} * MR_{00} * 2^{\text{floor}(QP_C/6)-1} \\ \text{If } QP_C < 6, Y_{C'_{DCij}} &= Z_{C_{DCij}} * MR_{00} \gg 1 \end{aligned} \quad \text{-- Equation 18}$$

The inverse chroma DC transform is exactly the same as the forward chroma DC transform with $Y_{C'_{DCij}}$ as the input.

3.3 Reorder

During encoding, after the forward transform and quantization in section 3.2, the resulting coefficients are in the frequency domain. For each 4x4 block, there are 16 coefficients. H.264 defines a certain way to form the 4x4 block into a scanned array. For example, Figure 15 in section 3.4 gives a scanning pattern for frame mode pictures. This scanned array has some statistical properties suitable for entropy coding. This will be shown in section 3.4. From the scanned array, information such as `total_coeff`, `total_zeros`, `run_before`, and `level` is extracted during encoding process. The information is then encoded by the CAVLC encoding block to generate the bitstream.

During decoding, in section 3.4, the CAVLC decoding block will parse the input bitstream and produce four types of output data, namely `total_coeff`, `level`, `total_zeros` and `run_before`, for either the 4x4 or the 2x2 block.

Then the reorder functional block in the decoder receives the four types of input data from the CAVLC decoding block. Based on these inputs, the function of reorder in the decoding process is to insert the zeros between the levels (non-zero coefficients). Generally, for each level, there is a corresponding `run_before`. The number of zeros inserted between the level each time is the value of the corresponding `run_before`. In this way, the scanned array of sixteen coefficients for a 4x4 block or four coefficients for a 2x2 block is generated by the reorder block during the decoding process.

By right, in the decoding process, after retrieving the scanned array, the 4x4 block should be reconstructed by reorder functional block according to the pattern in Figure 15 in section 3.4. However, in this design, the scanned array is saved into the memory first. Then according to the pattern in Figure 15 on the next page, the 4x4 block could be reconstructed at the system level.

3.4 Entropy coding

3.4.1 Introduction

In Figure 9 of the decoder, there is a functional block called entropy decode. In Figure 10 of the standard syntax hierarchy, the syntax element above slice layer like sequence or picture parameters are coded as fixed or variable length binary codes. All the syntax elements are coded using Exponential Golomb variable length codes with a regular construction. At the slice layer and below, syntax elements like motion vector or residual data are coded depending on the `entropy_coding_mode`.

$$\text{entropy_coding_mode} = \begin{cases} 0 & \begin{cases} 4 \times 4 \text{ (or } 2 \times 2 \text{ Chroma DC) residual block data: CAVLC} \\ \text{Others: Exp-Golomb codes} \end{cases} \\ 1 & \text{CABAC} \end{cases}$$

As for baseline profile, CAVLC (Context Adaptive Variable Length Code) is selected. It is based on assigning shorter codewords to symbols that occur with higher probabilities, and longer codewords to symbols with less frequent occurrences [14]. The CAVLC algorithm for H.264 is first proposed in [15]. CAVLC is based on the variable length code, which is also called the Huffman code. CAVLC has a series of VLC tables to be switched, depending on the previously coded elements. This results in its adaptive nature. In this section, its algorithm will be presented.

In the encoding process, CAVLC is designed to code the 4x4 (or 2x2 for Chroma DC) transformed and quantized coefficients. The 4x4 residual data is zig-zag scanned into an array of data from DC value to high frequency data. For example, Figure 15 shows the frame mode zig-zag scanning:

Figure 15. zig-zag scanning pattern for frame mode

For the example shown, the scanned array is (0, 3, 0, 1, -1, -1, 0, 1, 0, 0 ...). This scanned array data is selected for illustration in the CAVLC coding process. The following observations can be made:

- 1) Many quantized coefficients are zeros. So CAVLC uses run_before, level pairs to represent strings of zeros compactly.
- 2) At the high frequency positions, quite likely there is +/- 1. These continuous +/- 1 are called trailing ones. The CAVLC block can code in special manner for up to 3 trailing ones located at end of the scanned order array.
- 3) When coding the normal non-zero coefficients (levels), it codes the current level based on previously coded level.

Properties like 3) make this entropy coding method context adaptive. The data array is coded in reversed zig-zag scanned order. In the example, coefficient “1” will be coded first and coefficient “3” last.

3.4.2 CAVLC decoding process

The CAVLC codewords for a residual block includes the following five sets of codewords. The decoding process also follows steps from 1) to 5) in this section.

1) Total non-zero coefficients (total_coeff) and number of trailing ones (num_trailingones)

In this example, total_coeff = 5, and num_trailingones = 3. From the standard document [2], the codeword for total_coeff and num_trailingones is “0000 100” if $0 \leq nc < 2$.

The factor nc is the average total_coeff of the neighboring 4x4 blocks to the above and left of the current block. Depending on the value of nc, the possibility of total_coeff and num_trailingones combination is different. The codeword will be assigned accordingly, so as to make the codeword adaptive to the neighboring blocks (context adaptive). nc = -1 is for the case when the block is 2x2 Chroma DC data.

2) Sign of the trailing ones (trailing_ones_sign_flag).

This codes the sign of the trailing ones, which are the ± 1 s at the end of the scanned array. Bit value 0 is for +1 and bit value 1 is for -1.

CAVLC allows maximum 3 continuous trailing ones to be coded in this special manner. If there are more than 3 trailing ones, the extra ones will be coded as normal level in the level coding. For example, in the scanned array (0, 3, 0, 1, -1, -1, 0, 1, 0, ...) if it only considers non-zero coefficients, there are 4 continuous ones. However, only the last 3 ones (-1, -1, 1) are coded and the coding bits will be ("1", "1", "0"). The codeword for the trailing ones will be "011" in reversed order.

3) Non-zero coefficients value (level).

The level decoding process is summarized from [2] below:

```

/* Initialize the table index vlcnum
If (total_coeff>10 && num_trailingones < 3)
    vlcnum=1;
Else
    vlcnum=0;
End if;

Decode the trailing ones at this line

/* Start to decode the level value, looping over all the level
For (k=total_coeff - 1- num_trailingones; k ≥ 0; k - -)
{
    Decode one level each time at this line (explained on the next page)

    /* After decoding each level, there may be some modifications
    If (k= total_coeff - 1- num_trailingones && num_trailingones < 3)
        If (level>0)
            Level++;
        Else
            Level--;
        End if;
    End if;

    /* Update the table index vlcnum
    If (|level|>incvlc[vlcnum])
        vlcnum++;
    End if;

    If (k= total_coeff - 1- num_trailingones && |level|>3)
        vlcnum=2;
    End if;
}

End;

```

The variable vlcnum selects one of the seven level codeword tables in [15] for level decoding. Small vlcnum selects the table appropriate for small level. Big vlcnum selects the table for big level. In this way, more bits could be saved. Two tables are selected for illustration as shown in Table 4 and 5. For example, for codeword

“0000000000000001xxxx” in Table 5, “0000000000000001” is called level_prefix, while “xxxx” is called level_suffix. The number of zeros in the level_prefix is called leading_zero. For this codeword, leading_zero=14.

First, the vlcnun is initialized to 1 or 0 depending on the condition. This is reasonable, because after the trailing ones, the first few levels to be decoded are at the end of the scanned array, and they are generally very small.

Then it starts to decode the level one by one. Since the coding is done in reversed scanned order, the level is also decoded in reversed scanned order, from high frequency to low frequency.

Lev-VLC0 (vlcnun=0)		
Num_leading_zero	codeword	Level (±1, ±2..)
0	1	1
1	01	-1
2	001	2
3	0001	-2
..	..	..
13	00000000000001	-7
14	00000000000001xxxx	±8 to ±15
15	000000000000001xxxxxxxxxxxx	±16 ->

Table 4. Level codeword table of vlcnun=0 [15]

Lev-VLC4 (vlcnun=4)		
Num_leading_zero	Codeword	Level (±1, ±2..)
0	1xxxx	±1 to ±8
1	01xxxx	±9 to ±16
..	..	..
14	00000000000001xxxx	±113 to ±120
15	000000000000001xxxxxxxxxxxxs	±121 ->

Table 5. Level codeword table of vlcnun=4 [15]

During decoding, given vlcnun, it will select the level codeword table. Then with the input bitstream, it will find a unique match with the level codeword and then outputs leading_zero and level_suffix. With the value of leading_zero, vlcnun and level_suffix, the standard document [2] provides a standard algorithm to compute the actual value for level. It is shown on the next page.

```

x = leading_zero<<(vlcnum+1) + level_suffix
If x[0] = '0',
    level= ((x+2)>>1)
Else x[0] = '1',
    level= ((-x-1)>>1).

```

The symbol << means binary left shift, while >> means binary right shift. Expression $x[0]$ stands for the least significant bit of x . In this way, the codeword could be mapped to the actual level value.

Finally it needs to update of the $vlcnum$. If the decoded level $> incvlc[vlcnum]$, then it will increment $vlcnum$ to select the level codeword table more suitable for bigger level. The threshold value to compare with is $incvlc[] = \{0, 3, 6, 12, 24, 48\}$. In this way, fewer coding bits can be realized by checking the value of previously coded level. This implies that the level coding is context adaptive.

In this example, the scanned array is $\{0, 3, 0, 1, -1, -1, 0, 1, 0, \dots\}$. After the 3 trailing ones, the level left is 3 and 1. Since coding is done in reverse scanned order, the first level to be coded is 1, followed by 3. For this case, $vlcnum$ is initialized to 0, so level codeword table ($vlcnum=0$) is selected. The corresponding codeword for level = 1 is "1". Then $vlcnum$ is updated to 1. So level codeword table ($vlcnum=1$) is selected for coding level = 3 and the codeword is "0010".

4) Total number of zeros before the last non-zero coefficients (total_zeros).

If $total_coeff$ is bigger, that means there are many non-zero coefficients. In common sense, $total_zeros$ will be smaller. So the codeword for $total_zeros$ is designed depending on $total_coeff$. The $total_coeff$ has already been coded. So in the example of Figure 15, the codeword for $total_zeros = 3$ is "111" by indexing the $total_zeros$ codeword table in the standard document [2] using $total_coeff=5$.

5) The number of zeros before each non-zero coefficients (run_before)

Variable run_before represents the number of zeros lying before each non-zero coefficient including that for the trailing ones. The algorithm is shown below:

```

for (i=1; i<total_coeff; i++)
{
    if(zerosLeft>0)
        run[i]=run_before
    else

```



```
run[i]=0
end if;
zerosLeft=zerosLeft - run[i];
}
end;
run[ total_coeff] = zerosLeft
```

The variable zerosLeft represents number of zeros left in the scanned array after deducting each decoded run_before from total_zeros. In the example, total_zeros=3, so zerosLeft is initialized to 3.

Figure 16. run_before illustration

As shown in Figure 16, the first non-zero coefficient is 1, and the number of zeros before it is one. Check the corresponding run_before codeword table in [2], and the codeword is “10”. Update the zerosLeft = 3 – 1 = 2. The second non-zero coefficient is -1, and there is no zero before it. The codeword is “1”. So zerosLeft remains to be 2. The third non-zero coefficient is -1, and there is no zero before it. The corresponding codeword is “1”. And zerosLeft remains to be 2. The fourth non-zero coefficient is 1, and the number of zeros before it is one. The codeword is “01”. So zerosLeft = 2-1=1. The last non-zero coefficient is 3, and no codeword is required for it since number of zeros before it is zerosLeft, which could be calculated from the decoded variables.

So for the scan array (0, 3, 0, 1, -1, -1, 0, 1, 0, ...), the CAVLC coded bitstream “0000100 011 1 0010 111 10 1 1 01” could be successfully parsed and the correct values for four types of data (total_coeff, level, total_zeros and run_before) could be decoded. The trailing ones are included into level type in the output.

3.5 Intra prediction

For the intra prediction process, the prediction modes will be obtained first. During encoding, the intra prediction may be either 16x16 intra prediction or 4x4 intra prediction for the luma data. Figure 17 shows a macroblock with the sixteen 4x4 blocks indexed (luma4x4BlkIdx) and the co-ordinate system. If it is the 4x4 intra prediction, then every 4x4 block itself will be predicted based on its neighboring pixels. Thus the 4x4 intra prediction process needs to be repeated sixteen times to finish the luma component of a macroblock. If it is the 16x16 intra prediction, then the whole macroblock can be predicted from the macroblock’s neighboring pixels. The chroma component is assigned with only 8x8 intra prediction, which is similar to the luma 16x16 intra prediction. The details will be described in the following sections.

Figure 17. luma4x4BlkIdx of a macroblock and the co-ordinate system

3.5.1 Luma 4x4 intra prediction

Figure 18. Neighboring pixel locations for a 4x4 luma block

As shown in Figure 18, the blank squares are sixteen pixels of the 4x4 luma block to be predicted. The squares labeled with capital letters are its neighboring pixels of the neighboring 4x4 blocks. There are nine prediction modes for luma intra 4x4 prediction as shown in Table 6. The corresponding prediction directions are shown in Figure 19.

Intra4x4PredMode[luma4x4BlkIdx]	Name of Intra4x4PredMode[luma4x4BlkIdx]
0	Intra_4x4_Vertical (prediction mode)
1	Intra_4x4_Horizontal (prediction mode)
2	Intra_4x4_DC (prediction mode)
3	Intra_4x4_Diagonal_Down_Left (prediction mode)
4	Intra_4x4_Diagonal_Down_Right (prediction mode)
5	Intra_4x4_Vertical_Right (prediction mode)
6	Intra_4x4_Horizontal_Down (prediction mode)
7	Intra_4x4_Vertical_Left (prediction mode)
8	Intra_4x4_Horizontal_Up (prediction mode)

Table 6. 9 luma 4x4 intra prediction modes

Figure 19. The luma 4x4 intra prediction directions

Before doing the luma 4x4 intra prediction, the process will initialize the values for the neighboring pixels (X, A, ...) labeled in Figure 18 depending on the availability of the neighboring pixels. The condition that “block_available_up” is true means that the up 4x4 block as indicated in Figure 18 (for pixels A, B, C and D) is available. This condition may be false, when the current block to be predicted lies at the top of the picture frame. The same applies to the other three block available conditions. The neighboring pixels initializations are shown below.

```
If (block_available_up)
    A, B, C, D are assigned from the memory;
Else
    A=B=C=D=128;
End if;

If (block_available_up_right)
    E, F, G, H are assigned from the memory;
Else
```



```

        E=F=G=H=D;
    End if;

    If (block_available_left)
        I, J, K, L are assigned from the memory;
    Else
        I=J=K=L=128;
    End if;

    If (block_available_up_left)
        X is assigned from the memory;
    Else
        X=128;
    End if;

```

Then the nine luma 4x4 intra prediction modes are specified with $\text{mpr}(x,y)$ as output and neighboring pixels' values as inputs as illustrated below:

a) DC prediction

```

    If (block_available_up && block_available_left)
        S0=(A+B+C+D+I+J+K+L+4)/8;
    Else if (!block_available_up && block_available_left)
        S0=(I+J+K+L+2)/4;
    Else if (block_available_up && !block_available_left)
        S0=(A+B+C+D+2)/4;
    Else
        S0=128;
    End if;

     $\text{mpr}(x, y) = S0$  for  $x=0\sim3, y=0\sim3$ .

```

b) Vertical prediction

```

     $\text{mpr}(0,0) = A$ 
     $\text{mpr}(1,0) = B$ 
     $\text{mpr}(2,0) = C$ 
     $\text{mpr}(3,0) = D$ 

```

```

     $\text{mpr}(0,1) = A$ 
     $\text{mpr}(1,1) = B$ 
     $\text{mpr}(2,1) = C$ 
     $\text{mpr}(3,1) = D$ 

```

```

     $\text{mpr}(0,2) = A$ 
     $\text{mpr}(1,2) = B$ 
     $\text{mpr}(2,2) = C$ 
     $\text{mpr}(3,2) = D$ 

```

```

     $\text{mpr}(0,3) = A$ 
     $\text{mpr}(1,3) = B$ 
     $\text{mpr}(2,3) = C$ 
     $\text{mpr}(3,3) = D$ 

```

-- Equation 19

c) Horizontal prediction

```

     $\text{mpr}(0,0) = I$ 
     $\text{mpr}(1,0) = I$ 

```

$$\begin{aligned} \text{mpr}(2,0) &= I \\ \text{mpr}(3,0) &= I \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,1) &= J \\ \text{mpr}(1,1) &= J \\ \text{mpr}(2,1) &= J \\ \text{mpr}(3,1) &= J \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,2) &= K \\ \text{mpr}(1,2) &= K \\ \text{mpr}(2,2) &= K \\ \text{mpr}(3,2) &= K \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,3) &= L \\ \text{mpr}(1,3) &= L \\ \text{mpr}(2,3) &= L \\ \text{mpr}(3,3) &= L \end{aligned}$$

-- Equation 20

d) Diag_down_right prediction

$$\begin{aligned} \text{mpr}(0,0) &= (I + 2*X + A + 2) / 4 \\ \text{mpr}(1,0) &= (X + 2*A + B + 2) / 4 \\ \text{mpr}(2,0) &= (A + 2*B + C + 2) / 4 \\ \text{mpr}(3,0) &= (B + 2*C + D + 2) / 4 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,1) &= (J + 2*I + X + 2) / 4 \\ \text{mpr}(1,1) &= (I + 2*X + A + 2) / 4 \\ \text{mpr}(2,1) &= (X + 2*A + B + 2) / 4 \\ \text{mpr}(3,1) &= (A + 2*B + C + 2) / 4 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,2) &= (K + 2*J + I + 2) / 4 \\ \text{mpr}(1,2) &= (J + 2*I + X + 2) / 4 \\ \text{mpr}(2,2) &= (I + 2*X + A + 2) / 4 \\ \text{mpr}(3,2) &= (X + 2*A + B + 2) / 4 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,3) &= (L + 2*K + J + 2) / 4 \\ \text{mpr}(1,3) &= (K + 2*J + I + 2) / 4 \\ \text{mpr}(2,3) &= (J + 2*I + X + 2) / 4 \\ \text{mpr}(3,3) &= (I + 2*X + A + 2) / 4 \end{aligned}$$

-- Equation 21

e) Diag_down_left prediction

$$\begin{aligned} \text{mpr}(0,0) &= (A + C + 2*B + 2) / 4 \\ \text{mpr}(1,0) &= (B + D + 2*C + 2) / 4 \\ \text{mpr}(2,0) &= (C + E + 2*D + 2) / 4 \\ \text{mpr}(3,0) &= (D + F + 2*E + 2) / 4 \end{aligned}$$

$$\begin{aligned} \text{mpr}(1,1) &= (B + D + 2*C + 2) / 4 \\ \text{mpr}(2,1) &= (C + E + 2*D + 2) / 4 \\ \text{mpr}(3,1) &= (D + F + 2*E + 2) / 4 \\ \text{mpr}(4,1) &= (E + G + 2*F + 2) / 4 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,2) &= (C + E + 2*D + 2) / 4 \\ \text{mpr}(1,2) &= (D + F + 2*E + 2) / 4 \\ \text{mpr}(2,2) &= (E + G + 2*F + 2) / 4 \\ \text{mpr}(3,2) &= (F + H + 2*G + 2) / 4 \end{aligned}$$

$$\text{mpr}(0,3) = (D + F + 2*E + 2) / 4$$

$$\begin{aligned} \text{mpr}(1,3) &= (E + G + 2 \cdot F + 2) / 4 \\ \text{mpr}(2,3) &= (F + H + 2 \cdot G + 2) / 4 \\ \text{mpr}(3,3) &= (G + 3 \cdot H + 2) / 4 \end{aligned}$$

-- Equation 22

f) Vertical_right prediction

$$\begin{aligned} \text{mpr}(0,0) &= (X + A + 1) / 2 \\ \text{mpr}(1,0) &= (A + B + 1) / 2 \\ \text{mpr}(2,0) &= (B + C + 1) / 2 \\ \text{mpr}(3,0) &= (C + D + 1) / 2 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,1) &= (I + 2 \cdot X + A + 2) / 4 \\ \text{mpr}(1,1) &= (X + 2 \cdot A + B + 2) / 4 \\ \text{mpr}(2,1) &= (A + 2 \cdot B + C + 2) / 4 \\ \text{mpr}(3,1) &= (B + 2 \cdot C + D + 2) / 4 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,2) &= (X + 2 \cdot I + J + 2) / 4 \\ \text{mpr}(1,2) &= (X + A + 1) / 2 \\ \text{mpr}(2,2) &= (A + B + 1) / 2 \\ \text{mpr}(3,2) &= (B + C + 1) / 2 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,3) &= (I + 2 \cdot J + K + 2) / 4 \\ \text{mpr}(1,3) &= (I + 2 \cdot X + A + 2) / 4 \\ \text{mpr}(2,3) &= (X + 2 \cdot A + B + 2) / 4 \\ \text{mpr}(3,3) &= (A + 2 \cdot B + C + 2) / 4 \end{aligned}$$

-- Equation 23

g) Vertical_left prediction

$$\begin{aligned} \text{mpr}(0,0) &= (A + B + 1) / 2 \\ \text{mpr}(1,0) &= (B + C + 1) / 2 \\ \text{mpr}(2,0) &= (C + D + 1) / 2 \\ \text{mpr}(3,0) &= (D + E + 1) / 2 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,1) &= (A + 2 \cdot B + C + 2) / 4 \\ \text{mpr}(1,1) &= (B + 2 \cdot C + D + 2) / 4 \\ \text{mpr}(2,1) &= (C + 2 \cdot D + E + 2) / 4 \\ \text{mpr}(3,1) &= (D + 2 \cdot E + F + 2) / 4 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,2) &= (B + C + 1) / 2 \\ \text{mpr}(1,2) &= (C + D + 1) / 2 \\ \text{mpr}(2,2) &= (D + E + 1) / 2 \\ \text{mpr}(3,2) &= (E + F + 1) / 2 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,3) &= (B + 2 \cdot C + D + 2) / 4 \\ \text{mpr}(1,3) &= (C + 2 \cdot D + E + 2) / 4 \\ \text{mpr}(2,3) &= (D + 2 \cdot E + F + 2) / 4 \\ \text{mpr}(3,3) &= (E + 2 \cdot F + G + 2) / 4 \end{aligned}$$

-- Equation 24

h) Horizontal_down prediction

$$\begin{aligned} \text{mpr}(0,0) &= (X + I + 1) / 2 \\ \text{mpr}(1,0) &= (I + 2 \cdot X + A + 2) / 4 \\ \text{mpr}(2,0) &= (X + 2 \cdot A + B + 2) / 4 \\ \text{mpr}(3,0) &= (A + 2 \cdot B + C + 2) / 4 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,1) &= (I + J + 1) / 2 \\ \text{mpr}(1,1) &= (X + 2 \cdot I + J + 2) / 4 \\ \text{mpr}(2,1) &= (X + I + 1) / 2 \\ \text{mpr}(3,1) &= (I + 2 \cdot X + A + 2) / 4 \end{aligned}$$

$$\begin{aligned} \text{mpr}(0,2) &= (J + K + 1) / 2 \\ \text{mpr}(1,2) &= (I + 2 * J + K + 2) / 4 \\ \text{mpr}(2,2) &= (I + J + 1) / 2 \\ \text{mpr}(3,2) &= (X + 2 * I + J + 2) / 4 \\ \\ \text{mpr}(0,3) &= (K + L + 1) / 2 \\ \text{mpr}(1,3) &= (J + 2 * K + L + 2) / 4 \\ \text{mpr}(2,3) &= (J + K + 1) / 2 \\ \text{mpr}(3,3) &= (I + 2 * J + K + 2) / 4 \end{aligned}$$

-- Equation 25

i) Horizontal up prediction

$$\begin{aligned} \text{mpr}(0,0) &= (I + J + 1) / 2 \\ \text{mpr}(1,0) &= (I + 2 * J + K + 2) / 4 \\ \text{mpr}(2,0) &= (J + K + 1) / 2 \\ \text{mpr}(3,0) &= (J + 2 * K + L + 2) / 4 \\ \\ \text{mpr}(0,1) &= (J + K + 1) / 2 \\ \text{mpr}(1,1) &= (J + 2 * K + L + 2) / 4 \\ \text{mpr}(2,1) &= (K + L + 1) / 2 \\ \text{mpr}(3,1) &= (K + 2 * L + L + 2) / 4 \\ \\ \text{mpr}(0,2) &= (K + L + 1) / 2 \\ \text{mpr}(1,2) &= (K + 2 * L + L + 2) / 4 \\ \text{mpr}(2,2) &= L \\ \text{mpr}(3,2) &= L \\ \\ \text{mpr}(0,3) &= L \\ \text{mpr}(1,3) &= L \\ \text{mpr}(2,3) &= L \\ \text{mpr}(3,3) &= L \end{aligned}$$

-- Equation 26

3.5.2 Luma 16x16 intra prediction

Figure 20 shows the neighboring pixel labels. For luma 16x16 intra prediction, it has four different modes for the luma data as in Table 7. The prediction direction could be deduced from the name easily.

Intra16x16PredMode	Name of Intra16x16PredMode
0	Intra_16x16_Vertical (prediction mode)
1	Intra_16x16_Horizontal (prediction mode)
2	Intra_16x16_DC (prediction mode)
3	Intra_16x16_Plane (prediction mode)

Table 7. Four prediction modes for Intra luma 16x16 intra prediction

The detailed descriptions for the four modes are shown in the following paragraphs. The neighboring pixels together with its availability are the input. The output is assigned to mpr(x, y).

Figure 20. Neighboring pixels and the predicted data for luma 16x16

(a) Intra_16x16_Vertical prediction mode

This mode is used when the pixels $p(x, -1)$ with $x = 0 \sim 15$ are marked as “available”.

$$\text{mpr}(x, y) = p(x, -1), \text{ with } x, y = 0 \sim 15 \quad \text{-- Equation 27}$$

(b) Intra_16x16_Horizontal prediction mode

This mode is used when the pixels $p(-1, y)$ with $y = 0 \sim 15$ are marked as “available”.

$$\text{mpr}(x, y) = p(-1, y), \text{ with } x, y = 0 \sim 15 \quad \text{-- Equation 28}$$

(c) Intra_16x16_DC prediction mode

This mode is used depending on whether the neighboring pixels are marked as “available” as follows.

(1) If all neighboring pixels $p(x, -1)$, with $x = 0 \sim 15$ and $p(-1, y)$, with $y = 0 \sim 15$ are marked as “available”, the prediction for all luma samples in the macroblock is given by:

$$\text{mpr}(x, y) = \sum_{x'=0}^{15} p(x', -1) + \sum_{x'=0}^{15} p(x', -1) + 16 \gg 5, \text{ with } x, y = 0 \sim 15 \quad \text{-- Equation 29}$$

(2) Otherwise, if any of the neighboring pixels $p(x, -1)$, with $x = 0 \sim 15$ are marked as “not available” and all of the neighboring pixels $p(-1, y)$, with $y = 0 \sim 15$ are marked as “available”, then the prediction for all luma pixels in the macroblock is given by:

$$\text{mpr}(x, y) = \sum_{y'=0}^{15} p(-1, y') + 8 \gg 4, \text{ with } x, y = 0 \sim 15 \quad \text{-- Equation 30}$$

(3) Otherwise, if any of the neighboring pixels $p(-1, y)$, with $y = 0 \sim 15$ are marked as "not available" and all of the neighboring pixels $p(x, -1)$, with $x = 0 \sim 15$ are marked as "available", the prediction for all luma pixels in the macroblock is given by:

$$\text{mpr}(x,y) = \sum_{x'=0}^{15} p(x', -1) + 8 \gg 4, \text{ with } x, y = 0 \sim 15 \quad \text{-- Equation 31}$$

(4) Otherwise (some of the neighboring pixels $p(x, -1)$, with $x = 0 \sim 15$ and some of the neighboring pixels $p(-1, y)$, with $y = 0 \sim 15$ are marked as "not available"), the prediction for all luma pixels in the macroblock is given by:

$$\text{mpr}(x,y) = 128, \text{ with } x, y = 0 \sim 15 \quad \text{-- Equation 32}$$

(d) Intra_16x16_Plane prediction mode

This Intra_16x16 prediction mode shall be used only when the pixels $p(x, -1)$ with $x = -1 \sim 15$ and $p(-1, y)$ with $y = 0 \sim 15$ are marked as "available for Intra_16x16 prediction".

$$\text{mpr}(x, y) = \text{Clip1}((a + b * (x - 7) + c * (y - 7) + 16) \gg 5), \text{ with } x, y = 0 \sim 15,$$

where:

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

$$b = (5 * H + 32) \gg 6$$

$$c = (5 * V + 32) \gg 6$$

$$H = \sum_{x'=0}^7 (x'+1) * (p(8+x', -1) - p(6-x', -1)) \quad V = \sum_{y'=0}^7 (y'+1) * (p(-1, 8+y') - p(-1, 6-y'))$$

$$(\text{Clip1}()) \text{ is the defined mathematical function at the beginning of the report} \quad \text{-- Equation 33}$$

3.5.3 Chroma 8x8 intra prediction

For video format 4:2:0, a macroblock contains two 8x8 chroma blocks. Only 8x8 intra prediction modes are defined for the chroma components. Chroma 8x8 intra prediction is similar to the luma 16x16 prediction and has four similar prediction modes. The detailed computations could be found in the document [2]. For a macroblock, the actual intra prediction mode for chroma component may be different from that of luma 16x16 prediction. However, for the two components of chroma data, Cb and Cr, their prediction modes are the same.

In the ASIC hardware design, chroma intra prediction could be multiplexed into the luma 16x16 prediction cases by adding a few extra circuitries.

3.6 Deblocking filter

3.6.1 Introduction

Blocking artifacts are introduced at the block edges. The two main sources are coarse quantization of the transformed coefficients and motion compensated prediction. So the deblocking filter process is required to do filtering for the reconstructed frames before they are displayed or used as reference frames for inter prediction.

In the standard document [2], it defines the filter process can only start when the entire reconstructed picture is available. The filter process is performed on a macroblock basis. The top left macroblock of the picture is filtered first. Then in the order from left to right, the first row of macroblocks is filtered one macroblock after another. Similarly, all the remaining rows of macroblocks are filtered, in the order from top to bottom, one row after another.

Figure 21. Deblocking filtering for a macroblock

Figure 21 shows the block edges for a macroblock. In Figure 21 (I), the bolded lines are the four vertical edges of the current luma macroblock. The filtering sequence in a macroblock is defined by the standard [2]: vertical edges are filtered first, one after another from left to right. For luma component, a, b, c and d are the four 4x4 blocks of the left neighboring macroblock. They serve as inputs when filtering the left most vertical edge. After vertical edges filtering, the pixels at two sides of the edges are modified. So in Figure 21 (II), these 4x4 blocks are all labeled with a prime, e.g. a', b' and so on, to indicate that they are modified.

After vertical edges filtering, the horizontal edges shown as bolded lines in Figure 21 (II) are filtered one after another, from top to bottom. For the luma component, x, y, z

and w are the four 4x4 blocks of upper neighboring macroblock. They serve as inputs when filtering the top most horizontal edge.

The filtering sequence for the different macroblocks and for the edges of a macroblock is important because during the filter process the sample values are modified. Samples above and to the left of the current macroblock may have already been modified by the deblocking filter process on the previous macroblocks. They (like the pixels of blocks a , b , c and d) will be used as input to the deblocking filter process on the current macroblock. These samples above and to the left of current macroblock may be modified again during the filtering of the current macroblock. The current macroblock samples are themselves modified during filtering of vertical edges. Then they are also used as inputs for the filtering of the horizontal edges and will be modified again.

Although the filtering sequence is defined as above, it is still possible to change the sequence as long as it does not change the actual filtering results.

Each luma vertical or horizontal edge has 16 samples. The vertical edges are to the left of the pixels $P_L(0, y)$, $P_L(4, y)$, $P_L(8, y)$ and $P_L(12, y)$ with $y = 0 \sim 15$; and the horizontal edges are above the pixels $P_L(x, 0)$, $P_L(x, 4)$, $P_L(x, 8)$ and $P_L(x, 12)$ with $x = 0 \sim 15$. The co-ordinate is shown in Figure 20. Each chroma vertical or horizontal edge has 8 samples. The vertical edges are to the left of the pixels $P_C(0, y)$ and $P_C(4, y)$ with $y = 0 \sim 7$; and the horizontal edges are above the pixels $P_C(x, 0)$ and $P_C(x, 4)$ with $x = 0 \sim 7$. The deblocking filter process is invoked for the luma and chroma components independently.

A 16-pixel vertical edge contains 16 elements as shown in Figure 22 (a). Each element contains 8 pixels at two sides of the edge (shown as bolded line in Figure 22). They are inputs to the filter process for each element (a) or (b). For each element, a variable called filtering strength bS ($0 \leq bS \leq 4$) is assigned and input to the filter process. Basically, bS measures the necessity of doing filtering for this edge element. The bigger bS is, the more important the filtering is for this edge element and a stronger filtering effect should result. If bS is zero, then the filtering is not required for this

edge element. The bS for the edge element of chroma component is equal to the corresponding edge element of the luma component.

Figure 22. Pixel labeling across the edge

There are variables α , β , and t_{C0} to be derived to perform the actual filter process. The derivation process is in the standard document [2] with a few tables. Generally, when the average value of the quantization parameters of pixels q_0 and p_0 are bigger, then α , β , and t_{C0} will be bigger. When bS is bigger, t_{C0} will also tend to be bigger.

Let: $\text{filterSamplesFlag} = (\text{bS} \neq 0 \ \&\& \ \text{Abs}(p_0 - q_0) < \alpha \ \&\& \ \text{Abs}(p_1 - p_0) < \beta \ \&\& \ \text{Abs}(q_1 - q_0) < \beta)$
Only when the variable filterSamplesFlag is true, then the filter process will be performed on this edge element.

As mentioned above, if $\text{bS} = 0$, no filtering will be applied to the edge element. When $\text{Abs}(p_0 - q_0)$, $\text{Abs}(p_1 - p_0)$ and $\text{Abs}(q_1 - q_0)$ are small, which generally will be the case, then the filter process is likely to be performed. However, if the absolute differences are big, there are two possible cases:

- 1) It is caused by the video compression, such as the quantization process.
- 2) The image itself appears to be like that.

So the compression standard sets certain threshold values like α and β . If the absolute differences are bigger than these threshold values, then they are more likely caused by the image content itself. In this case, no filtering is required, or else the unnecessary filtering will make the image appear to be blurred.

That is why α and β should increase with the quantization step. As for the bigger quantization step, the absolute differences between the sample values caused by the quantization process will be bigger as well. That means the possibility of this big

absolute differences caused by quantization process is higher. So the threshold values α and β need to be set bigger as well.

Let a_p and a_q be two variables specified by

$$a_p = \text{Abs}(p_2 - p_0)$$

$$a_q = \text{Abs}(q_2 - q_0)$$

The actual filtering algorithms are shown in the following subsections.

3.6.2 Filter process for edge element with $bS < 4$

Luma component

(a) p'_0 and q'_0

Let:

$$t_c = t_{c0} + ((a_p < \beta) ? 1 : 0) + ((a_q < \beta) ? 1 : 0)$$

The filtered result samples p'_0 and q'_0 are derived by

$$\Delta = (((q_0 - p_0) < 2) + (p_1 - q_1) + 4) >> 3$$

if $\Delta > t_c$ then

$$\Delta = t_c$$

elseif $\Delta < -t_c$ then

$$\Delta = -t_c$$

else

Δ unchanged

end if;

$$p'_0 = \text{Clip1}(p_0 + \Delta)$$

$$q'_0 = \text{Clip1}(q_0 - \Delta)$$

(b) p'_1 and q'_1

If a_p is less than β , then

$$p'_1 = p_1 + \text{Clip3}(-t_{c0}, t_{c0}, (p_2 + ((p_0 + q_0 + 1) >> 1) - (p_1 << 1)) >> 1)$$

else

$$p'_1 = p_1$$

end if;

If a_q is less than β , then

$$q'_1 = q_1 + \text{Clip3}(-t_{c0}, t_{c0}, (q_2 + ((p_0 + q_0 + 1) >> 1) - (q_1 << 1)) >> 1)$$

else

$$q'_1 = q_1$$

end if; /* $\text{Clip1}()$ and $\text{Clip3}()$ are the defined mathematical function at the beginning of the report

(c) p'_2 and q'_2 are unchanged

Chroma component

(a) $p'0$ and $q'0$

The only change is $t_c = t_{c0} + 1$. The rest follows the luma component computation.

(b) $p'1$ and $q'1$ are unchanged

(c) $p'2$ and $q'2$ are unchanged

3.6.3 Filter process for edge element with $bS = 4$

Luma component

(a) $p'0$, $p'1$ and $p'2$

If ($a_p < \beta$ && $Abs(p0 - q0) < ((\alpha >> 2) + 2)$), then

$$p'0 = (p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4) >> 3$$

$$p'1 = (p2 + p1 + p0 + q0 + 2) >> 2$$

$$p'2 = (2*p3 + 3*p2 + p1 + p0 + q0 + 4) >> 3$$

Else

$$p'0 = (2*p1 + p0 + q1 + 2) >> 2$$

$p'1$ and $p'2$ are unchanged

End if;

(b) $q'0$, $q'1$ and $q'2$

If ($a_q < \beta$ && $Abs(p0 - q0) < ((\alpha >> 2) + 2)$), then

$$q'0 = (p1 + 2*p0 + 2*q0 + 2*q1 + q2 + 4) >> 3$$

$$q'1 = (p0 + q0 + q1 + q2 + 2) >> 2$$

$$q'2 = (2*q3 + 3*q2 + q1 + q0 + p0 + 4) >> 3$$

Else

$$q'0 = (2*q1 + q0 + p1 + 2) >> 2$$

$q'1$ and $q'2$ are unchanged

End if;

Chroma component

(a) $p'0$, $p'1$ and $p'2$

$$p'0 = (2*p1 + p0 + q1 + 2) >> 2$$

$p'1$ and $p'2$ are unchanged

(b) $q'0$, $q'1$ and $q'2$

$$q'0 = (2*q1 + q0 + p1 + 2) >> 2$$

$q'1$ and $q'2$ are unchanged

Chapter 4 Overall hardware architecture

In this chapter, the overall design requirement and decoder hardware architecture are introduced. Then the design requirements of the functional blocks are analyzed. Finally, the literature review of some functional blocks of H.264 decoder hardware designs is presented.

4.1 Overall requirement and architecture design

As introduced in Chapter 1, this H.264 decoder is to be designed to work at real time for the HDTV application of video format 1280x720 at 60 Hz. So the H.264 decoder processing speed required for real time decoding is $(1280 \times 720 \times 60) / (16 \times 16) = 216,000$ macroblock/s.

Figure 23 shows a common hardware design of H.264 decoder. The input bitstream is connected to the Parser unit. The parameter sets and header information in the bitstream are then decoded and sent to the Processor for controlling or connected directly to data bus Bus1.

Figure 23. Hardware design of H.264 decoder

Data bus Bus1 is mainly used by the individual decoding units and System Memory. As shown, Bus1 is connected to the necessary functional blocks. There are buffers labeled as B0, B1, B2, B3, B4 and B5 between the functional blocks to hold temporary data. These buffers help to reduce the Bus1 bandwidth. For example, without buffer B2, data transmission between reorder block and inverse Q & T block may have to occupy Bus1, thus increasing the Bus1 bandwidth. Buffer B4 holds the reconstructed data before filtering. It may serve as input for the intra prediction.

Buffer B5 holds the intermediate filtered data. The filtered data may be sent to the output channel or to the System Memory for inter prediction.

Besides reducing the Bus2 bandwidth, these inter-functional block buffers also help to improve the pipeline operation efficiency. The macroblock level pipeline processing as shown in Figure 24 is implemented for the decoder design to accelerate the processing speed. For example, when the CAVLC block is decoding for the macroblock MB5 at time t0, inverse Q & T could process the macroblock MB4, which is previously decoded by CAVLC. Of course, the actual pipeline processing may not be the way as simple as shown in Figure 24.

	CAVLC	Reorder	Inverse Q & T	Prediction	Filter
t0	MB5	MB4	MB3	MB2	MB1
t2	MB6	MB5	MB4	MB3	MB2
t3	MB7	MB6	MB5	MB4	MB3

Figure 24. Macroblock level pipeline processing

In Figure 23, the buffers B0, B1, B2 and B3 are generally designed to hold the data of two macroblocks for each buffer. Each macroblock contains $16 \times 16 \times 1.5 = 384$ bytes. So B0, B1, B2 and B3 have a size of 768 bytes each. Buffer B4 holding the reconstructed pixels may be used as neighboring pixels input for intra prediction, so it must be able to keep at least one row of macroblocks data. There are $1280/16 = 80$ macroblocks in a row of the HDTV frame. So the size of B4 is $80 \times 384 = 30720$ bytes = 30 K bytes. Buffer B5 is designed to keep the temporary data of filter, so it also needs to keep at least one row of macroblock data, which is 30 K bytes.

The System Memory is designed to save the filtered data for inter prediction. One frame of HDTV of 1280×720 size including luma and chroma components is $1280 \times 720 \times 1.5 = 1.32$ M bytes. Since five reference frames are defined for H.264 baseline profile, the System Memory is designed to have a size of 6.6 M bytes.

In Figure 23, the filter block needs to transmit data to the output channel or System Memory through the bus Bus1. Every second, 60 frames are required to be transmitted

since the frame rate is 60 frames/s. So it requires a bandwidth of $60 \times 1.32 = 79.2$ M bytes/s for the filter block. The inter prediction needs to read data from System Memory through the bus Bus1. Assume all the frames require inter prediction, then according to [26], the worst case bandwidth requirement for inter prediction is 163 M bytes/s. So if neglecting the other data transmission, the total bandwidth requirement for Bus1 is 242.2 M bytes/s.

4.2 Design requirements for the functional blocks

In section 4.1, it is computed that the H.264 decoder processing speed required for real time decoding is 216,000 macroblock/s. As for the hardware architecture shown in Figure 23, this means the processing speed required for each functional block is also 216,000 macroblock/s. This is to say, the time period allocated to each functional block to process a macroblock is $1/216,000 = 4630$ ns.

Figure 25 shows the actual pipeline design in this H.264 decoder architecture. The Δt_1 , Δt_2 , Δt_3 and Δt_4 represent the buffer delays.

Figure 25. Macroblock level pipeline for H.264 decoder

The operating frequency of each functional block cannot be determined at this stage, because the number of clock cycles for each functional block to process a macroblock is unknown yet. For different macroblocks with different pixel contents, the number of clock cycles required by a functional block to process these macroblocks is different. In the worst case, the maximum number of clock cycles required to decode a macroblock would be determined. If the corresponding worst case period is shorter than 4630 ns, then real time decoding for the functional block can be achieved.

However, since there are inter-functional block buffers, they will help to improve the pipeline operation efficiency and also average out the processing period required by each functional block to decode one macroblock. This is to say, as long as the average processing period for a macroblock required by each functional block within certain

time interval is shorter than 4630 ns, the real time processing still can be achieved. However, this requires simulations on the real video contents to find the average processing period within certain time interval. Then with the buffers of enough size, as long as the average processing period is shorter than 4630 ns, the real time processing is still achievable.

4.3 Literature review of some functional blocks

A literature review on the hardware design of some functional blocks in H.264 decoder is provided below.

As compared to DCT in the previous standards, the integer transform has many advantages, such as multiplier free and shorter word length operations. In [7], an algorithm point of view analysis of the integer transform is described. Paper [16] presents an architecture for 8x8 DCT as shown in Figure 26.

Figure 26. 8x8 DCT architecture [16]

This architecture could be employed in the H.264 transform architecture as well. In [17], a similar architecture for 4x4 integer transform and Hadamard transform is designed. Due to the similar transform coefficients between integer transform and Hadamard transform, paper [17] combine the two transforms into one by multiplexing the transform coefficients. For example, coefficient $\frac{1}{2}$ in integer transform is changed to 1 in Hadamard transform. This multiplexed transform increases little in gates count as compared to a single integer or Hadamard transform. However, paper [17] does not consider the chroma DC transform and quantization component, which is necessary for a complete inverse quantization and transform design. Paper [25] presents a design for area optimized transform. The design produces one output each 7 clock cycles. It will be shown later in section 6.1 that the area optimized transform of [25] is too slow to meet the HDTV application.

In the old standards, there are various VLC decoders designed. However, in H.264, the specific codeword look-up tables have to be changed. In [24], it shows a common architecture for CAVLC design as shown in Figure 27. All the decoding blocks take one clock cycle to produce an output except that level decoding block needs two clock cycles. For level decoding, it decodes level_prefix first to generate leading_zero. Then level_suffix is decoded to compute the actual value of level. For the level computation, the design of [24] follows the H.264 standard algorithm directly to compute the value of level. That algorithm is easy for documentation, however, costly in terms of hardware.

Figure 27. Architecture of CAVLC [24]

Papers [18] [19] and [20] present the designs of deblocking filter. The design of [19] is like a direct implementation from the H.264 algorithm. It starts to do filtering until a complete frame is reconstructed. So in the design of [19], it requires the buffer B4 of a picture frame size in Figure 23. This is a much bigger buffer size compared to my design and this also requires the longer system latency.

Papers [18] and [20] present the macroblock based filtering architecture, which means filtering can start when a macroblock is reconstructed. So the buffer B4 could be of a size around several macroblocks.

The architecture presented in [18] is shown in Figure 28 on the next page. The thick lines represent 64 bits, which can transmit 8 pixels. The thin lines represent 32 bits,

which can transmit 4 pixels. The 1-D Filter block is the actual computation element doing the edge filtering.

First, one reconstructed macroblock data is loaded to the local memory. Then vertical edges filtering starts by reading 8 pixels from the local memory to the 1-D Filter block. The temporary filtered data at Net1 is saved back to local memory.

After vertical edges filtering finishes, the local memory will send out two 4x4 blocks data to Reg0 and Reg1. Then the transposed data at Net2 is sent to the 1-D Filter block to do horizontal edges filtering. Meanwhile, the filtered data at Net1 is saved back to Reg0 and Reg1. When the horizontal edges filtering of this two 4x4 blocks is finished, the saved filtered data in Reg0 and Reg1 will be latched out as output. At the same time, local memory is sending data to Reg0 and Reg1 to do horizontal edges filtering for other 4x4 blocks.

Figure 28. Architecture of deblocking filter [18]

However, the design in [18] needs to wait for loading an entire reconstructed macroblock data before actual filtering can start. So [20] improves the design by reducing the loading period. The equivalent loading period of the design in [20] is reduced to load four 4x4 blocks, instead of an entire macroblock. However, it still needs a local memory of one macroblock size to save the vertical edges filtered data. Then the horizontal edges filter process will start.

Chapter 5 Designs of the five functional blocks

In this chapter, the hardware designs of the five functional blocks are to be presented in details. They are the Inverse quantization and transform block, Reorder block, CAVLC block, Intra prediction block and Deblocking filter block. Their functional simulations are shown at the end of each section. Their algorithms have been described in the corresponding sections in Chapter 3. So it is helpful to refer back to their algorithms if necessary.

5.1 Inverse quantization and transform

In the algorithm section 3.2, it shows that there is a need to design six components for the inverse Q and T block, namely, inverse Hadamard transform, luma DC quantization, AC quantization (for both luma and chroma), integer transform, chroma DC transform and chroma DC quantization. As the quantization and transform are closely related, especially for the luma 16x16 intra prediction mode where the inverse Hadamard transform is performed first, followed by inverse quantization, and subsequently the inverse integer transform, it is more efficient and meaningful to integrate the design of quantization and transform components.

Hence it is proposed the integrated architecture for inverse Q and T as shown in Figure 29. This integrated circuit is a complete design able to perform the inverse Q and T. In comparison, the paper [17] proposed the architecture for only the integer and Hadamard transform operation. The data inputs are four 16-bit wide buses. This is to meet the requirement of the parallel operations of the 4x4 transform. In Figure 29, the thick lines represent 4 16-bit data bus and the thin lines represent 16-bit data bus except for MR_{ij}, which is 5-bit wide.

QP selects the MR_{ij} to be used in the inverse quantization. The multiplexer labeled as “1” selects either direct inputs or inverse AC quantized inputs. Direct inputs are needed for Hadamard transform for luma 16x16 intra prediction residual data and for the DC transform of chroma components. The multiplexer labeled as “2” inserts the DC data into each 4x4 AC data at the proper time. The multiplexer labeled as “3” selects either the chroma DC or luma DC data.

Figure 29. Inverse quantization and transform architecture

For example, if it is luma 16x16 intra prediction, the input data goes through multiplexer “1” for the “Hadamard transform” first, and then followed by the “Inverse DC quantization”, whose output goes through multiplexer “3”, then “2” and ready to be inserted into the AC elements. The remaining input data goes for “Inverse AC quantization”. Its output combines with the DC data from multiplexer “2” and goes through multiplexer “1” for 4x4 inverse integer transform.

Quantization values MR_{ij} are stored as 5 bits hardwired binary numbers instead of ROM, as there are only 18 MR_{ij} values with the biggest value of 29.

5.1.1 Inverse Hadamard transform

As mentioned, paper [16] suggests an architecture for 8x8 DCT. In fact, Hadamard transform is very similar to DCT. Paper [17] adopts the architecture as shown in Figure 30.

Figure 30. Hadamard transform architecture

1-D Hadamard transform

Equation 13 in section 3.2.5 shows the complete Hadamard transform. The 1-D Hadamard transform is specified in Equation 34:

$$M = \begin{bmatrix} w_{00} & w_{01} & w_{02} & w_{03} \\ w_{10} & w_{11} & w_{12} & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \\ w_{30} & w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \quad \text{-- Equation 34}$$

Let $W_0 = [w_{00} \ w_{01} \ w_{02} \ w_{03}]$ and let m_{00} , m_{01} , m_{02} and m_{03} be the four elements of the first row of M . Then Equation 35 is derived:

$$\begin{aligned} m_{00} &= [1 \ 1 \ 1 \ 1] * W_0 = w_{00} + w_{01} + w_{02} + w_{03}; \\ m_{01} &= [1 \ 1 \ -1 \ -1] * W_0 = w_{00} + w_{01} - w_{02} - w_{03}; \\ m_{02} &= [1 \ -1 \ -1 \ 1] * W_0 = w_{00} - w_{01} - w_{02} + w_{03}; \\ m_{03} &= [1 \ -1 \ 1 \ -1] * W_0 = w_{00} - w_{01} + w_{02} - w_{03}; \end{aligned} \quad \text{-- Equation 35}$$

It is a direct translation in mathematics without taking into account hardware resources like adders. It requires 12 additions/subtractions.

Let $a = w_{00} + w_{02}$	Then $m_{00} = a + c$
$b = w_{00} - w_{02}$	$m_{01} = b + d$
$c = w_{01} + w_{03}$	$m_{02} = b - d$
$d = w_{01} - w_{03}$	$m_{03} = a - c$

Here common operations are grouped together and shared. This results in only 8 additions/subtractions of the 1-D transform, saving 4 additions/subtractions.

Transposed register

In the matrix operation of Equation 13, it could be seen that before the second 1-D Hadamard transform (B) could be performed, the matrix M of Equation 34 needs to be transposed.

Figure 31 shows the transposed register structure. The operations are illustrated in paper [16]. The squares are registers. There are four multiplexers, each of which is of 4 inputs and 1 output.

Figure 31. Transposed register

5.1.2 Inverse 4x4 integer transform

If some ± 1 coefficients in Equation 13 are changed to $\pm \frac{1}{2}$, the Hadamard transform becomes the 4x4 integer transform in Equation 6. So paper [17] suggests a good idea of multiplexing the Hadamard transform with the 4x4 integer transform.

Besides, in Figure 30, 1-D Hadamard transform A and B is the same. So it is proposed in this project that the two 1-D transforms can be multiplexed into one as shown in Figure 32 to reduce the hardware. In this way, the hardware for transform contains only one 1-D transform instead of two.

Figure 32. Modified Integer transform or Hadamard transform

The modified design in Figure 32 can only latch in one set of four input elements every two clock cycles as compared to Figure 30, which can latch in one set of four input elements every clock cycle. This is because in Figure 32, one clock cycle is required to perform the 1-D transform for the input data and another clock cycle for 1-D transform for the data from transposed register. This reduces the circuit speed by half.

However, as shown section 6.1, this speed is fast enough for the HDTV application. So the architecture of Figure 32 is adopted in the design for the Hadamard and integer transform.

5.1.3 Inverse luma AC quantization

Equation 12 in section 3.2.3 is $H_{ij} = W_{ij} * MR_{ij} * 2^{\text{floor}(QP/6)}$.

Since MR_{ij} is of 5 bits only, $W_{ij} * MR_{ij}$ could be implemented using a multiplier of two inputs, which are 16 bits and 5 bits, instead of 16 bits and 16 bits. In the previous section 5.1.2, it is designed to latch in one set of four input elements every two clock cycles, so the inverse AC quantizations should be designed to perform quantization on four input elements every two clock cycles. This means that two times of inverse AC quantizations are required every clock cycle. So only two multipliers ($W_{ij} * MR_{ij}$) instead of four are required. Multiplication of $2^{\text{floor}(QP/6)}$ can be done by left shift operations.

Figure 33 below shows the architecture. AC_Q is the actual computational part performing the inverse AC quantization.

Figure 33. Inverse AC quantization

5.1.4 Chroma DC transform

Expand the matrix multiplication Equation 16 in section 3.2.6:

$$YC_{00} = WC_{00} + WC_{10} + WC_{01} + WC_{11}$$

$$YC_{10} = WC_{00} + WC_{01} - WC_{10} - WC_{11}$$

$$YC_{11} = WC_{00} - WC_{01} - WC_{10} + WC_{11}$$

$$YC_{01} = WC_{00} - WC_{01} + WC_{10} - WC_{11}$$

-- Equation 36

Notice that the order of the four equations has been changed to be YC_{00} , YC_{10} , YC_{11} , and YC_{01} . In this order, these four equations of Equation 36 appear to be the similar as the 1-D Hadamard transform of Equation 35 in section 5.1.1. In fact, the actual operations are exactly the same. So it can make use of the 4x4 integer transform and Hadamard transform multiplexed block to do the chroma DC transform.

5.1.5 Inverse luma (and chroma) DC quantization

The inverse luma DC quantization Equation 15 is shown again below:

$$\text{If } QP \geq 12, WDC_{ij} = YDC_{ij} * MR_{00} * 2^{\text{floor}(QP/6)-2}$$

$$\text{If } QP < 12, WDC_{ij} = [YDC_{ij} * MR_{00} + 2^{1-\text{floor}(QP/6)}] >> (2 - \text{floor}(QP/6))$$

It is already known that one DC quantized element should be inserted into each 4x4 AC block data. Since one set of four input elements is latched into the transform block every two clock cycles, eight clock cycles are needed for the 4x4 AC data. This means that eight clock cycles are allocated for the inverse DC quantization to output one DC element. Hence one multiplier is enough to do the inverse luma DC quantization.

Inverse chroma DC quantization has the similar operation as compared to inverse luma DC quantization. They both require the hardware resources like the multiplier, which can be shared.

5.1.6 Functional simulation for inverse Q and T block

After the low level circuit and micro functional blocks for inverse Q and T have been designed, VHDL code is written to describe the hardware. Then functional simulation is carried out to check whether the logic of the circuit is correct.

H.264 has a series of reference software versions written in C to test the general encoder and decoder models. A few video sequences are used for testing in the reference softwares. The video sequence `foreman_qcif.yuv` contains hundreds of frames of 176x144 resolution.

The functional simulation is set up as below:

- Reference software version JM7.5C is used to encoder and decoder video sequence `foreman_qcif.yuv`.
- The test data input and output of the inverse Q and T is read out from the corresponding data variables of version JM7.5C and written to test input and test output files.
- ModelSim is then to read the input from test input file and compares the simulated output with the data in test output file.

There are hundreds of frames in the foreman_qcif.yuv video sequence, so it is enough for the functional simulation. The same simulation procedures are performed for the rest of functional blocks, which will not be mentioned again in their specific functional simulation sections. The waveforms in this report are mainly to let the readers to have an idea about the input and output timing relationship.

The simulation results of the inverse Q and T block are exactly the same as the reference software test data output since it is an integer operation. So the logic function of the circuit is correct. Figure 34 shows a sample graphic interface of inverse Q and T block from the ModelSim simulation tool. It shows the signal names with their waveforms. All the signal values are shown in decimals.

Figure 34. Functional simulation for inverse Q and T block

The signal “prediction_mode”=1 tells the inverse Q and T block that the residual data is either inter predicted or luma non-16x16 intra predicted residual data. This means the inverse Q and T block will select the flow chart in Figure 12 instead of Figure 13 to process the data for the luma data. The quantization value “qp” is 16. The inputs are

x0~x3. They are the real test input data from the reference software JM75.C and latched in once every two clock cycles, which follows the design in section 5.1.2.

The input data in text form is 4x4 data shown below:

$$\begin{bmatrix} 61 & 3 & 3 & 0 \\ -12 & -3 & -1 & 0 \\ -11 & 0 & 1 & 0 \\ -5 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 8 & 7 & 3 & 0 \\ 2 & 0 & -2 & 0 \\ -1 & 1 & 0 & 0 \\ 2 & 0 & -1 & 0 \end{bmatrix} \dots\dots$$

The outputs are y0~y3. There is a signal “oe” to indicate when the output is ready. In the previous standards, where there are floating point operations, the hardware simulation results can never be exactly the same as the software simulation results.

The outputs are shown below:

$$\begin{bmatrix} 34 & 72 & 83 & 81 \\ 28 & 71 & 72 & 67 \\ 28 & 69 & 66 & 59 \\ 36 & 69 & 71 & 64 \end{bmatrix} \begin{bmatrix} 20 & 18 & 20 & 19 \\ 15 & 8 & 11 & 2 \\ 5 & 1 & 3 & -8 \\ 0 & 3 & 5 & -1 \end{bmatrix} \dots\dots$$

5.2 Reorder

The input to the reorder block during decoding is from the CAVLC block. The input sequence of these data is: total_coeff, level, total_zeros and then run_before. There is another input signal in parallel with these input data to the reorder block, which is “input_type” to tell us which type of data it is. Data total_coeff and total_zeros are constants for a block; however, there are more than one level and run_before for a block. The output produced is the scanned array for the 4x4 or 2x2 block.

Figure 35. Hardware architecture of reorder

Figure 35 shows the hardware architecture of the reorder block. The input signal, level, and output signals are of 16 bits. However, run_before is the number of zeros preceding the level. The maximum number of zeros should not exceed 15 for a 4x4 block, so it is of 4 bits. There are two registers to hold total_zeros and total_coeff since they will be used by the controller. Multiplexer “1” will select either level or zero for actual coefficients output.

There are two LIFO (Last In First Out) registers to store the level and run_before. The LIFO registers are required, since the scanned array is coded in reversed order. This means the first inputted level should be the last non-zero coefficient and last inputted level should be the first non-zero coefficient in the array after reordering. Based on the total_coeff, total_zeros and run_before, the controller controls the multiplexer labeled

as “1”. If the run_before is not zero, the controller sets the multiplexer to output zero and counts the number of zeros to being outputted. If the run_before is zero, then the controller latches out the level as output.

5.2.1 Design of level and run_before LIFO registers

There are level and run_before blocks in Figure 35 circuit. The detailed circuit for the level block is shown in Figure 36.

Figure 36. Level block circuit (a) Common LIFO registers
(b) Modified LIFO registers

Figure 36 (a) shows a common architecture for the level LIFO registers. There will be two signals controlled by the state machine, i.e. latch0 and c4. Signal c4 controls the multiplexers in Figure 36 (a). Signal latch0 controls whether to latch the registers n0~n15 during the rising edge of the clock. Since the maximum number of level is sixteen, it is designed using sixteen registers from m0 to m15 to hold the input data. The operation will be: If signal “input_type” indicates the input data is level, then the registers m0~m15 start to latch in the input. The number of level is unknown, so all the possible numbers of level should be considered. When it is required to latch out the first level, signal c4 will control the multiplexers to select data from m0~m15 as input to n0~n15 and latch0 will control the registers n0~n15 to latch. So the registers n0~n15 will hold the data from m0~m15 automatically. After this, signal c4 will control the multiplexers to select data from n0~n15 to latch out the level.

Figure 36 (b) shows a modified architecture for the level LIFO registers. This intends to save almost half of the registers. When it is latching in the level, the circuit in (b) operates in the same way as that in (a). However, when latching out the level, it will use the same series of registers to hold the level. The same signals latch0 and c4 are used to control the operations.

The run_before block structure is similar to the level block. One difference is that the run_before block uses one register less than level block to hold the input since the maximum number of run_before input is fifteen. Another difference is that the register size of run_before is 4-bits register since the maximum number of non-zero coefficients is fifteen.

5.2.2 Reorder state diagram and operations

The state diagram is shown in Figure 37. It includes all the possible situations when reordering a block of data during decoding. Many control signals are not shown due to the limitation of space.

First the signal “input_type” indicates the different data types. And input_type=“001” means the input data type is total_coeff, which also means a valid block data is started to be latched in.

State S0 is a reset state doing nothing. At state S1, the circuit keeps latching in and checking the data. If it is a valid data, i.e. input_type=“001”, which means the data stands for total_coeff, then the next state is S2. If it is not valid data, i.e. input_type != “001”, it will remain in state S1.

At state S2, it will check total_coeff value:

- If total_coeff = 0, that means this block’s coefficients are all zeros.
 - If nc = -1, it is a 2x2 block. Four zeros will be outputted in state S3.
 - If nc != -1, it is a 4x4 block. Sixteen zeros will be outputted in state S4.
- If total_coeff != 0, this means there are some non-zero coefficients, i.e. level. So it will go to state S5 to latch in the levels values. At the same time, the counter in Figure 37 will store the number of the levels being latched in at this state.

Figure 37. Reorder state diagram

When level is latched in during state S5, signal counter is incremented to store the number of level latched in. At the same time, it will check total_coeff value again:

- If total_coeff=16, this means the 4x4 block contains sixteen non-zero coefficients, so it will output all the coefficients directly during state S6.
- If nc = -1 && total_coeff = 4, this means the 2x2 block contains four non-zero coefficients, so it will output the four coefficients during state S7.
- For other cases, which means the block contains some zero coefficients, the next state will be S8.

At state S8, it latches in total_zeros first. The value of total_zeros stands for the number of zeros before the last level in the scanned array. It does not include those zeros located after the last level.

- If total_zeros=0, it means this block does not have any zeros before the last level in the scanned array without any zeros in between. The next state is S9 to output all the level. After this:
 - If nc = -1, it is a 2x2 block. Then the circuit will output the remaining zero coefficients until there are four coefficients being outputted during state S10.
 - If nc != -1, it is a 4x4 block. Then the circuit will output the remaining zero coefficients until there are sixteen coefficients being outputted during state S11.
- If total_zeros !=0, that means there are some zero coefficients in between the level. Then the circuit will latch in the run_before during state S12.

Generally, every level has a run_before. For example, in scanned array (1) “0 0 0 A₁ xxx..”, every level including A₁ has a run_before from the CAVLC block. The character like A₁ stands for level and xxx for don't care coefficients.

However, there are a few exceptions for this run_before. Arrays (2) “A₁ 0 0 A₂ xxx..”; (3) “A₁ A₂ A₃ 0 0 A₄ xxx..” are used to explain the exceptions. In array (2), A₁ does not have a run_before and in array (3) A₁, A₂ and A₃ do not have run_befores from the CAVLC block. In order to cater for the exceptions, counter3 will be assigned with the

number of level latched in state S8. During state S12, for each run_before latched in, counter3 is decremented.

Signal “input_type” tells the reorder block what data type the input is. So during state S12, the following will be checked:

- If input_type=“101”, that means there is run_before data available to be latched in. So the next state is S12 again.
- Else, that means there is no more run_before.
 - If counter3 = 0, that means each level has a run_before from the CAVLC block. This is situation when the scanned array is (1) “0 0 0 A₁ xxx..”. So the next state is S13 to output zero.
 - Elself counter3 = 1, that means one run_before is not assigned to the first level. This is the situation when the scanned array is (2) “A₁ 0 0 A₂ xxx..”. So the next state is S14 to latch out coefficient A₁.
 - Else, that means a few run_before are not assigned to a few level. This is the situation when the scanned array is (3) “A₁ A₂ A₃ 0 0 A₄ xxx..”. So the next state is S14 to latch output coefficient A₃.

During state S13, it will output the zero coefficients until counter1 = run_before. Then it will go to state S14.

During state S14, it will output the “level”, and then it will check the value of the counter signal. The counter stores the number of levels for this block. For every level latched out in S14, counter will be decremented. At the same time, it will also reduce counter3 value for each level latched out during state S14.

- If counter = 0, that means all the level are outputted, the circuit should check whether there are any remaining zero coefficients after the level.
 - If (nc=-1 && counter2=4)|| (counter2=16), this means there are four coefficients outputted for a 2x2 block or sixteen coefficients outputted for a 4x4 block. Then this block has finished reordering.
 - Else
 - If nc=-1, the next state is S15. It will output the remaining zero coefficients until counter2=4.

- If nc!=1, the next state is S16. It will output the remaining zero coefficients until counter2=16.
- Else, that means there are some levels left. Then it will check counter3 value first. This is to cater for those exceptions. Generally, in state S14, it will check the value of run_before.
 - If run_before = 0, this means that there is no zero coefficient before the next level. Then it will remain in S14 to output the next level.
 - If run_before != 0, there are zero coefficients before the next level. So the next state will be S13 again to output the zero coefficients.

5.2.3 Functional simulation for reorder block

Figure 38 shows the functional simulation graphic interface for the reorder block. The signal nc=1 tells the circuit that it is processing a 4x4 block. The signal x0 is the actual input data from the CAVLC block. The signal input_type tells the circuit the data type of the input. For example, input_type="001" indicates the input is total_coeff, which is 5 in Figure 38. Those input_type="111" data are garbage and are automatically rejected by the buffer before the reorder block.

When current_state=13, it latches out 0. When current_state=14, it latches out level. The scanned array output is at y0. Signal counter2 counts the number of coefficients latched out. So from Figure 38, the scanned array output is 0, 3, 0, 1, -1, -1, 0, 1, 0....

Figure 38. Functional simulation for reorder block

5.3 Context Adaptive Variable Length Coding

5.3.1 Introduction

Because the codeword lengths are variable, the codeword boundary in a bitstream cannot be detected until the decoding procedures of previous codeword is completed [21]. This limits the processing speed of the VLC (Variable Length Coding) decoder. This makes the VLC decoding hardware computationally intensive, memory intensive or both.

The VLC hardware designs generally fall into two categories:

- (1) Latch in the input bitstream bit by bit and then generate the output. This type of hardware needs a variable number of clock cycles to get a valid codeword being decoded; hence, it tends to be slow, especially when the codeword is long. Moreover the decoded symbol output rate will not be constant.
- (2) Clock in the input codeword by codeword. This type of hardware clocks in the codeword of variable number of bits, and generates the decoded symbol at a constant rate [22].

In this project, the second method is employed for this Context Adaptive Variable Length Coding (CAVLC) block design. According to the algorithm for the decoding sequence, first `total_coeff` and `num_trailingones` are decoded. Then the coefficient level is decoded. After this, `total_zeros` is decoded and finally the `run_before` is decoded. This finishes a complete 4x4 (or 2x2 Chroma DC) block.

According to [2] and [15], and the maximum codeword length is 16 bits except for level codeword. The maximum level codeword length is 28 bits, including a maximum 16 bits `level_prefix` and maximum 12 bits `level_suffix`. In the level codeword table, like Table 5, “00..01” is `level_prefix` and “xx...xs” is `level_suffix`.

5.3.2 Architecture of CAVLC

In this design, the level is decoded in 2 clock cycles, one for the `level_prefix` decoding and the other for the `level_suffix` decoding and level computation. All the other information like `total_coeff` is decoded in one clock cycle. In this way, it is sure that

the maximum bit length to be processed in each clock cycle is not more than 16 bits. This explains why the barrel shifter output is 16 bits in the next paragraph. The hardware architecture is shown in Figure 39.

Figure 39. Architecture of CAVLC

The hardware structure has two 16-bit registers to hold the data from the input buffer. The barrel shifter takes 32 bits from the registers as input. The control signal “shifter” tells the barrel shifter the starting position to output from the 32 bits input. This 16 bits output of barrel shifter serves as an input to the different decoding components. The outputs of these components consist of actual data to the output buffer and its codeword length.

The codeword length is accumulated to signal the barrel shifter the new starting position of the 32 bits input. When the accumulator is more than 15, it sends a request to the input buffer to latch in 16 bits of new data. Signal “shifter” value is updated to the correct starting position from the 32 bits input.

The `total_coeff` and `num_trailingones` codeword table, `level` codeword table, `total_zeros` codeword table and `run_before` codeword table are implemented using look-up tables. Signal “nc” is required by `total_coeff` and `num_trailingones` codeword table as defined in section 3.4.2 on page 30. The trailing ones decoding block outputs a value of 1 for input bit 0 and -1 for input bit 1.

In Figure 39, the `level_prefix` decoding component outputs `leading_zero` and the length of `level_suffix`. Then with `level_suffix`, `leading_zero` and `vlcnum`, the level computation component can calculate the actual value of the level.

Level computation

In section 3.4.2, the standard document [2] provides the algorithm below:

```
x = leading_zero << (vlcnum+1) + level_suffix
If x[0] = '0',
    level = ((x+2) >> 1)
Else x[0] = '1',
    level = ((-x-1) >> 1).
```

The operations seem to be more standardized. In the final result for level, it has a right binary shift 1 bit to get the correct result. So in the proposed algorithm for level computation, this right binary shift 1 bit is eliminated by modifying the algorithm. The proposed level computation method is shown below.

When `vlcnum=0`,

If `leading_zero < 14`: Value of level is shown in the Table 4 on page 32.

If `leading_zero=14`: There are 4 xxxx named as `level_suffix` in the codeword. It is represented as `m1(3 downto 0)`. The last bit `m1(0)` is the sign bit (0 for positive level and 1 for negative level). `Absolute(level) = m1(3 downto 1) + 8`

If `leading_zero=15`: There are 12 xx...xxx named as `level_suffix` in the codeword. It is represent it as `m2(11 downto 0)`. The last bit `m2(0)` is the sign bit (0 for positive level and 1 for negative level). `Absolute(level)=m2(11 downto 1)+16`.

When `vlcnum!=0`,

If `leading_zero < 15`: There are `(vlcnum-1)` bits `level_suffix` `x...x`, followed by another explicit sign bit `s`. If the value of the unsigned binary number of `x...x` is

represented by $m3$, then $\text{Absolute}(\text{level}) = (\text{leading_zero}) \ll (\text{vlcnum} - 1) + 1 + m3$.

If $\text{leading_zero} = 15$: There are 11 bits $\text{level_suffix } x \dots x$, followed by another explicit sign bit s .

If the value of the unsigned binary number of $x \dots x$ is represented by $m4$, then $\text{Absolute}(\text{level}) = (15 \ll (\text{vlcnum} - 1)) + 1 + m4$.

In this way, the operation bit width is reduced by one bit. This could shorten the delay and have smaller hardware. As will be shown in the synthesis results, the CAVLC block critical path delay is mainly caused by this level computation component.

5.3.3 CAVLC block state diagram

The state diagram is shown in Figure 40. During state S1 and S2, the input data is clocked into the two 16-bits registers from the input buffer.

In state S3, the circuit starts to decode the total_coeff and num_trailingones . Counter1 and counter2 are initialized to 0. Counter1 stores the number of non-zero coefficients decoded. Counter2 is to store the number of trailing ones decoded. If the decoded $\text{total_coeff} = 0$, it means that there is no non-zero coefficient. The decoding of the CAVLC bitstream for this block is finished. If $\text{total_coeff} \neq 0$, then it checks the num_trailingones . If $\text{num_trailingones} = 0$, that means there is no trailing ones. Then the next state will be S5 for level_prefix decoding. Or else it will go to S4.

During state S4, it decodes the trailing ones. Counter1 and counter2 are incremented for each trailing one decoded. When $\text{counter2} = \text{num_trailingones}$, which means the trailing ones have all been decoded, it checks whether $\text{counter1} = \text{total_coeff}$. If $\text{counter1} = \text{total_coeff}$, that means in this CAVLC coded block, there is no other non-zero coefficient except the trailing ones. So it is needless to decode the level and the next state will be S7 for decoding total_zeros . If $\text{counter1} < \text{total_coeff}$, that means there are other non-zero coefficients besides trailing ones. So the next state will be S5 for level_prefix decoding.

During S5, it decodes the level_prefix information. Basically it produces the value of leading_zero and length of level_suffix to be used in state S6 for level computation. The detailed decoding process is in the algorithm section.

Figure 40. State diagram for CAVLC architecture

The next state is S6 for level_suffix decoding and level computation. The value for vlcnum is updated depending on the just decoded level. Counter1 is incremented for each level decoded. When counter1 = total_coeff, this implies that all the non-zero coefficients including trailing ones and level have been decoded. Then it checks whether total_coeff = maxNumCoeff. If total_coeff = maxNumCoeff, this means all the coefficients for this CAVLC block is non-zero coefficients. If there are no zero coefficients, the CAVLC bitstream for this block is then finished. Otherwise, there are some zero coefficients, so the next state will be S7.

During state S7, total_zeros is decoded. The counter1 resets to 2. Counter1 stores the number of run_before decoded in state S8. The value of total_zeros refers to the number of zero coefficients before the last non-zero coefficients in the zig-zag scanned array. If total_zeros = 0, which means the zero coefficients are all located after the last non-zero coefficients in the array, so the decoding of the CAVLC bitstream for this block is finished. Otherwise, if total_zeros != 0, which means there are zero coefficients located before the last non-zero coefficients, then the next state will be S8.

During state S8, run_before is decoded. The value of run_before is the number of zero coefficients before each non-zero coefficient. Each time run_before is decoded, zeroleft will be calculated and latched into R_zeroleft during next clock cycle.

For each clock cycle in S8, counter1 is incremented. It stores the number of run_before decoded plus 1 because it initializes counter1=2 instead of 1 in state S7. The run_before of the first level is not coded by the CAVLC algorithm of H.264, so the maximum number of run_before will be total_coeff-1. When counter1=total_coeff, which means that the number of run_before decoded equals to total_coeff - 1, there is no more run_before left to be decoded.

During state S8:

- If R_zeroleft = run_before just decoded, this means that there are no more zeros left. So the decoding of this block of data is finished. The next state will be either S0 or S3 depending on the condition of I_start.

- If $R_zeroleft > run_before$ just decoded, this means there are more some zeros left in this scanned array.
 - If $counter1 = total_coeff$, that means those zeros are at the starting position of the scanned array. They are not coded in run_before by CAVLC algorithm in H.264. The next state is S9.
 - If $counter1 < total_coeff$, that means there are still run_before , then $next_state$ will be S8 again to do decoding.

During state S9, register $R_zeroleft$ is holding the number of zeros at the starting position of the scanned array. So $R_zeroleft$ will be the output. The next state will be either S0 or S3 depending on the condition of I_start .

Generally a level (non-zero coefficient) is assigned a run_before by CAVLC process in H.264 standard. There are a few exceptions as discussed on page 66 of section 5.2.2. For example, the number of starting continuous zeros in front of the first level is not coded by CAVLC process in H.264 algorithm. However, the CAVLC block design may output this run_before . The three scanned arrays on page 66 are used again to explain states S8 and S9 clearer for the CAVLC block design.

The three arrays are: (1). "0 0 0 A_1 xxx..". (2). " A_1 0 0 A_2 xxx..". (3). " A_1 A_2 A_3 0 0 A_4 xxx..". And A_1 , A_2 represent the level and x for don't care coefficients. In the array (1), the starting three zeros are not coded as $run_before = 3$ by the CAVLC algorithm in H.264 since this value could be calculated as $zeroleft$. However, in the design of this CAVLC circuit, after it gets the value of $zeroleft$, it will still output this value as run_before in state S9, which is to be used in the reorder block.

In the array (2), the run_before for A_1 is not coded by the CAVLC algorithm of H.264. The reason is the same as for array (1). A_1 is the first non-zero coefficient. In the array (3), the run_before for A_1 , A_2 , and A_3 is not coded by the CAVLC algorithm of H.264. The reason is that $zeroleft=0$ after run_before for A_4 is decoded.

In the design of CAVLC circuit, it will not calculate the run_before of these levels as well. After it finishes decoding the run_before of A_2 in array (2) or the run_before of A_4 in array (3), the circuit will start to do CAVLC decoding for another block.

In the following paragraphs, the example in the algorithm section 3.4.2 is used to illustrate specifically what is performed in each state. The coded bitstream for this block is {0000 1000 11 10 0101 1110 1101}. This will be in the buffer at state S0. When state S1 comes, the first 16 bits (0000 1000 1110 0101) will be latched into the register. When state S2 comes, another 16 bits (1110 1101 xyzu vwmn) will be latched into the register. Here xy..mn is the bitstream of the next CAVLC block or other information. The signal “shifter” is initialized to be 0.

When S3 comes, the barrel shifter output is (0000 1000 1110 0101). Counter1 and counter2 are initialized to 0. Depending on the input “nc” value, the circuit will select the codeword table for total_coeff and num_trailingones in [2]. In this example, it is known that $0 \leq nc < 2$, so if it compares those codewords in the table in [2] with the input 16 bits, it can find one unique match. That is num_trailingones = 3 and total_coeff = 5, whose codeword is “0000100”. Its codeword length is 7. The signal “shifter” takes this value and sets the new starting position at the 7th bit of its 32 bits input when the next state comes.

Accordingly, the next state will be S4 to decode the trailing ones. The barrel shifter 16 bits output is changed to (0111 0010 1111 0110). The first bit is 0, so the output will be +1. Then counter1 and counter2 are updated to 1. The codeword length is 1. So the signal “shifter” value will be 7+1=8.

Next state will still be S4 since there are 3 trailing ones. The barrel shifter 16 bits output is changed to (1110 0101 1110 1101). The first bit is 1 so the output will be -1. counter1 = counter2 = 2. The signal “shifter” is accumulated to be 9.

Another clock cycle in state S4 is required to decode the last trailing ones. The barrel shifter 16 bits output is (1100 1011 1101 101x). The first bit is 1 so the output is -1. Then counter1 = counter2 = 3. The next state is S5 since counter2 = num_trailingones = 3. The signal “shifter” is updated to be 10.

At state S5, level_prefix decoding starts. The barrel shifter 16 bits output is now (1001 0111 1011 01xy). The value for vlcnum is initialized to 0 since condition

(total_coeff>10 && num_trailingones<3) could not be satisfied. Accordingly, Table 4 on page 32, which is level codeword table for vlnum=0 will be chosen for decoding. It can find the first entry with codeword “1” is matched. Then the output leading_zero equals to zero. The decoded bit length is 1, so the signal “shifter” is 11 now.

At state S6, barrel shifter 16 bits output will be (0010 1111 0110 1xyz). Since for vlnum=0, when leading_zero<14, the length of level_suffix code is zero. So no level_suffix decoding is required and the decoded level will be 1. The signal “shifter” remains 11. Counter1 = 3+1 = 4. Counter1<total_coeff, so the next state will be S5 again. The value for vlnum is updated to 1.

At S5, the barrel shifter 16 bits output is still (0010 1111 0110 1xyz). The level codeword table for vlnum=1 in [15] is chosen. In the bitstream, there is “001” matched to the codeword “001s” in the level codeword table. So leading_zero equals to 2. Level_prefix “001” is decoded and level_suffix decoding “s” is left to be decoded in S6. The decoded bit length is 3, so signal “shifter” is 14.

At S6, the barrel shifter 16 bits output is (0111 1011 01xy zuvw). The first bit is 0 for level_suffix. The leading_zero decoded in S5 is 2, so the actual level could be computed to be 3. The decoded bit length is 1, so the signal “shifter” is 15. Counter1 = 4+1=5. Since counter1 = total_coeff, next state will be S7.

During S7, counter1 resets to 2 for run_before decoding in S8. The barrel shifter 16 bits output is (1111 0110 1xyz uvwm). Using total_coeff = 5 to index the total_zeros codeword table in [2], it can find a match of codeword “111” with total_zeros=3. The value zeroleft is initialized to 3 as well. And this will be latched into R_zeroleft during first clock cycle of state S8. The codeword length is 3. So the signal “shifter” is accumulated to 18.

At this moment, if it does not do anything, the barrel shifter output will be (1011 01xy zuvw mn). There are only 14 bits left, which may not be enough for decoding those symbols with codeword length more than 14. In fact this situation will happen whenever “shifter” is bigger than 16. So what it does here is whenever “shifter” is

accumulated more than 15, it will require the input buffer to latch in 16 bits of data. At the same time “shifter” is deducted by 16. In the example, after deduction, “shifter” will be 2 in the next state.

So when S8 comes, the new 32 bits input of barrel shifter will be (1110 1101 xyzu vwmn dddd dddd dddd dddd). Those dd..d is the new input from the buffer. Since “shifter”=2, the output of the barrel shifter will be (1011 01xy zuvw mndd). $R_zeroleft = 3$ is used to index the run_before codeword table in [2]. The matched codeword will be “10” and run_before = 1. Then $R_zeroleft$ is then updated by $R_zeroleft - run_before = 3 - 1 = 2$ during the next clock cycle. The codeword length is 2, so “shifter” is updated to 4. The next state is still S8 and counter1 will be updated to 3. During this state counter1 = 3 and barrel shifter 16 bits output is (1101 xyzu vwmn dddd). $R_zeroleft = 2$, so the matched codeword is “1” and run_before = 0. So the updated $R_zeroleft$ remains to be 2 in the next clock cycle. The codeword length is 1, so “shifter” is now 5. The next state is still S8 and counter1 will be updated to 4.

In this state, counter1 = 4 and barrel shifter 16 bits output is (101x yzuv wmn dddd). $R_zeroleft = 2$, and the matched codeword is “1” and run_before = 0. So $R_zeroleft$ remains to be 2 during the next clock cycle. The codeword length is 1, so “shifter” is now 6. The next state is still S8 and counter1 will be updated to 5.

During this state counter1 = 5 and barrel shifter 16 bits output is (01xy zuvw mn dd dddd). $R_zeroleft = 2$, and the matched codeword is “01” with run_before = 1. The codeword length is 2, so the “shifter” is now 8. Since counter1 = total_coeff, it will compare $R_zeroleft$ and run_before. If $R_zeroleft > run_before$, this means there are zeros in the starting position of the scanned array. The next state will be S9 and $R_zeroleft$ will be updated to $2 - 1 = 1$.

During state S9, the number of zero at the starting position of the scanned array is latched out. The value is 1.

These decoded output will go for some further processing in the reorder block, and final output is (0, 3, 0, 1, -1, -1, 0, 1, 0, ...).

5.3.4 Functional simulation for CAVLC block

The functional simulation result is shown in Figure 41. The signal x0 is the input from the input buffer. The first two 16 bits are shown below Figure 41. This is the same as the example explained in section 5.3.3. When the circuit starts to decode, the signal “request_input_buffer” is set to 1 for two clock cycles in order to latch in the two 16 bits of input first.

The signal y0 is the output and “out_type” indicates the data type of y0, i.e. total_coeff, level, total_zeros and run_before. The signal “out_type”=“111” indicates that y0 is garbage data at this clock cycle. These outputs are the inputs of the reorder block. The data when “out_type”=“111” will automatically be rejected by the buffer after the CAVLC block.

5.4 Intra prediction

As mentioned in section 3.5, for luma data, if it is luma 4x4 intra prediction, there are nine prediction modes and if it is luma 16x16 intra prediction, there are four prediction modes. For the chroma data, there is only chroma 8x8 intra prediction, with four prediction modes. In this section, their hardware designs are shown clearly. During the design, it is found that luma 4x4 intra prediction is largely different from luma 16x16 and chroma 8x8 intra prediction, so it is designed as a separate component. However, the luma 16x16 and chroma 8x8 intra predictions are similar, so they are designed together to share the common hardware resources.

5.4.1 Luma 4x4 intra prediction

Depending on the availableness conditions of the neighboring pixels, A~L values will be assigned and latched into registers m0~m12 as shown in Figure 42. Signal c0 will control the registers m0~m12 to either latch in data (c0="11"), shift right (c0="01"), shift left (c0="10"), or remain unchanged (c0="00").

Figure 42. Registers for neighboring pixels

After latching in the pixels, the four computational elements in Figures 43~46 will operate concurrently to generate one row of four outputs for the luma 4x4 block every clock cycle. Each computational element generates one output. For example, in Figure 43, either g0 or tt0 is selected as one of the output in a row. Output g0 is selected for computation like $mpr(0,0) = (X+A+1)/2$ of vertical_right prediction mode in Equation 23, while tt0 is selected for computation like $mpr(0,0) = (I+2*X+A+2)/4$ of diag_down_right prediction mode in Equation 21.

From the section 3.5.1, it is known that the luma 4x4 intra prediction has many add operations. For the nine luma 4x4 intra prediction modes, their operations are similar. So in the Figures 43~46, they make use of multiplexers to avoid redundant adder resources.

Figure 43. Computational element 0

Figure 44. Computational element 1

Figure 45. Computational element 2

Figure 46. Computational element 3

Signals `ccc0`, `ccc1`, `ccc2`, `ccc3` and `ccc4` are the control signals from the state machine in Figure 48. They are the multiplexer select signals as specified below. For example, `ccc0` is 6 bits. It controls the three multiplexers as shown in Figure 43.

`ccc0` is a 6-bit signal:

```
aa0 <= m1 when (ccc0(5 downto 4)="00") else
      m7 when (ccc0(5 downto 4)="01") else
      m2;

bb0 <= m0 when (ccc0(3 downto 2)="00") else
      m8 when (ccc0(3 downto 2)="01") else
      m1;

cc0 <= m5 when (ccc0(1 downto 0)="00") else
      m9 when (ccc0(1 downto 0)="01") else
      m0;
```

`ccc1` is a 3-bit signal:

```
aa1 <= m0 when (ccc1(2)='0') else
      m8;

bb1 <= m5 when (ccc1(1)='0') else
      m9;

cc1 <= m6 when (ccc1(0)='0') else
      m10;
```

`ccc2` is a 5-bit signal:

```
aa2 <= m5 when (ccc2(4)='0') else
      m4;

bb2x<= m6 when (ccc2(3)='0') else
      m3;

bb2y<= m6 when (ccc2(2 downto 1)="00") else
      m3 when (ccc2(2 downto 1)="01") else
      m1;

cc2 <= m7 when (ccc2(0)='0') else
      m2;
```

`ccc3` is 5-bit signal:

```
aa3 <= m6 when (ccc3(4)='0') else
      m3;

bb3x<= m5 when (ccc3(3 downto 2)="00") else
      m7 when (ccc3(3 downto 2)="01") else
      m2;

bb3y<= m7 when (ccc3(1)='0') else
      m2;

cc3 <= m8 when (ccc3(0)='0') else
      m1;
```

`ccc4` signal is 1 bit:

It will select the left input when `ccc4 = '1'`, and right input when `ccc4 = '0'`

The circuit in Figure 47 is for luma 4x4 DC prediction. There are four possible outputs depending on the neighboring blocks availability.

Figure 47. Circuit for luma 4x4 DC prediction output

Figure 48 shows state diagram for luma 4x4 intra prediction. In the following paragraphs, the diag_down_right mode is used to explain the hardware operation.

Diag_down_right prediction mode

The series of equations labeled as Equation 21 in section 3.5.1 are shown again below:

$$\begin{aligned} \text{mpr}(0,0) &= (I + 2 \cdot X + A + 2) / 4; \\ \text{mpr}(1,0) &= (X + 2 \cdot A + B + 2) / 4; \\ \text{mpr}(2,0) &= (A + 2 \cdot B + C + 2) / 4; \\ \text{mpr}(3,0) &= (B + 2 \cdot C + D + 2) / 4; \\ &\dots \\ \text{mpr}(0,3) &= (L + 2 \cdot K + J + 2) / 4; \\ \text{mpr}(1,3) &= (K + 2 \cdot J + I + 2) / 4; \\ \text{mpr}(2,3) &= (J + 2 \cdot I + X + 2) / 4; \\ \text{mpr}(3,3) &= (I + 2 \cdot X + A + 2) / 4; \end{aligned}$$

The expression $\text{mpr}(0,0)$, $\text{mpr}(1,0)$, $\text{mpr}(2,0)$ and $\text{mpr}(3,0)$ are the first row of the 4x4 block. The four computational elements of Figures 43~46 will operate and generate these four data. So in the state S1, control signal $\text{ccc}0 \sim \text{ccc}3$ are set as shown in Table 8 to select the required input.

For example, for $\text{mpr}(0,0)$, expression $(I+2 \cdot X+A)$ is realized in the computational element by three adders $(I+X)+(X+A)$ instead of two adders and a shifter $(I+A)+2 \cdot X$. It seems that it is a waste of hardware. However, the three adders structure can be reused for other luma 4x4 prediction modes. For example, in vertical_right prediction mode, operation $(X+A)$ is required. That is why in the design, the three adders structure is used.

At the state S2 of Figure 48, it could be observed that second row of four outputs $mpr(0,1)$, $mpr(1,1)$, $mpr(2,1)$ and $mpr(3,1)$ can be achieved by just changing the input pixels and the actual computation is exactly the same as the first row of outputs. It is observed that by shifting data in Figure 42 registers towards right, the required four outputs could be achieved from the four computation elements. So $c0$ is set to “01”, and multiplexers $ccc0\sim ccc3$ do not to change. The same thing applies to another two rows of four outputs of the luma 4x4 block.

In Table 8, control signals like $ccc0$ or $c0$ are set as shown for proper operation of `diag_down_right` mode during states S1~S4. The signals $Y0\sim Y1$ are the luma 4x4 intra predicted outputs.

	State S1	State S2	State S3	State S4
Y0	g0	g0	g0	g0
Y1	g1	g1	g1	g1
Y2	g2	g2	g2	g2
Y3	g3	g3	g3	g3
ccc0	000000	000000	000000	000000
ccc1	000	000	000	000
ccc2	00000	00000	00000	00000
ccc3	00100	00100	00100	00100
ccc4	Not care	Not care	Not care	Not care
c0	01	01	01	11

Table 8. Operations for `diag_down_right` prediction mode

The above paragraphs only explain the `diag_down_right` prediction mode. In fact, the circuits in Figures 43~46 can perform all the luma 4x4 prediction modes when the control signals are set correctly.

In Figures 43~46, there are simple circuits on the right hand side. This is to meet the operations in the prediction modes that have operation like $(X + A + 1) / 2$.

Figure 45 and 46 circuits are a bit different from Figure 43 and 44 circuits. This is because Figure 45 and 46 circuits also compute the luma 4x4 DC prediction. In Figure 47, the circuit takes the signals $f2$, $f3$, $g2$, and $g3$ from Figure 45 and 46 to do luma 4x4 DC prediction. There are four possible outputs (dcy , $g2$, $g3$ and 128) for DC prediction as shown in Figure 47.

State diagram for luma 4x4 intra prediction

Figure 48 shows the state diagram. At a given state, the different luma 4x4 prediction modes will output different control signals. At state S1, the neighboring pixels are latched into the registers, and the four computational elements start to do computations to generate the one row of output. At state S2, the first row output is latched out, and at the same time, the four computational elements start to do computations for the second row of output.

Figure 48. State diagram for luma 4x4 intra prediction

There are sixteen 4x4 luma blocks in one macroblock. States S1~S4 are for the first 4x4 luma block. Another fourteen 4x4 luma blocks should be manipulated repeatedly

as shown in the state diagram. States S9~S12 are allocated for the sixteenth 4x4 luma block. State S13 is to latch the last row of luma output data. Then the next state will be S29, in which states it starts to do chroma 8x8 intra prediction.

5.4.2 Luma 16x16 and chroma 8x8 intra prediction

In the four luma 16x16 intra prediction modes, the DC prediction, the vertical and the horizontal prediction mode can be designed easily. The difficult part is the Plane prediction mode. First the input neighboring pixels are latched into the input registers accordingly as shown in Figure 49. As shown, there are 33 registers to hold all the neighboring pixels for a macroblock. If it is chroma component, then the neighboring pixels will be latched into registers mx, mh0~mh7 and mv0~mv7.

Figure 49. Registers to hold the input neighboring pixels

Luma 16x16 and chroma 8x8 Vertical and Horizontal intra prediction

As mentioned in section 3.2.1, the decoded 4x4 residual data block for a macroblock will be generated with the sequence from index 0 to 15 for the luma component, then 18 to 25 for the chroma component as labeled in Figure 11. This requires the intra predicted output should also follow this sequence of 4x4 block. So in Figure 49, the registers mx, mh8~mh15, and mv8~mv15 will always latch in the input only. Signals ccc20a and ccc20b from the state machine in Figure 54 control the operation of registers mh0~mh7 and mv0~mv7.

Signal ccc20a controls the operation of registers mh0~mh7 as followed:

```
if ccc20a="00" then
    mh0~mh7 <= Input;
```

```

elsif ccc20a="01" then
    mh0~mh7 <= mh8~mh15; /*i.e. mh0<=mh8, mh1<=mh9, ...
elsif ccc20a="10" then
    mh0~mh3 <= mh4~mh7; /*i.e. mh0<=mh4, mh1<=mh5, ...
    mh4~mh7 unchanged
end if;

```

Signal ccc20b controls the operation of registers mv0~mv7 as followed:

```

if ccc20b="00" then
    mv0~mv7 <= Input;
elsif ccc20b="01" then
    mv0~mv7 <= mv8~mv15; /*i.e. mv0<=mv8, mv1<=mv9, ...
elsif ccc20b="10" then
    mv0~mv3 <= mv4~mv7; /*i.e. mv0<=mv4, mv1<=mv5, ...
    mv4~mv7 unchanged
else
    /*The registers mv0~mv3 latch upward
    mv0<=mv1;
    mv1<=mv2;
    mv2<=mv3;
end if;

```

For example, if it is vertical prediction mode, the output will be four pixels from the neighboring sixteen pixels above the current macroblock, depending on which 4x4 block of Figure 11 is predicted. This could be controlled by ccc20a, so that the output always appears at the registers mh0, mh1, mh2 and mh3.

Luma 16x16 and chroma 8x8 plane prediction mode

Luma 16x16 plane prediction mode almost covers all the operations of the chroma 8x8 plane prediction mode. Based on the circuit of luma 16x16 plane prediction mode, chroma 8x8 plane prediction could be realized by adding a few multiplxers. This will be noticed in the explanation for luma 16x16 plane prediction mode hardware design.

In Equation 33 of section 3.5.2, the computation for H is shown again below:

$$\begin{aligned}
 H &= \sum_{x'=0}^7 (x'+1) * (p(8+x', -1) - p(6-x', -1)) \\
 &= 1 * (p(8, -1) - p(6, -1)) + \\
 &\quad 2 * (p(9, -1) - p(5, -1)) + \\
 &\quad 3 * (p(10, -1) - p(4, -1)) + \dots\dots \\
 &\quad 8 * (p(15, -1) - p(-1, -1))
 \end{aligned}$$

Expression $p(x, -1)$ with $x=0\sim15$ are the neighboring pixels above the current macroblock. This equation has eight multiplications. The multiplier coefficients are 1,

2, 3, 4, 5, 6, 7, and 8. The multiplications of 1, 2, 4, and 8 could be simply realized by binary left shift. The coefficient 3 can be separated into 2+1 and 5 equals to 4+1. Coefficient 7 equals to 8-1. In this manner, the multiplication operations could be realized by binary left shift with some extra additions or subtractions. It is shown in Figure 50. R0~R7 are registers to hold the intermediate values.

Figure 50. The circuit for calculating H and V

In Equation 33, V is generated with the same operation as H, except that $p(x, -1)$ with $x = 0 \sim 15$ is changed to $p(-1, y)$ with $y = 0 \sim 15$ for V. So in Figure 49, a multiplexer controlled by ccc11 is used to share the hardware for computing V and H.

In Equation 33, after calculating H and V, b and c could be found by.

$$b = (5 * H + 32) \gg 6$$

$$c = (5 * V + 32) \gg 6$$

Again, multiplication of 5 could be replaced by 4+1. So it could be realized by some shift and add operations.

In Figure 51, signals a0~a7 are connected to registers R0~R7. The value of H or V will appear at a14 and be latched into register R8 at two consecutive clock cycles. It goes through some extra shift and add operations to get b and c at m41 at different timing. The register R8x is to hold the H or V for the chroma case, since H or V for

chroma case only needs to sum up four expressions. Control signal ccc14 has two selection modes, either for chroma or luma cases.

In fact, the seven adders to sum up a0~a7 also appear in hardware design of luma 4x4 DC prediction modes in the previous section. However, they are not shared. This is because it requires a lot of multiplexers to the adders after the input registers and this may increase the delay of the data path. The design will also be very messy.

Figure 51. Hardware to get the m41 (b or c at different clock cycles)

Then the final predicted data is given in Equation 33 by expression below:

$$p(x, y) = \text{Clip1}((a + b * (x - 7) + c * (y - 7) + 16) \gg 5), \text{ with } x, y = 0..15,$$

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

The first row of the first 4x4 block data:

$$p(0, 0) = \text{Clip1}((a + b * (-7) + c * (-7) + 16) \gg 5)$$

$$p(1, 0) = \text{Clip1}((a + b * (-6) + c * (-7) + 16) \gg 5)$$

$$p(2, 0) = \text{Clip1}((a + b * (-5) + c * (-7) + 16) \gg 5)$$

$$p(3, 0) = \text{Clip1}((a + b * (-4) + c * (-7) + 16) \gg 5)$$

Here, expression $(a+b*(-7)+c*(-7)+16)$ for $p(0, 0)$ could be calculated first. Then by adding b , $2b$, $3b$, the values for $p(1, 0)$, $p(2, 0)$ and $p(3, 0)$ could be computed.

As for the second row of the same 4x4 block data, it could be achieved by adding the first row data with variable c . The same idea applies to the rest of the rows in the same 4x4 block. However, when it starts to output another different 4x4 block, according to the block output sequence, a different value should be added.

For example,

$p(0, 3) = \text{Clip1}((a + b * (-7) + c * (-4) + 16) \gg 5)$, which is the first pixel of the fourth row of the 4x4 block labeled as “0” in Figure 11.

$p(4, 0) = \text{Clip1}((a + b * (-3) + c * (-7) + 16) \gg 5)$, which is the first pixel of the first row of the 4x4 block labeled as “1” in Figure 11.

When it starts to output the $p(4,0)$ of the 4x4 block labeled as “1”, the value added to $p(0,3)$ is $4b-3c$. There are four possible values $(4b-3c)$, $(-4b+c)$, $(4b-7c)$, or $(-12b+c)$ to be added when it starts to output prediction data of another different 4x4 block.

Figure 52 on the next page shows the hardware to calculate the predicted output. Again as mentioned, the signal $ccc14$ controls multiplexers either for chroma case or luma case. Signal $m42$ will have the value $(a+16+b*(-7)+c*(-7))$ for $p(0, 0)$ if it is luma case. Signal $m42$ will be selected by $ccc12$ to be latched into $R16$ for this clock cycle. Registers $R17$, $R18$ and $R19$ will store b , $2b$ and $3b$ respectively. Finally the first row of the first 4x4 block is outputted.

After that, register $R21$ stores value for c , and $R12\sim R14$ stores $4b-3c$, $-4b+c$, $4b-7c$ and $-12b+c$ respectively. Control signals $ccc13$ and $ccc20$ will select one of the values from the five to be added to the previous predicted data. And then $ccc12$ will select the signal $m53$ to be latched into register $R16$ for the other rows of data.

Figure 52. Hardware for the predicted output.

Luma 16x16 and chroma 8x8 DC prediction mode

As for the chroma 16x16 or chroma 8x8 DC prediction mode, there are add operations of eight or four neighboring pixels. By controlling inputs connected to signals a0~a7 in Figure 51 properly, the output a12 and a14 in Figure 51 can be reused.

For chroma 8x8 DC prediction mode, there are either eight or four inputs to be added depending on the availableness of the neighboring pixels. The signal a12 is the sum for four inputs, while a14 is the sum for eight inputs by controlling the inputs connected to a0~a7 properly.

As shown in Figure 53, signal a12 will be divided by 4 (right shift 2 bits) and a14 will be divided by 8 (right shift 4 bits) to get the predicted output. Signal ccc23 selects either of them connected to m76 as the output for chroma 8x8 DC prediction mode.

Figure 53. The data path for the chroma 8x8 DC and luma 16x16 DC prediction

For luma 16x16 DC prediction mode, there are either 32 or 16 neighboring pixel inputs to be added. However signal a14 is the sum for eight inputs. As shown, the sum of eight inputs will be latched into R8y. So in the design, more clock cycles may be

required to add up signal R8y repeatedly. The right side of the circuit in Figure 53 performs this. The operation will be explained in more detailed in section 5.4.3.

5.4.3 Simplified state diagram for intra prediction

Figure 54 on the next page shows the simplified state diagram for the whole intra prediction process. States S1~S12 are allocated to luma 4x4 intra prediction, which is shown in Figure 48. It has already been discussed earlier in section 5.4.1. States start from S14 to S28 are for the luma 16x16 intra prediction mode, including Vertical, Horizontal, DC and Plane prediction modes. From S29 onwards, the chroma 8x8 intra prediction starts to operate.

Since chroma 8x8 intra prediction is similar to luma 16x16 intra prediction, it is not shown. Basically, it differs from luma 16x16 at two points:

- 1). Chroma 8x8 intra prediction has less input pixels to sum up.
- 2). There are two chroma 8x8 block need to be predicted.

State S28 is for vertical or horizontal prediction. For these two prediction modes, no actual computation is performed. It simply latches out the corresponding four neighboring pixels every clock cycle.

States S23~S27 are allocated for DC prediction mode. First, at state S23, signal a14 in Figure 51 will have variable sum1, which is the sum of 8 neighboring pixels.

There are two possible cases depending on the availability conditions of the neighboring pixels.

If it is required to sum up 16 pixels, the next state is S26. At state S26, signal a14 will have variable of sum2, the sum of another 8 pixels. The sum1 is latched into R8y and appears at m72 in Figure 53. At state S27, sum2 is latched into R8y and sum1 is latched into R20. Then these two variables are added. The result will appear at m71, m72 and finally appear at m72x as the output in Figure 53.

If there are 32 input pixels need to be added, it is required to go through states S24 and S25 first, then to S26 to sum up all the inputs as shown in Figure 54.

Figure 54. State diagram of the intra prediction

If it is plane prediction mode, Figure 55 explains the operations corresponding to the different states briefly.

Figure 55. Timing diagram of luma 16x16 Plane prediction

5.4.4 Functional simulation for intra prediction block

Figure 56. Functional simulation for intra prediction block

Figure 56 shows the functional simulation for the intra prediction block. The signal “prediction_mode” = “01101” tells the circuit that it is in a plane prediction mode. The input data is x0~x32. Inputs like x2~x7 are not shown, because they are equal to x1 and x8. This makes sense because they are all neighboring pixels.

The neighboring pixels are all available. They are indicated by the four signals like “block_available_left” =1 at the bottom of Figure 56.

The outputs are y0~y3. They are available at state S19. The outputs are also the same as that in the reference software of H.264. Figure 55 has already explained the detailed operations at each state.

5.5 Deblocking filter

The H.264 standard defines that filtering can only start when a complete frame picture is reconstructed. However, in the actual implementation, as long as a macroblock and its required neighboring blocks are available, the filtering can be started.

The neighboring blocks are required, because these neighboring pixels will serve as input when filtering the current macroblock boundary and may be modified during filter process. The actual filter process is clearly explained in the algorithm section 3.6.

5.5.1 Architecture of the filter block

Figure 57 shows the proposed architecture of the deblocking filter. All data paths are of 32 bits, which convey four pixels of 8 bits each. The connections between the labeled ports are very complicated with a number of multiplexers.

The register arrays labeled as “4x4 pixel array” Reg10, Reg11, Reg12, and Reg13 are all of 16 registers. They are called Reg1x. Each register is of 8 bits to store a pixel. The signal c0 not shown in Figure 57 controls the registers to either latch upward or remain unchanged.

The register array Reg3 will transpose the input 4x4 array of data controlled by signal c3. The architecture of the transpose register array Reg3 could be found in [20].

The register array Reg2 is similar structure to Reg3. Reg2 has two sets of input m7 and m8, one set of output m9. Basically Reg2 is designed to perform two operations:

- (1) One operation is to latch the input from m8 and delay for 4 clock cycles before latching out at m9
- (2) Another operation is to transpose the input 4x4 data from input m7 and latch out at m9 after 4 clock cycles

In section 4.3 of the literature review, papers [18], [19] and [20] have been discussed with their designs. Basically, design in [19] is a direct implementation from H.264 standard and is frame based. So it is of long latency and needs big memory. The design in [18] needs long input delay and a local memory to hold one macroblock data. As for [20], it reduces the input delay, however still needs a local memory to hold one

entire macroblock data. This is because after filtering the vertical edges, all the 4x4 blocks need to be transposed and saved again for horizontal edges filtering. For the design in this report, the input delay is short; what's more, it does not a local memory. Instead, this is replaced by the register arrays Reg1x (including Reg10, Reg11, Reg12 and Reg13).

Figure 57. Proposed architecture of deblocking filter

Operations

In the designs of [18] and [20], they all follow the rule: filtering all the vertical edges followed by filtering all the horizontal edges. This is because the pixels being modified after filtering the vertical edges will be used as input for filtering the horizontal edges.

Figure 58 (1) shows the labeling of the 4x4 blocks. Figure 58 (2) and (3) show the labeling of vertical edges and horizontal edges to be filtered for the luma component

respectively. The labels like V1 or H1 represent the edges containing 4 edge elements. The edge element has been introduced in Figure 22. Chroma component filtering analysis is similar to the luma component, so it is not discussed.

In [18], the filtering sequence is V0, V1, V2, V3, ..., V15 for vertical edges first, followed by H0, H1, H2, H3, ..., H15 for horizontal edges. This is what is defined in the algorithm section 3.6.1. In [20], the filtering sequence is V0, V4, V8, V12, ..., V3, V7, V11, V15 for vertical edges and H0, H4, H8, H12, ..., H3, H7, H11, H15 for horizontal edges. In both [18] and [20], it is filtering all the vertical edges first, then followed by the horizontal edges.

In my design, the sequence is V0, V1, V2, V3, then V4, H0, V5, H4, V6, H8, V7, H12, followed by V8, H1, V9, H5, V10, H9, V11, H13 and V12, H2, V13, H6, V14, H10, V15, H14, and finally is H3, H7, H11, H15. When it is filtering a particular horizontal edge element, as long as the input 8 pixels have already been filtered in the vertical edge filtering, it does not need to care about whether other vertical edge filtering is completed. This design makes use of this property so that it does not need the memory to hold all the data from vertical edge filtering.

Figure 58. Filtering for luma component

Assume there are left and upper boundaries filtering required. The basic operations are explained below. S1~S7 are actually the states from the state diagram in Figure 59.

S1: First the register arrays Reg13, Reg12, Reg11 and Reg10 (Reg1x) will latch in the left neighboring pixels (block a, b, c and d in Figure 58 (1)) in 16 clock cycles.

S2: Then blocks e, f, g and h will be latched in and connected to the 1-D filter input. Meanwhile the 4 register arrays Reg1x will latch out a, b, c, d and connected to the 1-D filter as well. So the vertical edge is filtered and the modified e, f, g and h are latched into Reg1x. The modified a, b, c and d are outputted. Thus, the vertical edges between a and e, b and f, c and g, d and h (i.e. V0, V1, V2 and V3) are filtered. This takes 16 clock cycles.

S3: After this, block i will be latched in and connected to 1-D filter directly. At the same time, the register arrays Reg1x will latch out the modified block e and connected to the 1-D filter as well. Then the modified block i will be latched to Reg13, while the second time modified e will be latch into Reg3. Here it is filtering the vertical edge elements between blocks e and i (i.e. V4). It remains in S3 for 4 clock cycles.

S4: Then block x is latched into Reg2 at m7. This takes 4 clock cycles.

S5: Then the block x is latched out in transposed form from Reg2 and connected to 1-D filter. At the same time, the second time modified block e is latched out in transposed form from Reg3 and connected to 1-D filter as well. This time it is filtering the horizontal edge between blocks x and e (i.e. H0). So block x and e must be transposed before filtering. The third time modified block e is latched into Reg2 at m8 and the modified block x will be sent to output. It remains in S5 for 4 clock cycles.

S6: The block j is latched in and connected to the 1-D filter. At the same time, the register arrays Reg13, Reg12, Reg11 and Reg10 will latch out the modified block f and connected to the 1-D filter as well. The modified block j is latched into Reg13, while the second time modified block f is latched into Reg3. In this state, it is filtering the vertical edge between block f and j (i.e. V5). It remains in S6 for 4 clock cycles.

S7: The third time modified block e is latched out from Reg2 and the second time modified block f is latched out in transposed form from Reg3. They are the inputs to the 1-D filter. Meanwhile, the third time modified block f is latched into Reg2. The fourth time modified block e is sent to the output. In this state, it is filtering the horizontal edge between blocks e and f (i.e. H5). It remains in S7 for 4 clock cycles.

Then it goes back to state S6 to filter vertical edge elements between blocks g and k (i.e. V6), followed by filtering horizontal edge elements between blocks f and g (i.e. H8) in S7, filtering vertical edge elements between blocks h and l (i.e. V7) in S6 and filtering horizontal edge elements between blocks g and h (i.e. H12). Then it will repeat from states S3 again to work on the other vertical edge together with the corresponding horizontal edge elements.

In this way, it does not need an on-chip memory to hold all the results from vertical edge filtering before horizontal edge filtering. Instead, it does some vertical edge elements filtering, followed by some horizontal edge elements filtering, and so on. So data need to be saved is reduced.

5.5.2 Filter block state diagram

The state diagram in Figure 59 concerns the luma component filtering of a macroblock, while Figure 60 is for the chroma component filtering. For filtering a macroblock, there are three types of edges: upper boundary edge, left boundary edge and internal edges. In the state diagram, the possibility whether upper or left boundary edges need filtering is also considered. In the state diagrams, all the states are repeated itself for 4 times, except state S0 and those states are indicated more than 4 times.

5.5.3 Functional simulation for filter block

Figure 61 shows the functional simulation for the filter block. The signals “left”=1 and “upper”=0 indicate that there is left boundary edge filtering and no upper boundary edge filtering. The signals “alpha”, “beta”, “bs” and “tc0” are the necessary inputs. They remain constant for these edge elements.

The unfiltered inputs are $x_0 \sim x_3$, while filtered outputs are $y_0 \sim y_3$. During state S1, the left neighboring pixels (in Figure 58, blocks a, b, c and d) are latched in. When it is state S2 and “counter2”=0, the first 4 pixels of block e in Figure 58 are latched in and filtering computation starts. When “counter2”=1 in states S2, the 4 filtered pixels of block a in Figure 58 are latched out since they do not need to do horizontal filtering.

For this simulation result, the filtered left neighboring pixels do not change at all.

Figure 59. State diagram of filtering the luma data

Figure 60. State diagram of filtering the chroma data

Figure 61. Functional simulation for filter block.

Chapter 6 Synthesis results

It has been computed in section 4.1 that the H.264 decoder processing speed required for real time decoding is 216,000 macroblock/s. As for the macroblock level pipeline processing, as long as the average processing period for a macroblock required by each functional block within certain time interval is shorter than 4630 ns, the real time processing still can be achieved.

After the functional simulation is passed, the synthesis to gate level is performed. The library is AMS (AustriaMicroSystems) 0.35 μm at 3.3 Volt library. The synthesis is performed using Synopsys Design Compiler tool running on a UNIX system. There is another Synopsys tool called Design Analyzer. It has user graphic interface, which allows the user to see the mapped gate level circuit.

The following paragraphs show a sample synthesis for the integer transform. The results are summarized in Table 10 of section 6.1.

First the VHDL source code is imported into the Design Compiler tool.

```
dc_shell> read -format vhd1 InverseTQ/integer_transform.vhd
```

Since there are other files such as row_it.vhd and transposed_reg.vhd for integer transform circuit, it can repeat the read statement for the other files until all the VHDL files for integer transform block are read in. Then the top file (integer_transform.vhd) is set and saved as a *.db file. This *.db file automatically link all its components.

The following statement will save the design as an unmapped design.

```
dc_shell> write -format db -hierarchy -output db/integer_transform.db
```

Then next time, it only needs to read in this *.db file to do synthesis.

```
dc_shell> read -format db db/integer_transform.db
```

After this, the timing constraint is set and synthesis is carried out below.

```
dc_shell> create_clock -name myclk -period 10 find (port clk)
```

```
dc_shell> set_dont_touch_network myclk
```

```
dc_shell> uniquify
```

```
dc_shell> compile
```

When it finishes the synthesis, timing and gate count information can be retrieved.


```
dc_shell> report_timing
```

Point	Incr	Path

clock myclk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
transpose_register1/second_reg/m0_reg[1]/C (DFE1)	0.00	0.00 r
transpose_register1/second_reg/m0_reg[1]/Q (DFE1)	0.69	0.69 f
transpose_register1/second_reg/y0[1] (skew_reg1)	0.00	0.69 f
transpose_register1/y3[1] (transpose_register)	0.00	0.69 f
mux1_label/y3[1] (mux1)	0.00	0.69 f
mux1_label/U64/Q (MUX22)	0.34	1.03 f
mux1_label/z3[1] (mux1)	0.00	1.03 f
row_it1/x3[1] (row_it)	0.00	1.03 f
row_it1/add_47/plus/B[0] (row_it_DW01_add_16_3)	0.00	1.03 f
row_it1/add_47/plus/U4/Q (NAND20)	0.25	1.28 r
row_it1/add_47/plus/U6/Q (CLKIN0)	0.41	1.68 f
row_it1/add_47/plus/U1_1/CO (ADD32)	0.44	2.12 f
row_it1/add_47/plus/U1_2/CO (ADD32)	0.34	2.46 f
row_it1/add_47/plus/U1_3/CO (ADD32)	0.34	2.81 f
row_it1/add_47/plus/U1_4/CO (ADD32)	0.34	3.15 f
row_it1/add_47/plus/U1_5/CO (ADD32)	0.34	3.49 f
row_it1/add_47/plus/U1_6/CO (ADD32)	0.34	3.83 f
row_it1/add_47/plus/U1_7/CO (ADD32)	0.34	4.18 f
row_it1/add_47/plus/U1_8/CO (ADD32)	0.34	4.52 f
row_it1/add_47/plus/U1_9/CO (ADD32)	0.34	4.86 f
row_it1/add_47/plus/U1_10/CO (ADD32)	0.34	5.20 f
row_it1/add_47/plus/U1_11/CO (ADD32)	0.34	5.55 f
row_it1/add_47/plus/U1_12/CO (ADD32)	0.34	5.89 f
row_it1/add_47/plus/U1_13/CO (ADD32)	0.34	6.23 f
row_it1/add_47/plus/U1_14/CO (ADD32)	0.31	6.54 f
row_it1/add_47/plus/U1_15/Q (XOR31)	0.34	6.89 f
row_it1/add_47/plus/SUM[15] (row_it_DW01_add_16_3)	0.00	6.89 f
row_it1/add_59/plus/B[15] (row_it_DW01_add_16_1)	0.00	6.89 f
row_it1/add_59/plus/U1_15/Q (XOR31)	1.11	7.99 f
row_it1/add_59/plus/SUM[15] (row_it_DW01_add_16_1)	0.00	7.99 f
row_it1/y0[15] (row_it)	0.00	7.99 f
transpose_register1/x0[15] (transpose_register)	0.00	7.99 f
transpose_register1/fourth_mux/x0[15] (mux42_0)	0.00	7.99 f
transpose_register1/fourth_mux/U66/Q (INV3)	0.18	8.17 r
transpose_register1/fourth_mux/U95/Q (OAI212)	0.08	8.25 f
transpose_register1/fourth_mux/y[15] (mux42_0)	0.00	8.25 f
transpose_register1/second_reg/x0[15] (skew_reg1)	0.00	8.25 f
transpose_register1/second_reg/U183/Q (MUX22)	0.29	8.54 f
transpose_register1/second_reg/m0_reg[15]/D (DFE1)	0.00	8.54 f
data arrival time		8.54
clock myclk (rise edge)	10.00	10.00
clock network delay (ideal)	0.00	10.00
transpose_register1/second_reg/m0_reg[15]/C (DFE1)	0.00	10.00 r
library setup time	-0.12	9.88
data required time		9.88

data required time		9.88
data arrival time		-8.54

slack (MET)		1.34

```
dc_shell> report_qor
```

```
Cell Count
```

```
-----
Hierarchial Cell Count:      17
Hierarchial Port Count:     1427
```

```
Leaf Cell Count:          1163
-----
Area
-----
Combinational Area:      93129.257812
Noncombinational Area:  83865.812500
Net Area:                28044.000000
-----
Cell Area:              176995.000000
Design Area:            205039.000000
```

The two commands are report_timing and report_qor. The valuable information is copied from Design Compiler. The results of report_timing show that the critical path delay for integer transform is 8.54 ns when the clock period is set to 10 ns. The results of report_qor show that the Leaf Cell Count is 1163. Leaf Cell Count is just the gate count in terms of the 2-input NAND equivalent gates. It is totally different from the cell count.

In the following subsections, the synthesis results of the five functional blocks are summarized.

6.1 Inverse quantization and transform

The design of Figure 30 proposed in [17] is synthesized using the same AMS 0.35 μm library as shown in Table 9. It can do operations for only Hadamard and integer transform.

Components	Critical path delay (ns)	Gate count
Multiplexed Hadamard and integer transform based on Figure 30 [17]	7.78	1421
Modified multiplexed Hadamard, integer and chroma DC transform based on Figure 32 (proposed)	8.83	1215

Table 9. Comparison of the transform architecture

The modified design for transform based on Figure 32 also includes chroma DC transform. Besides, it reduces the two 1-D transforms to one, so in Table 9 its gate count is 200 less than the design in [17], with very small increment in the critical path delay. As mentioned earlier, the design of Figure 32 needs two clock cycles to latch in four inputs instead of one clock cycle for the design of Figure 30. However, the design

of Figure 32 still meets HDTV application, as it will be shown at the end of the section.

Table 10 shows the synthesis results for the six modules of the algorithm individually without considering any resource sharing in between. Table 11 shows the synthesis results of the integrated quantization and transform components. As shown in Figure 29, the dashed loop includes the inverse chroma and luma DC quantization components.

6 individual components	Critical path delay (ns)	Gate count
Inverse AC quantization	8.52	934
Inverse luma DC quantization	8.55	1399
Inverse chroma DC quantization	8.51	766
Hadamard transform	8.59	1157
Integer transform	8.54	1163
Inverse chroma DC transform	7.26	269
The total number of gate count of separate parts:		5688

Table 10. Individual components without integration for quantization and transform

Integrated blocks	Critical path delay (ns)	Gate count
Hadamard, integer and chroma DC transform	8.83	1215
Inverse chroma and luma DC quantization	8.71	1493
Inverse AC quantization	8.52	934
The three integrated blocks total gate count:		3642
The final integrated quantization and transform	9.31	4676

Table 11. Integrated blocks and integrated circuit for quantization and transform (in Figure 29)

In Table 10, the total number of gate count of the six parts before integration is 5688. After integration, the corresponding six parts becomes three integrated blocks. In Table 11, it shows the three integrated blocks have altogether 3642 gate count. In fact, in the integrated architecture, the 1-D Hadamard transform hardware is shared and reused to perform the three transforms (Hadamard, interger and chroma DC transform). Besides, the chroma quantiztaion and luma DC quantization share the common multiplier. Thus, by integration, about 36% saving of hardware is achieved

with only slight increase in the critical path delay for the components of the quantization and transform.

Table 11 also shows the final integrated quantization and transform circuit results. The gate count is 4676. It includes the integrated components, the finite state machine for control operations and quantization step table. The critical path delay is 9.31 ns, which means that the designed circuit can operate at about 107 MHz.

In the worst case, for a 16x16 luma intra predicted macroblock, it needs to process sixteen 4x4 luma blocks plus one 4x4 DC block. In 4:2:0 sampling format there are eight 4x4 chroma blocks and two 2x2 DC blocks. Altogether there are 408 data in one macroblock. Since the circuit is designed to latch in and process one set of four data inputs every two clock cycles, one macroblock is restricted to $(408/4)*2=204$ clock cycles to finish its inverse quantization and transform operation.

As the clock frequency is 107 MHz, it can handle $107 \times 10^6 / 204 = 524,510$ macroblocks/s. The HDTV application requires the processing speed of 216,000 macroblocks/s, so the design is fast enough to meet this HDTV application.

In section 4.3 of literature review, it is mentioned that the area optimized transform design of [25] can produce each output in 7 clock cycles. Then the normal transform for a macroblock needs $16 \times 16 \times 1.5 \times 7 = 2688$ clock cycles. By a glance, it is more than 10 times slower than my design. And it also cannot meet my HDTV application.

6.2 Reorder

The time allocated to a macroblock is 4630 ns and there are $16 + 1 + 2 \times 4 = 25$ 4x4 blocks. That means that the time period allocated for one 4x4 block is $4630/25 = 185.2$ ns.

The state diagram in Figure 37 shows the flow of reordering one 4x4 block. The maximum number of states taken will be the flow path of S1, S2, S5, S8, S12... . The states S1, S2, S8 takes only one clock cycle each. After state S12 there will be 16 clock cycles to latch out the 16 outputs. In state S5 it is latching in level and S12 it is latching in run_before. The worst case of the number of clock cycles taken by S5 is 16, which means it does not have zero coefficient. However, this means that there is no run_before to be latched in at S12. So in combination, the worst case clock cycles taken by S5 and S12 will be $15 + 14 = 29$. So for the worst case, the maximum number of clock cycles taken will be 48. So the required clock period should be less than $185.2/48 = 3.86$ ns.

The synthesis results for gate count are shown below:

components	Gate count
level LIFO	830
run_before LIFO	201
Controller	535
Reorder	1605

Table 12. Reorder synthesis results

As shown in Table 12, level LIFO has bigger gate count than run_before. This is because the register in run_before LIFO is of 4 bits, while in level LIFO it is of 16 bits. The synthesis results reflect this ratio of 4 times in the gate count. The controller needs to do the simple counting to compare the value of run_before.

The critical path delay for reorder block is also very small at 1.91 ns. It meets the requirement quite well since $1.91 < 3.86$.

6.3 Context Adaptive Variable Length Coding

Table 13 shows the synthesis results for the CAVLC block in Figure 39. Table 13 (a) shows the synthesis results for specific components of CAVLC. The complete CAVLC block has a critical path delay of 4.74 ns while the level computation component has a critical path delay of 4.51 ns. The critical path delay of the complete CAVLC block is mainly determined by this level computation component. The delay of the level computation component is mainly due to the add operation in the algorithm. So the level computation component design is very critical to the CAVLC design.

Components	Critical path delay (ns)	Gate count
Barrel shifter	1.68	582
total_coeff and num_trailingone decoding	2.45	968
Level_prefix decoding	1.31	57
Level computation	4.51	741
Run_before decoding	2.51	176
Total_zeros decoding	2.28	403
Vlcnum update	0.7	105
CAVLC	4.74	3374

(a)

Components	Critical path delay (ns)	Gate count
Paper [24] level computation	5.25	902
Level computation (proposed)	4.51	741

(b)

Table 13. CAVLC synthesis results

Table 13 (b) compares the level computation component in my design with that in paper [24]. As explained in the section 5.3.2 architecture of CAVLC, level computation in paper [24] directly implements the algorithm in standard document [2]. This appears neater and easier in documentation, however it is not optimized in hardware. It increases the operation bit width of adders by one bit. The VHDL code is written for level computation component in paper [24]. The same AMS library is used to do synthesis for a fair comparison. The improvement is obvious in Table 13 (b). The critical path of level computation component is reduced in my design by 0.74 ns while the gate count is also reduced.

The worst case for the number of clock cycles to decode a 4x4 block happens when there are fifteen levels and there is another zero in the starting position of the scanned array. This will require fifteen level decodings and fifteen run_before decodings. In the state diagram of Figure 40, it has 3 clock cycles from S1 to S3. Fifteen repetitions of level decoding during S5~S6 take 30 clock cycles. State S7 takes another 1 clock cycle. Fifteen S8 takes another 15 clock cycles. State S9 takes another 1 clock cycle. So the worst case for a 4x4 block needs 50 clock cycles.

The time period allocated to a macroblock is 4630 ns. For a macroblock, there are altogether $16 + 1 + 2 \times 4 = 25$ 4x4 blocks. Assume the clock period is T, then

$$\rightarrow 25 * 50 * T < 4630$$

$$\rightarrow T < 3.704 \text{ ns.}$$

The critical path delay in Table 13 (a) of CAVLC block is 4.74 ns. So if the video appears to be the worst case for CAVLC, the CAVLC block could not work in real time operation for the HDTV application of video format 1280x720 at 60 Hz.

Of course, the worst case for a block will rarely happen, and even more rarely happen for a few 4x4 blocks within a certain continuous period. Most of the time, it takes much fewer number of clock cycles than 50 to decode a 4x4 block. The video sequence foreman_qcif.yuv is simulated using JM7.5C. The result shows that it only takes 10 clock cycles on average to decode a 4x4 block. Since there will be buffers in between the different functional blocks, as long as the average number of clock cycles taken to decode a 4x4 block within certain continuous period is small, real time operations could be achieved.

As explained earlier, the critical path delay of CAVLC block is mainly decided by the level computation component. So in order to have a shorter critical path delay, it ought to reduce the critical path delay of the level computation component.

Other options to make CAVLC block process faster are available. For example, it can be designed that the CAVLC block decodes each level in one clock cycle by combining the state S5 and S6 together. This increases the actual critical path delay, however it reduces the number of clock cycles to finish a 4x4 block from 50 to 35.

6.4 Intra prediction

In the intra prediction, for luma 4x4 intra prediction, it requires 65 clock cycles to finish at state S13.

For luma 16x16 intra prediction, its plane and DC prediction modes both need 69 clock cycles, while its vertical and horizontal prediction modes both need 65 clock cycles. So the worst case for luma 16x16 intra prediction is 69 clock cycles.

For chroma 8x8 intra prediction, the worst case is 36 clock cycles.

Thus for the complete macroblock, the worst case will be $36 + 69 = 105$ clock cycles. Assuming that the clock period is T , in order for intra prediction process to be in real time operation, $105 \cdot T < 4630$.

→ $T < 44.1 \text{ ns}$.

Components	Critical path delay (ns)	Gate count
Luma 4x4	11.05	1269
Luma16x16 & chroma8x8	11.72	2712
Intra prediction	13.88	4820

Table 14. Intra prediction synthesis results

As the synthesis results shown, the critical path delay of luma 4x4 is close to the critical path delay of luma 16x16 block, around 11 ns. This is good because for the whole intra prediction, it is always needed to consider the one with the longest critical path. The intra prediction component will have another few multiplexers, which will increase the critical path delay slightly. This could be observed. As far as the critical path delay is concerned, the design meets the timing requirements.

6.5 Deblocking filter

The time allocated to a macroblock is 4630 ns. In the state diagram, it can be found that the worst case happens when there are both upper and left boundary filtering.

So for the luma component, the number of clock cycles taken is:

$$16 + 16 + (4 + 4 + 4 + (3 \times 8)) \times 3 + 4 + 16 = 160$$

For the chroma component, the number of clock cycles taken is:

$$2 \times (4 + 8 + 4 + 20 + 8) = 88$$

In order to filter a macroblock, it takes 248 clock cycles.

→ $248 \times T < 4630$

→ $T < 18.67 \text{ ns}$

Synthesize the hardware at $T = 10 \text{ ns}$, and then the synthesis results for the architecture in Figure 57 is shown in Table 15.

Components	Critical path delay (ns)	Gate count
Reg1x	--	1395
Reg2	--	468
Reg3	--	450
I-D Filter	8.04	3034
Filter block	9.92	6051

Table 15. Deblocking filter synthesis results

In the previous designs of [18] or [20], there is a large on-chip memory of about 160x32 bits (to save 40 4x4 blocks data). In my design, this on-chip memory is replaced by the register array Reg1x in Figure 57, of size 16x32 bits (to save 16 4x4 blocks data). However, for the memory, it requires extra circuitries like address generator. As for the register arrays in my design, it does not need that.

In this project, only gate level synthesis is performed. So it cannot show the gate count of the memory. So for gate count comparison purpose, it is assumed that on-chip memory in the previous designs is replaced by the registers of the same size, which is 160x32 bits. The register array Reg1x of size 16x32 bits has a gate count of 1395, so

the gate count of the registers of 160x32 bits is about 13950 if it is synthesized in the same AMS library.

The architecture of Figure 28 in [18] uses the same components like Reg2, Reg3 and 1-D filter as in my design. Their gate counts are 468, 450 and 3034 respectively. Since the on-chip memory modeled as register of the same data storage size has a gate count of 13950, the total gate count of these components of the design in [18] could be calculated as $468+450+3034+13950 = 17902$. Yet this gate count does not include those logic gates in its state machine for the design in [18].

The filter block gate count is 6051 in my design, so my design reduces the hardware area sharply, by almost 66%.

In turn, if the register array Reg1x in my design is implemented as memory, the storage requirement is only 16x32 bits. In comparison, the design in [18] needs a memory of size about 160x32 bits, which is 10 times the size of memory in my design.

Chapter 7 Conclusions and recommendations

7.1 Conclusions

In this project, most of the important components of H.264 baseline profile decoder have been designed for HDTV application of video format 1280x720 at 60 Hz. The algorithms are summarized and explained with their improved features in the report.

The hardware circuits have been synthesized to the gate level using the AMS (AustriaMicroSystems) 0.35 μm at 3.3 Volt technology. The hardware includes the inverse Q and T block, the CAVLC block, the Reorder block, the Intra prediction block and the Deblocking filter block.

For inverse Q and T block, an integrated design has been implemented with hardware saving of 36% over the using the separate blocks. The results have been presented by us in the paper [23]. For the CAVLC block, by optimizing the level computation component, its critical path delay is reduced by 0.74 ns compared to that in paper [24]. By analyzing the hardware specifically, the deblocking filter has also been designed with hardware saving of 66% over the architecture in [18] if the on-chip memory is modeled by registers for comparison. Table 16, Table 17 and Table 18 compare the features of the previous designs and my design of inverse quantization and transform block, level computation component of CAVLC block and filter block respectively.

Designs	Features
The area optimized transform design of [25]	It produces each output in 7 clock cycles. It is too slow for HDTV application.
Multiplexed transform based on Figure 30 [17]	The transform can do operation for Hadamard and integer transform. There are two 1-D transform blocks in Figure 30. Four inputs are latched in each clock cycle.
The proposed integrated architecture of inverse quantization of transform based on Figure 29 [23]	It is a complete block for inverse quantization and transform. Transform: Transform can do operation for Hadamard, integer and chroma transform. The design in Figure 32 is proposed with only one 1-D transform block. Four inputs are latched in every two clock cycles. Quantization: The luma DC quantization and chroma DC quantization share one multiplier. The AC quantization in Figure 33 only needs two multipliers instead of four.

Table 16. Features comparison of inverse Q and T block

Designs	Features
Level computation component of CAVLC block in [24]	For the critical level computation component, it implements the method in the standard [2] directly.
Level computation component in my design	The method is modified such that it can reduce the operation bit width by one bit. This reduces the delay and gate count. Since this is the critical path of the CAVLC block, so the critical path delay of CAVLC block is also reduced.

Table 17. Features comparison of level computation component of CAVLC block

Designs	Features
Deblocking filter in [18]	It needs to load the entire macroblock before actual filtering could start. It needs a local memory to hold one macroblock data, about 160x32 bits
Deblocking filter in [20]	The equivalent loading period is reduced to load four 4x4 blocks. It still needs a local memory to hold one macroblock data.
Deblocking filter in my design	The equivalent loading period is reduced to load four 4x4 blocks. Instead of a local memory of about 160x32 bits, it is replaced by registers of size 16x32 bits

Table 18. Features comparison of deblocking filter block

The synthesis results are summarized in Table 19. The second column is the gate count for the different functional blocks. The third column shows the critical path delay together with the corresponding achievable operating frequency. This frequency is the highest frequency that the functional block could work without timing violation. The fourth column calculates the required minimum clock frequency for the worst case in my design to meet the processing speed of HDTV application of video format 1280x720 at 60 Hz. As shown, they all meet the requirements very well except for the CAVLC block.

Components	Gate count	Critical path delay (ns) / frequency (MHz)	Required minimum clock frequency (MHz)
Inverse Q and T	5680	9.31 / 107	44
Reorder	1605	1.91 / 523.6	250
CAVLC	4972	4.74 / 211	270
Intra prediction	4820	13.88 / 72	22.67
Deblocking filter	6051	9.92 / 100.8	53.56

Table 19. Summarized synthesis results

The achievable operating frequencies of inverse quantization and transform, reorder, intra prediction, and deblocking filter functional blocks are about double that of the

required frequencies or even higher. This is good since during final integration of these functional blocks to the layout level, extra wire delays will be introduced.

In order to illustrate the decoder latency, it is assumed that the basic clock frequency is 270 MHz. Then it can have the synchronized clock frequencies of 135 MHz, 54 MHz and 27 MHz.

Let 270 MHz be operating frequency for the reorder block. So the period for reordering one macroblock is $16 \times 48 \times (1/270) = 2844$ ns. Let 54 MHz be the operating frequency for inverse quantization and transform, and deblocking filter blocks. So the period to inverse transform and quantize a macroblock is $204 \times (1/54) = 3778$ ns, while for deblocking filter it is $248 \times (1/54) = 4593$ ns. Let 27 MHz be the operating frequency for the intra prediction block. So the period to intra predict one macroblock is $105 \times (1/27) = 3889$ ns. In the Table 19, it could be found that the assigned operating frequencies are all below the achievable operating frequency while bigger than the required minimum frequency. All these processing periods of the functional blocks for one macroblock are for the worst case. In the actual decoding process, the periods taken by the functional blocks could be much shorter.

While for CAVLC block, it is assigned the operating frequency of 135 MHz. This frequency is below the achievable operating frequency, however also below the required minimum frequency in the worst case. However, as discussed in section 6.3, the CAVLC block takes 10 clock cycles on average to decode a 4x4 block data. For a safer analysis, the average clock cycle is doubled, which is 20 clock cycles. Then the period to decode a macroblock for CAVLC is $25 \times 20 \times (1/135) = 3704$ ns. This is smaller than 4630 ns, which is the period allocated to decode a macroblock for real time operation. If ignoring the delays caused by the inter-functional block buffers, the latency introduced by all the five functional blocks is $2844 + 3778 + 4593 + 3889 + 3704 = 18808$ ns = 18.808 μ s.

7.2 Recommendations

There are some further improvements to be made for the project.

1) The complete H.264 decoder has not yet been finished due to the huge amount of work. The system level integration is a difficult part for a complete decoder. It involves with the complicated memory management. Besides, buffer control at the system level is also very critical to make the decoding successful. And there is still another component, the inter prediction yet to be designed. Some extra time is required to finish this component.

2) In this project, the separate functional process has been investigated carefully. However, the overall H.264 decoding algorithm could have been explored in more detail so that more features could be found. This may save some redundant processing.

3) Although it is shown that the CAVLC block is able to decode in real time in practical video, however, theoretically, it may not be able to work in real time for the worst case. More work may be put on to improve its performance. As recommended in section 6.3, we can design the CAVLC block to decode each level in one clock cycle instead of two clock cycle.

4) The actual hardware needs to do layout and layout simulation. During this step, extra wire delays will be introduced. This has not been done.

Bibliography:

- [1] Wiegand T., Sullivan G. J., Bjntegaard, G. and Luthra, A., “Overview of the H.264/AVC Video Coding Standard”, IEEE Transactions on Circuits and Systems for Video Technology, Volume 13, Issue 7, pp. 560–576, July 2003
- [2] Wiegand T., “Joint Final Committee Draft (JFCD) of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC)”, document JVT-D157, JVT of ISO/IEC MPEG & ITU-V VCEG, 4th Meeting, Klagenfurt, Austria, 22–26 July 2002
- [3] Gomila, C. and Yin, P., “New features and applications of the H.264 video coding standard”, Proceedings of International Conference on Information Technology: Research and Education, pp. 6–10, 11–13 August 2003
- [4] A. Joch, F. Kossentini, H. Schwarz, T. Wiegand, and G. J. Sullivan, “Performance comparison of video coding standards using Lagrangian coder control”, Processing of International Conference on Image, Volume 2, pp. II-501 – II-504, 22–25 September 2002
- [5] Iain E. G. Richardson, “H.264 and MPEG-4 video compression: video coding for next generation multimedia”, Wiley press, pp. 3, 61, 88, 160. 2003
- [6] List P., Joch A., Lainema J., Bjntegaard G. and Karczewicz M., “Adaptive deblocking filter”, IEEE Transactions on Circuits and Systems for Video Technology Volume 13, Issue 7, pp. 614–619, July 2003
- [7] Malvar H., Hallapuro A., Karczewicz, M. and Kerofsky L., “Low-complexity Transform and quantization with 16-bit arithmetic for H.26L”, Proceedings of International Conference on Image Processing, Volume, 2, pp. II-489 – II-492, 22–25 September 2002
- [8] Jianning Zhang, Yuwen He, Shiqiang Yang and Yuzhuo Zhong, “Performance and complexity joint optimization for H.264 video coding”, Proceedings of International Symposium on Circuits and Systems, Volume 2, pp. II-888 – II-891, 25–28 May 2003
- [9] Tamhankar A. and Rao K. R., “An overview of H.264/Mpeg-4 Part 10”, 4th EURASIP 2003 Conference focused on Video/Image Processing and Multimedia Communications, Volume 1, pp. 1–51, 2–5 July 2003

- [10] Yun Q. Shi and Huifang Sun, "Image and video compression for multimedia engineering", CRC press, chapter 1.1, 11.2, 16.1. 2000
- [11] Iain E. G. Richardson, "Video codec design", Wiley press, pp. 31, 42, 49, 128,164. 2002
- [12] Marpe D., Blattermann G. and Wiegand T., "Video compression using context-based adaptive arithmetic coding", Proceedings of International Conference on Image Processing, Volume 3, pp. 558–561, 7–10 October 2001
- [13] Hu Y., Simpson A., McAdoo K. and Cush J., "A high definition H.264/AVC hardware video decoder core for multimedia SoC's", 2004 IEEE International Symposium on Consumer Electronics, pp. 385–389, 1–3 September 2004
- [14] Amer I., Badawy W. and Jullien G., "Towards MPEG-4 part 10 system on chip: a VLSI prototype for context-based adaptive variable length coding (CAVLC)", IEEE Workshop on Signal Processing Systems, 2004, pp. 275–279, 13–15 October 2004
- [15] Gisle B. and Karl L., "Context-adaptive VLC (CVLC) coding of coefficients ", document JVT-C028, JVT of ISO/IEC MPEG & ITU-V VCEG, 3rd Meeting, Fairfax, Virginia, USA, 6–10 May 2002
- [16] Aggoun A. and Jalloh I, "Two-dimensional DCT/IDCT architecture", Proceedings of Computers and Digital Techniques, IEE, Volume 150, Issue 1, pp. 2–10, 20 January 2003
- [17] Tu-Chih Wang, Yu-Wen Huang, Hung-Chi Fang, and Liang-Gee Chen, "Parallel 4x4 2D transform and inverse transform architecture for MPEG-4 AVC/H.264", Proceedings of the 2003 International Symposium on Circuits and Systems, Volume 2, pp. II-800 – II-803, 25–28 May 2003
- [18] Yu-Wen Huang, To-Wei Chen, Bing-Yu Hsieh, Tu-Chih Wang, Te-Hao Chang and Liang-Gee Chen, "Architecture design for deblocking filter in H.264/JVT/AVC", Proceedings of 2003 International Conference on Multimedia and Expo Volume 1, pp. I - 693–696, 6–9 July 2003

- [19] Miao Sima, Yuanhua Zhou and Wei Zhang, "An efficient architecture for adaptive deblocking filter of H.264/AVC video coding", IEEE Transactions on Consumer Electronics, Volume 50, Issue 1, pp. 292–296, February 2004
- [20] Shih-Chien Chang, Wen-Hsiao Peng, Shih-Hao Wang and Tihao Chiang, "A platform based bus-interleaved architecture for deblocking filter in H.264/MPEG-4 AVC", IEEE Transactions on Consumer Electronics, Volume 51, Issue 1, pp. 249–255, February 2005
- [21] Bai-Jue Shieh, Yew-San Lee and Chen-Yi Lee, "A high-throughput memory-based VLC decoder with codeword boundary prediction", IEEE Transactions on Circuits and Systems for Video Technology, Volume 10, Issue 8, pp. 1514–1521, December 2000
- [22] Change S. F. and Messerschmitt D. G., "Designing high-throughput VLC decoder. I. Concurrent VLSI architectures", IEEE Transactions on Circuits and Systems for Video Technology, Volume 2, Issue 2, pp. 187–196, June 1992
- [23] Ho D. J. and Zhang C. C., "Hardware architecture for an integrated inverse transform and quantization of the H.264 decoder", 2004 the 10th International Symposium on Integrated Circuits, Devices and Systems, 8–10 September 2004
- [24] Wu Di, Gao Wen, Hu Mingzeng and Ji Zhenzhou, "A VLSI architecture design of CAVLC decoder", Proceedings of 2003 5th International Conference on ASIC volume 2, pp. 962–965, 21–24 October 2003
- [25] Kordasiewicz R. C. and Shirani S., "ASIC and FPGA Implementations of H.264 DCT and Quantization blocks", IEEE International Conference on Image Processing, Volume 3, pp. 1020–1023, 11–14 September 2005
- [26] Jiahui Zhu, Ligang Hou and Wuchen Wu, "High Performance Synchronous DRAMs Controller in H.264 HDTV Decoder", Proceedings of 2004 7th International Conference on Solid-State and Integrated Circuits Technology, Volume 3, pp. 1621–1624, 18–21 October 2004