Imperial College London

FPGA/ASIC Implementation of JPEG 2000 Encoder for Real-time Image Compression

Author

Zhiyuan Zhu

CID: 02261066

Supervised by

Dr Pantelis Georgiou

A Thesis submitted in fulfillment of requirements for the degree of
Master of Science in Analogue and Digital Integrated Circuit Design

Department of Electrical and Electronic Engineering
Imperial College London
2023

Abstract

In the past 20 years, the JPEG image standard was in the leading position in most of the fields due to its efficient algorithm and remarkable compression capability. However, in some specific area, JPEG is not suitable for some area due to some drawbacks such as the artifact produced along the compression process and the lack of compression capability at high compression rate which makes the image bluer. Therefore, JPEG2000 is introduced when facing those problems.

This thesis implement the hardware of still image compression system called JPEG2000. It provides many new features including controllable compression rate, better compression quality and region of interest (ROI) and so on.

First, it introduces the cutting-edge image compression system, JPEG2000. which uses new transform algorithm and coding system. Subsequently, it compares the compression capabilities of JPEG and JPEG2000 on the Matlab platform. Following this, the algorithm is implemented in Matlab, and its accuracy is validated using the open-source project Openjpeg. Lastly, it designs the hardware specifically for the JPEG2000 algorithm, aiming for a fully pipelined architecture with an iteration interval of one.

The comparison part is implemented on Matlab and the compression ability of different standards including the compression type, compression rate, quality of compression are compared.

In the realm of hardware design, the algorithmic-to-hardware mapping unfolds in a structured, five-stage process, engineered to operate with a fully pipelined architecture. The initial stage entails pre-processing the image to condition it for subsequent computational steps. The second stage involves the application of Discrete Wavelet Transform (DWT) to the image data, facilitating the compaction of image energy. The process then transitions to the third stage, where the transformed image stream undergoes lossy quantization to approximate its original informational content. Subsequently, the quantized data is subjected to Embedded Block Coding with Optimal Truncation (EBCOT) in the fourth stage, a specialized entropy coding system designed for optimal data compression. The final stage culminates in the formation of the compressed image bit-stream. First three of these stages has been meticulously designed to function within a fully pipelined architectural framework, thereby optimizing computational efficiency and minimizing data throughput

latency.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced.

Copyright Declaration

Acknowledgments

I would like to thank Dr. Pantelis Georgiou for offering the project that I am really interested in.

I would also like to thank Mr.Lei Kuang for his helpful suggestion and valuable knowledge on the subject which support me a lot in doing the project. He taught me a lot of experiences when doing the designing of the project.

Lastly, I would like to thank my parents who support me and encourage me finishing the project.

Contents

List of Acronyms

dB decibel

CR Compression Ratio

DCT Discrete Cosine Transform

FDCT Forward Discrete Cosine Transform

PSNR Peak Signal to Noise Ratio

SSIM Structural Similarity Index

WT Wavelet Transform

DWT Discrete Wavelet Transform

2D-FDWT 2 Dimensional Forward Discrete Wavelet Transform

USDZQ uniform scalar dead-zone quantization

RLC Run Length Coding

MSB Most Significant Bit

LSB least Significant Bit

ROI Region Of Interest

EBCOT Embedded Block Coding with Optimal Truncation

ICT Irreversible Color Transformation

IDE Integrated Development Environment

2D-FDCT 2 Dimensional Forward Discrete Wavelet Transform

CDF 9/7 Cohen–Daubechies–Feauveau wavelet 9tap / 7tap wavelet

CDF 5/3 Cohen–Daubechies–Feauveau wavelet 5tap / 3tap wavelet

List of Figures

1 Introduction

Contents

This thesis details the development of a fully pipelined hardware designed to execute the JPEG2000 lossy compression process, tailored specifically for images of high volume and high throughput. In Section 1.1, the motivation behind this project is elucidated, drawing inspiration from an ultra-high frame rate imaging system. Subsequently, an overview of the JPEG2000 compression system is presented. This is followed by a introduction of the JPEG and JPEG2000 still image compression system. Lastly some novel implementations of the algorithm in Section 1.5.

1.1 Motivation

This project is initiated to optimize the output image viewing application of an ion imaging system[1]. As shown in the data compression comparison figure 1.1, The system throughput is 128*128 size gray scale image with a high frame rate up to 6100fps. This feature requires a high compression rate real time compression system. Using the optimized JPEG real time compression system brought up by the paper, it reaches a compression rate of 6.63 with Peak Signal to Noise Ratio (PSNR) of 28 dB decibel (dB). The Compression Ratio (CR) is 13.03% higher than the

1

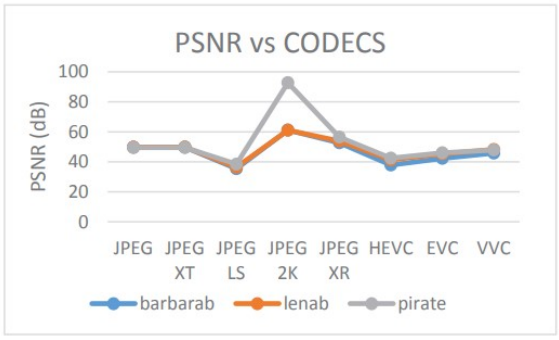


Fig. 9(e) PSNR vs CODECS for greyscale images

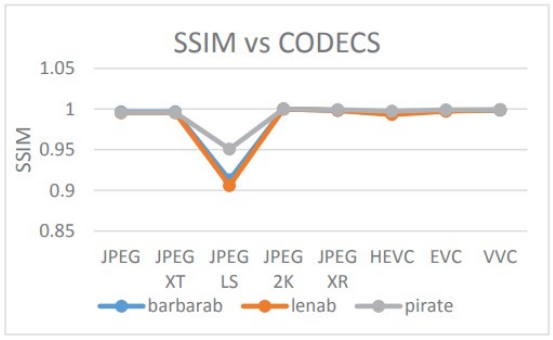


Fig. 9(f) SSIM vs CODECS for greyscale images

Figure 1.1: The PSNR and SSIM comparison for gray scale image using various standard [3].

standard JPEG CR. Whereas, under such a high throughput pressure, these level of compression gives a relatively accepted compression effect. Besides, the artifact of Discrete Cosine Transform (DCT) turns out which creates some blocking artifacts and mosquito noise [2]. The resulting image from JPEG is not the most suitable for rough inspecting in terms of its compression ratio and fidelity. A new compression system is needed to further optimize the compression capability and corresponding quality.

1.2 Background

A general overview of the image compression techniques are investigated and mentioned in [4]. It says that the basic redundancy of image consists of 1. Coding redundancy where the information can be stored in shorter length for transmission. 2. Interpixel redundancy which due to the correlation between pixels. 3. Psycho visual redundancy which is the data than can be ignored by human perception system. The image compression techniques are employed to reduce those redundancies that lies in the image. Lots of the image coding methods are introduced for lossless compression and some transform coding theory are explained for lossy compression system. It is a general overview of the compression techniques. For lossy compression, DCT and DWT techniques are mentioned and explained in general which is not enough for researching the algorithm in detail but is good at realizing the basic working principle.

In this thesis, the aim is to investigate the JPEG2000 standard as a mechanism for enhancing image quality while reducing the compression artifacts commonly associated with JPEG. JPEG2000 serves as an internationally recognized standard for still image compression, established by the International Organization for Standardization (ISO) and delineated in the ISO/IEC CD 15444-1 international standard document. The basic procedures are explained in [5]. Where the JPEG2000

consists of tiling, Wavelet Transform (WT), Quantization and Entropy coding. However, after careful inspection, the entropy coding system explained in this paper is not the same as the one used in the international standard which may due to the update of the ISO standard.

After all, The core philosophy behind the compression techniques and similar compression standards lies in exploiting various types of redundancy present in still images, namely spatial, spectral, and psycho-visual redundancies.

Spatial redundancy refers to the correlations between adjacent pixels. When neighboring pixels exhibit similar or identical values, it signals an opportunity for optimization. Spectral redundancy, on the other hand, pertains to frequency correlations. In the context of still image compression, the spatial and frequency domains serve roles analogous to the time and frequency domains in video or audio encoding.

Human perception assigns different weights to different frequency components within an image. Lower frequency elements generally convey the essential shapes and structures, providing a comprehensive overview of the image and facilitating object recognition. Research in Visual Psychophysics, as evidenced by works from C.A. Parraga, T. Troscianko, and others [6], suggests that the human visual system prioritizes these general features. Consequently, the high-frequency components, which are less critical to human perception, can be selectively discarded or quantized without significantly compromising the perceived image quality. From which, quantization is employed to reduce the none significant information. The [7] explained the quantization step in detail which explain the mathematical theory of uniform scalar dead-zone quantization (USDZQ) strategy defined in JPEG2000 part1 which is helpful to the implementation of the quantization part of JPEG2000. For the entropy coding system, it is detailed explained in [8] where the whole process of JPEG2000 is discussed and demonstrated in detail including the new features that it supports such as the Region Of Interest (ROI) which can decode the interested area of image first.

Under this predecessor, through a set of operation, as mentioned above such as transform, quantization and entropy coding (lossless or lossy), those redundancy can be overlooked. With some intelligent coding algorithm applied, excellent compression performance and practical features are achieved.

When compared to other image compression algorithms such as JPEG, JPEG XT, JPEG LS, JPEG XR, HEVC, EVC, and VVC, JPEG2000 has demonstrated superior performance in terms of gray scale image compression. Empirical evidence supports this claim, showing that JPEG2000 outperforms other standards in both Peak Signal to Noise Ratio (PSNR) and Structural

Similarity Index (SSIM) metrics [3]. These metrics serve as quantitative measures for evaluating the degradation of image quality pre- and post-compression.

Specifically, JPEG2000 boasts a range of innovative and robust features that set it apart from previous standards. As discussed in the contrast research conducted by Reena Singh, V and K. Srivastava [9], the JPEG2000 standard provides both effective lossy and lossless compression capabilities. The standard is driven by the evolving requirements of modern applications, catering to demands for advanced features while also providing better compression performance, particularly at low bit rates. Other Key features include progressive transmission based on pixel accuracy and resolution, multiple resolution representation, robust error resilience, random access and processing and region-of-interest capabilities. A seminal text for both learning and engaging in advanced research on JPEG2000 is delineated in [10]. This comprehensive resource meticulously explores the fundamental principles underlying digital imaging, the techniques employed for redundancy exploitation, and the intricate architecture of the compression system. Notably, the work offers an in-depth examination of various entropy coding methodologies, quantization techniques, and image transformations. Moreover, it elucidates the code stream syntax and file formatting specifications. By using JPEG2000 as an illustrative example of an image compression system, the text provides invaluable insights, making it an indispensable reference for this project.

In addition to the previous comprehensive resource on image compression techniques, there exists another seminal work that investigates specifically into the compression workflow of JPEG2000 [11], elucidating each step in granular detail. Beyond its focused scope on JPEG2000, the article also conducts a performance comparison between JPEG2000 and other prevailing compression standards. This article serves as an exemplary tutorial on the JPEG2000 standard, offering invaluable insights that significantly contribute to a nuanced understanding of the subject matter.

1.3 JPEG

JPEG is a pervasive standard for still image compression that has been widely adopted since its introduction in 1992 by the Joint Photographic Experts Group. Known for its versatile features, JPEG features compact file sizes, high-resolution graphics, adjustable compression rates, and extensive cross-platform compatibility. The JPEG compression process encompasses a series of linked stages as mentioned in [12]: color mapping, preprocessing, 2D Forward Discrete Cosine Transform 2 Dimensional Forward Discrete Wavelet Transform (2D-FDCT), quantization, Zigzag sequencing,

entropy coding, and finally, bitstream formation. This comprehensive procedure ensures efficient data reduction while maintaining visual fidelity, solidifying JPEG's enduring popularity in various applications The details of the process of JPEG is given in [13] which explains the details of each processing step for lossy and lossless compression. However only lossy part is used in this thesis.

Color mapping is an optional step, particularly when dealing with grayscale images, such as the image in ion imaging systems mentioned before. In such cases, only one color component is processed. For images in the multiple components domain, channels must be encoded separately. D.J. Granrath has conducted research on the human visual system in perceptual models [14]. It's convinced that using Fourier domain coding can achieve 10 times bit rate reduction without any noticeable loss when the image is divided into different color components. In basic image processing, images are originally represented in the Red, Green, and Blue (RGB) color domain. However, in his research, it transforms this into an alternative color domain consisting of Luminance, Red-Green, and Yellow-Blue channels. This alternative domain bears similarities to the established YCbCr or YUV color spaces. Notably, our experiments show that the degradation of the Red-Green and Yellow-Blue channels results in negligible loss in image quality. This underscores the varying significance of different color components in image representation.

During the preprocessing stage, the input image is divided into multiple 8x8 blocks, Each element in this 8*8 block is level shifted to have a symmetric magnitude range above or under the zero. which are then fed into the 2D-FDCT module. The 2D-FDCT process involves applying both vertical and horizontal 1D-DCT operations on each column and row, respectively. Upon simplification, this operation can be represented as matrix multiplication.

The resulting coefficients transform the original image into a weighted sum of 64 fundamental cosine waves. The coefficients in the upper-left corner of the matrix correspond to low-frequency components, while those in the lower-right corner represent high-frequency components. After Forward Discrete Cosine Transform (FDCT), the low-frequency region usually contains coefficients with larger magnitudes, significantly different from zero. In contrast, the high-frequency region often has coefficients close to zero. This is attributed to the energy compaction property of the DCT, as cited in existing literature [15].

Quantization is the process in which less critical information is selectively eliminated, serving to streamline the overall data. After undergoing quantization with a given quantization table, high-frequency coefficients often get quantized to zero. In the resultant coefficient matrix, most of the energy is concentrated in the upper-left corner, while the lower-right corner typically contains

minimal energy and zeros.

To enhance the efficiency of encoding, a zigzag sequence is employed. This sequence prioritizes low-frequency, more important components before high-frequency, less crucial ones, thereby deploying more effective data compression. The result sequence is applied to Run Length Coding (RLC) to remove the redundancy.

Besides, the entropy coding system is applied. For JPEG, the Huffman coding system is used which further compresses the resulting bitstream from RLC. One thing that is worth noting is that The first element of each 8*8 block called the DC coefficient is coded differently from the other coefficient which is called the AC coefficient. The encoded image is then organized in a specific form to produce a JPEG image file.

1.4 JPEG2000

The JPEG2000 standard was developed by the Joint Photographic Experts Group between 1997 and 2000 and subsequently ratified as an international standard by ISO and IEC. Unlike its predecessor JPEG, JPEG2000 excels in scenarios with bandwidth limitations, such as the transmission of large medical or satellite images. This is due to its superior compression capabilities and exceptional image quality, particularly at low bit rates. The JPEG2000 compression algorithm consists of several steps which is depicted in [16]: tiling, color mapping, preprocessing, 2 Dimensional Forward Discrete Wavelet Transform (2D-FDWT), quantization, and Embedded Block Coding with Optimal Truncation (EBCOT).

In JPEG2000, the introduction of tiling represents a different strategy from the 8x8 block splitting method used in the original JPEG standard. In this stage, the image is divided into non-overlapping blocks, or tiles, with dimensions that are powers of two. This approach minimizes memory requirements during the compression process. However, there is a trade-off: reducing the size of each tile can compromise image quality due to boundary discontinuities between adjacent tiles.

After undergoing color mapping and preprocessing, steps that are consistent with the original JPEG standard, the image is subjected to a 2D Forward Discrete Wavelet Transform (FDWT)—the cornerstone of the JPEG2000 algorithm. Unlike Discrete Cosine Transform (DCT), FDWT employs wavelets, which are particularly efficient at representing non-stationary signals due to their

adaptive time-frequency windows. Their energy compaction efficiency in the frequency domain is demonstrated to be superior [2].

In FDWT, the image is divided into four distinct subbands: LL, HL, LH, and HH. The LL subband located at the top-left corner captures low-frequency components in both vertical and horizontal directions. Conversely, the HH subband at the bottom-right corner represents high-frequency components along both axes. The HL and LH subbands, situated at the upper-right and bottom-left corners respectively, contain high frequencies in one direction and low frequencies in the other.

Subsequent decompositions can be recursively applied to the LL subband to achieve multiple levels of decomposition. This concentrates energy into the LL subband at each successive level. The depth of this recursive decomposition is governed by the desired compression rate; a higher compression requirement necessitates more decomposition levels. The decomposition filters employed in JPEG2000 are rigorously examined in [17], where the mathematical characteristics such as Riesz bounds, order of approximation, and regularity properties—namely Holder and Sobolev—are elucidated in a comprehensive manner. Notably, the CDF 9/7 filter distinguishes itself due to its near-orthonormal attributes. However, this attribute may paradoxically serve as a detriment to its asymptotic performance, particularly when compared against other wavelet filters with four vanishing moments.

In JPEG2000, the quantization process employs a scalar quantizer, which converts the coefficients of each subband into integer multiples of a specific quantization step size. Unlike JPEG, where a single quantization table is used, JPEG2000 allows for tailored quantization steps for each subband and each level of decomposition. These customized step sizes are encoded directly into the code stream, offering a finer level of control over the quantization process.

Finally, the quantized wavelet coefficients are encoded using the Embedded Block Coding with Optimized Truncation (EBCOT) algorithm, which comprises two stages: Tier-1 and Tier-2. The algorithm is investigated deeply in [18]0. However, it proposed new compression algorithm and it's not used here.

In Tier 1, each subband is initially partitioned into code blocks. Using entropy coding theory, these code blocks are processed bitplane by bitplane, starting from the Most Significant Bit (MSB) down to the least Significant Bit (LSB). Each coefficient has an associated "significance" identifier, and within each bitplane, coding passes are allocated from a set of three possible passes to each bit, each pass with its own set of coding operations. The outcomes from Tier-1 are then fed into

an MQ arithmetic coder for further encoding. The resulting code streams from individual code blocks are amalgamated into a packed stream.

JPEG2000 offers several unique scalability features. It provides resolution scalability, facilitated by the specific resolution allocation for each code block. Additionally, it offers spatial scalability, as each code block impacts only a designated spatial area of the image.

In Tier 2, these encoded data are sorted into different layers, with each layer consisting of several consecutive sub-bitplanes. Thanks to this layered structure, JPEG2000 can offer rate-distortion scalability and supports progressive image transmission.

In summary, the EBCOT algorithm endows JPEG2000 with multiple forms of scalability, including resolution, spatial, and rate distortion, while also enabling progressive image transmission. JPEG2000 also represents a new way of interacting with the server. If browsing the image remotely as described in [19]. It uses a JPIK framework which is a connection-oriented network communication protocol using TCP, and optionally UDP, for the underlying network transport. It can flexibility process the uploaded and download JP2 file as required which JPEG doen't have.

1.5 Novel implementation

The JPEG2000 algorithm is much more complex than JPEG algorithm. There are several proposed architectures for reference.

As shown in [20], it's an overall design that shows the full hardware design of wavelet module, bit plane coding module, arithmetic coders and memory interfacing module between the coders. Where the Discrete Wavelet Transform (DWT) processor structure is really useful in help of design the DWT module. Besides, they also provide the performance measurement of their design which is a good point for referencing.

[21] shows a competitive design of the DWT module which gives a 35 times speed up comparing with the software computation. However, it's design consists of Cohen–Daubechies–Feauveau wavelet 9tap / 7tap wavelet (CDF 9/7) and Cohen–Daubechies–Feauveau wavelet 5tap / 3tap wavelet (CDF 5/3) filters where CDF 5/3 is not used in this project.

2

Algorithm and software verification

Contents

This chapter discusses the software implementation of the JPEG2000 image compression algorithm and compares its various performance metrics with those of the JPEG algorithm. Due to time constraints and limited personal expertise, this study focuses solely on the replication of the DWT algorithm and its comparison with DCT.

2.1 JPEG VS JPEG2000

The DWT (Discrete Wavelet Transform) compression algorithm in JPEG2000 was initially researched by the JPEG expert group starting in 1997 and eventually became the new generation standard surpassing the JPEG compression algorithm in 2000. Compared to the JPEG algorithm,

JPEG2000 employs different strategies in transformation, quantization, and entropy coding processes. DWT was ultimately adopted by the JPEG expert group as the transformation algorithm for JPEG2000.

As mentioned in [22], the information loss in DWT is smaller than in DCT. To verify this, Matlab is used to simulate the scenario of the image compression process. In Matlab, there are writing functions that can store the matrix as various forms of image files including JPEG, TIFF, BMP, GIF, PNG, and so on. The original BMP image file will first be loaded and stored the image into JPEG and JPEG2000 formats respectively by specifying the parameters of the two functions, the Output image is controlled to have relatively the same compression ratio. Under this situation, the image will be loaded again to see the PSNR and SSIM compared with the original image.

2.1.1 Metrics

The parameters used to compare the performance and quality of images are PSNR and SSIM which is the mostly commonly used metrics used in image quality measurement as explained in [23]. It analysis two parameters which are PSNR and SSIM and find the relationship between the two parameters using mathematical model. PSNR represents the the ratio between the maximum signal power over error power and it's normally expressed in decibel form. The equation is shown below:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2. \tag{2.1}$$

$$PSNR = 10 \cdot \log_{10} \left(\frac{MAX_I^2}{MSE} \right) \tag{2.2}$$

MSE represents the mean square error which calculates the square of each pixel's error and sums them all by dividing it by the number of pixels in the image. The results are then converted into decibels in equation 2.2. The larger the MSE is, the better the quality of the resulting compressed image. Theoretically, if the image is exactly the same as the original, the PSNR is infinite. The Matlab code is shown below in fig2.1.

The Structural Similarity Index (SSIM) is a metric used to evaluate the degree of similarity between two images. It is composed of three key components: brightness comparison, contrast comparison, and structural comparison. Each of these elements generates a similarity index that ranges from 0 to 1. The final SSIM value is calculated by multiplying these individual indices together which also ranges between 0 and 1. The SSIM provides an intuitive measure of image

PSNR.m

```
function y = PSNR(x1,x2)
% This function is used to calculate the PSNR between two input, x1 as the
% reference signal and x2 as the actual signal
% calculate the MSE
[R,C] = size(x1);
size_array = R*C;
y = double(x1 - x2);
y = y.^2;
MSE = sum(y,'all')/size_array;
y1 = 10 * log10(255^2/MSE);
y2 = 20 * log10(255/sqrt(MSE));
y3 = 20 * log10(255) - 10 * log10(MSE);

y=y1;

end

```

Figure 2.1: The Matlab code for PSNR module

quality: the closer the value is to 1, the more similar the assessed image is to the reference image. The equation is shown below in eq2.3 :

$$SSIM(x,y) = [l(x,y)]^{\alpha}[c(x,y)]^{\beta}[s(x,y)]^{\gamma} \tag{2.3}$$

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \quad c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \quad s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \tag{2.4}$$

where: $\mu_x\mu_y$ and $\sigma_x\sigma_y$ are the mean and standard deviation of x and y. σ_{xy} is the covariance of x and y. C_1 C_2 C_3 are all constant for stabilizing the fraction. The Matlab implementation is shown below in fig 2.2 and fig 2.3:

After completing the definition of comparison parameters, It's time to begin to compare the compression capabilities of two different transformation algorithms on the same image. In Matlab, the imwrite function can save a data matrix as images in different formats. For example, imwrite can save a 128x128 matrix, where each element is 8-bit unsigned data, as BMP, GIF, JPEG, JPEG2000, PNG, and TIFF formats. For different data formats, imwrite allows you to change its default compression parameters by adding additional arguments, achieving the desired compression specifications.

The original Lena picture is firstly stored in tif format which is a non-compressed picture with a size of 512*512 shown in fig 2.4. it's loaded by the read function. The loaded function is then stored in JPEG and JPEG2000 format. The compressed image in each format is then again loaded by the read function and compared with each other by calculating its corresponding PSNR and

SSIM.m

```
function re=SSIM(X,Y) %return value range from 0 to 1, The higher, the similiar
        X = normalize01(X);
        Y = normalize01(Y);

        X=double(X);
        Y=double(Y);

        ux=mean(mean(X));
        uy=mean(mean(Y));

        sigma2x=mean(mean((X-ux).^2));
        sigma2y=mean(mean((Y-uy).^2));
        sigmaxy=mean(mean((X-ux).*(Y-uy)));

        k1=0.01;
        k2=0.03;
        L=255;
        c1=(k1*L)^2;
        c2=(k2*L)^2;
        c3=c2/2;

        l=(2*ux*uy+c1)/(ux*ux+uy*uy+c1);
        c=(2*sqrt(sigma2x)*sqrt(sigma2y)+c2)/(sigma2x+sigma2y+c2);
        s=(sigmaxy+c3)/(sqrt(sigma2x)*sqrt(sigma2y)+c3);

        re=l*c*s;
end
```

Figure 2.2: The Matlab code for SSIM module

normalize01.m

```
function re = normalize01(img)
    a = min(min(img));
    b = max(max(img));
    re = (img-a)./(b-a);
end
```

Figure 2.3: The Matlab code for normalization module

SSIM and file size.

Before the comparison is conducted, a comparison of my own PSNR and SSIM function with the built-in function provided by Matlab is conducted. Through investigation of comparison of the Matlab function and my own module, It is noticeable that the result of built-in PSNR, SSIM functions are different from the result of my functions. The implementation of the comparison is shown in figure 2.5. The parameters of the two formats are set to be the easiest form. The compression ratio of JPEG2000 is set to be the lowest as well and the quality of JPEG is set to the highest to ensure the lowest error of two different functions. Other parameters of JPEG2000 are set to the simplest such as the reduction level (decomposition level) being 1, tile size being the

Figure 2.4: The original picture of lena

same as the image size and there is only one quality layer. According to the result presented in fig 2.6, The built-in function normally will reduce the PSNR by 2 to 3 db based on my own function. It's more reasonable to use the built-in function since it's more authoritative.

Having established the parameters and their meanings for comparison, the next steps involved data analysis and results evaluation. In JPEG, 'quality' is the only parameter influencing the compression ratio, as its lossy compression is tied only to the quantization table. On the other hand, JPEG2000 relies on a variety of factors for compression ratio control, including decomposition level, tile size, coding sections, and quality layers. This makes JPEG2000 more versatile but also more complex in terms of adjustable parameters. To draw a comprehensive comparison, each parameter for both formats was iterated through, and the most similar compression results were analyzed for performance. The original picture takes 424922 bytes in bmp format and testing implementation is shown in fig 2.7. The file_size in the figure stores the actual file size in bytes of each image quality. PSNR_JPEG and SSIM_JPEG store the PSNR and SSIM values for each quality as well. The compression ratio is iterated from 1 to 100. The final table is shown in 2.1.

The JPEG2000 has many factors which may affect the compression ratio and its quality. The Matlab provides compression ratio, reduction levels, tile size and so on. Basically, those parameters have determined most of the compression performance. In the following part, the PSNR, SSIM and their corresponding file size are investigated. The corresponding data is presented in Appendix

Quality	PSNR(dB)	SSIM	File size	Quality	PSNR(dB)	SSIM	File size
100	58.46	0.9965	162527	50	35.78	0.6738	20983
99	54.51	0.9914	152245	49	35.76	0.6732	20917
98	49.82	0.9755	131169	48	35.66	0.6673	20430
97	46.91	0.9552	116380	47	35.61	0.6638	20080
96	45.13	0.9355	103169	46	35.57	0.6630	19954
95	43.78	0.9149	91207	45	35.48	0.6587	19616
94	42.90	0.8985	81817	44	35.42	0.6560	19366
93	42.16	0.8831	73342	43	35.33	0.6522	18931
92	41.60	0.8704	66606	42	35.26	0.6499	18753
91	41.16	0.8594	62804	41	35.22	0.6472	18530
90	40.78	0.8486	59351	40	35.11	0.6418	18043
89	40.43	0.8392	55251	39	35.07	0.6404	17939
88	40.14	0.8315	52645	38	35.00	0.6375	17652
87	39.86	0.8233	49666	37	34.88	0.6326	17223
86	39.66	0.8178	48061	36	34.84	0.6304	17075
85	39.41	0.8094	45395	35	34.74	0.6261	16760
84	39.21	0.8039	43969	34	34.63	0.6202	16301
83	39.03	0.7988	42623	33	34.57	0.6175	16125
82	38.84	0.7923	40565	32	34.46	0.6132	15760
81	38.69	0.7885	39509	31	34.36	0.6099	15480
80	38.50	0.7807	37977	30	34.26	0.6047	15177
79	38.35	0.7763	36851	29	34.15	0.6003	14854
78	38.22	0.7714	35984	28	34.07	0.5965	14600
77	38.11	0.7680	34925	27	33.93	0.5907	14247
76	37.93	0.7599	33486	26	33.81	0.5868	13951
75	37.80	0.7559	32630	25	33.69	0.5804	13587
74	37.74	0.7548	32455	24	33.56	0.5751	13261
73	37.59	0.7465	31273	23	33.41	0.5687	12936
72	37.49	0.7434	30493	22	33.28	0.5648	12617
71	37.40	0.7401	29944	21	33.12	0.5578	12278
70	37.30	0.7368	29339	20	32.95	0.5495	11900
69	37.18	0.7311	28560	19	32.77	0.5424	11577
68	37.10	0.7284	28109	18	32.61	0.5367	11251
67	36.99	0.7230	27245	17	32.39	0.5271	10883
66	36.91	0.7197	26802	16	32.17	0.5176	10469
65	36.83	0.7176	26482	15	31.93	0.5080	10083
64	36.73	0.7134	25803	14	31.67	0.4975	9675
63	36.67	0.7111	25517	13	31.42	0.4860	9293
62	36.59	0.7070	24962	12	31.08	0.4733	8880
61	36.52	0.7052	24624	11	30.76	0.4593	8464
60	36.43	0.7011	24106	10	30.40	0.4439	8014
59	36.36	0.6983	23800	9	29.94	0.4218	7553
58	36.31	0.6962	23596	8	29.46	0.3998	7095
57	36.23	0.6922	23104	7	28.89	0.3765	6665
56	36.16	0.6891	22697	6	28.24	0.3510	6191
55	36.09	0.6855	22370	5	27.32	0.3148	5668
54	36.02	0.6834	22102	4	26.47	0.2773	5146
53	35.97	0.6823	21943	3	24.83	0.2339	4675
52	35.91	0.6792	21537	2	24.25	0.1966	4374
51	35.82	0.6742	21031	1	24.25	0.1963	4371

Table 2.1: Table for the data of JPEG image

main.m

```
%code for JPEG and JPEG2000 comparison
Input = imread('Origin.bmp');
% JPEG format         % 'Quality'range 0-100, higher with higher quality
imwrite(Input, 'D:\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG1.jpg' ...
    , 'Quality', 100);
% JPEG2000 format     % 'CompressionRatio', lower with higher quality
imwrite(Input, 'D:\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG20001.jp2', ...
    'CompressionRatio', 1, "Mode","lossy", ...
    "ReductionLevels",1,"TileSize",[512 512],"QualityLayers",1);

output1_jpeg = imread('D:\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG1.jpg');
output2_jpeg2000 = imread('D:\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG20001.jp2');

PSNR11 = PSNR(output1_jpeg,Input);% using own function
PSNR12 = psnr(output1_jpeg,Input);% using matlab function
PSNR21 = PSNR(output2_jpeg2000,Input);
PSNR22 = psnr(output2_jpeg2000,Input);
SSIM11 = SSIM(output1_jpeg,Input);% using own function
SSIM12 = ssim(output1_jpeg,Input);% using matlab function
SSIM21 = SSIM(output2_jpeg2000,Input);
SSIM22 = ssim(output2_jpeg2000,Input);

figure;
subplot(1,2,1),subimage(output1_jpeg);
title('lena JPEG');
subplot(1,2,2),subimage(output2_jpeg2000);
title('lena JPEG2000');
```

Figure 2.5: The Matlab code for self-defined PSNR SSIM function and built in function

Input	512x512 uint8
output1_jpeg	512x512 uint8
output2_jpeg2000	512x512 uint8
pic	512x512 uint8
PSNR11	65.5740
PSNR12	62.5349
PSNR21	58.4890
PSNR22	56.1500
SSIM11	1.0000
SSIM12	0.9998
SSIM21	1.0000
SSIM22	0.9987

Figure 2.6: The Result for self-defined PSNR SSIM function and built in function

A. For the decomposition level, it only supports 1 to 8 levels. Each of the levels is tested. The compression ratio can be set to an arbitrary number and the test starts from 1 to 100 which finally reduces the PSNR down to 20 dB. Further compression will be meaningless and there is no equivalent of JPEG. Tile size supports a minimum size of 128*128 and can be set as an arbitrary number. 128, 256 and 512 is chosen to compare. The iteration code of each case is shown in the fig 2.8.

```
% code for JPEG and JPEG2000 comparison
Input = imread('C:\document\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\Lena.tif');
file_size_jpeg = zeros(100,1);
PSNR_jpeg = zeros(100,1);
SSIM_jpeg = zeros(100,1);
for i= 100:-1:1
    imwrite(Input, 'C:\document\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG1.jpg' ...
    , 'Quality', i);
    output1_jpeg = imread('C:\document\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG1.jpg');

    file_info = dir('C:\document\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG1.jpg');
    file_size_jpeg(101-i) = file_info.bytes;
    PSNR_jpeg(101-i) = psnr(Input,output1_jpeg,255);
    SSIM_jpeg(101-i) = ssim(Input,output1_jpeg);
end
```

Figure 2.7: The JPEG data acquisition

```
% JPEG2000 format     % 'CompressionRatio', lower with higher quality
Input = imread('Lena.tif');
file_size_jpeg2000 = zeros(100,8);
PSNR_jpeg2000 = zeros(100,8);
SSIM_jpeg2000 = zeros(100,8);
%for k = 1:1:3
    for i = 1:1:8 % define the decomposition levels
        for j = 1:1:100 %define the compression ratio
            imwrite(Input, 'C:\document\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG20001.jp2', ...
                'CompressionRatio', j, "Mode","lossy", "ReductionLevels",i ...
                ,"TileSize",[128 128],"QualityLayers",1);
            output2_jpeg2000 = imread('C:\document\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG20001.jp2');
            file_info = dir('C:\document\Imperial_study\spring_term\Individual_Project_jpeg2000\WOKSPA\Comparison\JPEG20001.jp2');
            file_size_jpeg2000(j,i) = file_info.bytes;
            PSNR_jpeg2000(j,i) = psnr(output2_jpeg2000,Input);
            SSIM_jpeg2000(j,i) = ssim(output2_jpeg2000,Input);
        end
    end
%end
```

Figure 2.8: The JPEG2000 data acquisition

2.1.2 Data analysis

When examining the PSNR values for JPEG images at varying compression ratios, some interesting observations can be made. As depicted in Figure 2.9, with a high-quality factor around 99 or 98, the PSNR values hover around 54 dB. In contrast, at a low-quality factor setting of 1 or 2, the PSNR dramatically drops to approximately 24 dB. This represents a substantial information loss of nearly 30 dB when comparing the least compressed images to the most compressed ones. The tendency of the three parameters are all monotonically decreasing which is expected.

Turning our attention to the SSIM metrics, as shown in Figure 2.10, the SSIM index initially exhibits a modest decline as the 'quality' parameter decreases. However, this decline becomes more pronounced when the 'quality' drops below 10. Unexpectedly, the lowest SSIM value reached down to 0.196, with a minimum of 0.1963 observed at a 'quality' setting of 1, indicating bad similarity between the original and compressed images. The reason why the SSIM reduces dramatically in the low-quality case is that the more it is compressed, the larger the quantization step size of the quantization table is used in the quantization step which may eliminate some significant coefficients

in low frequency, therefore introducing more errors compared to the high quality situation.

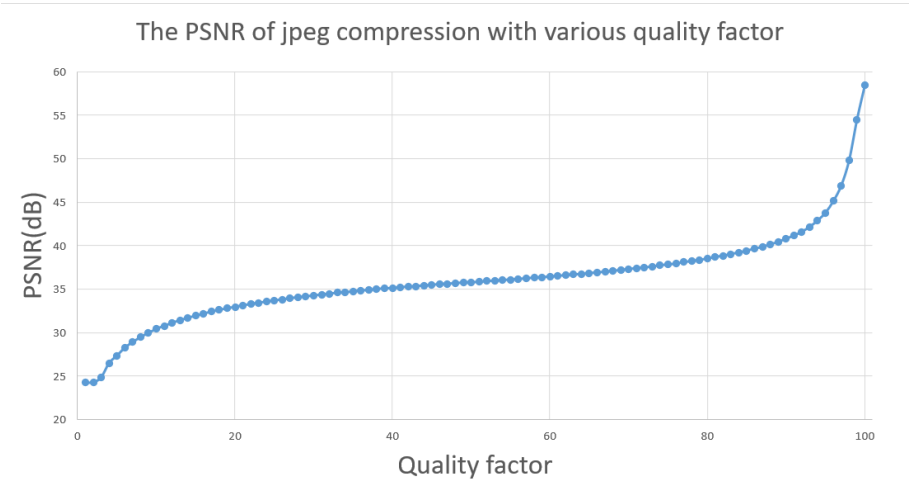


Figure 2.9: PSNR for JPEG standard with different quality

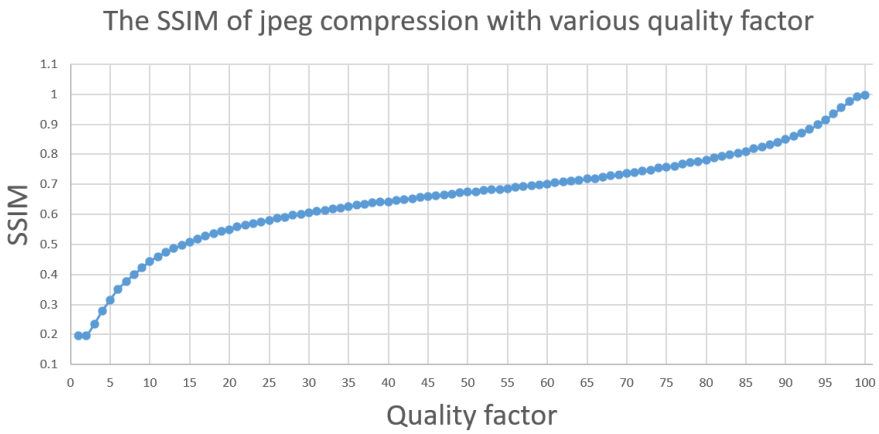


Figure 2.10: SSIM for JPEG standard with different quality

Regarding file size organized in fig 2.11, it's intuitive that lowering the quality of the image would correspondingly reduce the storage space required. In our experiments, the most compact file recorded was 4371 bytes, attaining an impressive compression rate of 61.45, the highest achieved in this study. Interestingly, the file size decreasing speed exhibited a diminishing rate when the quality decreased. The reason why the rate of change shows this phenomenon is that when the quality factor decreases, further compression can only drive the few non-zero high-frequency coefficients to zero which is not as obvious as the case at the beginning of the decrease of quality factor. Therefore, further compression will not change the distribution of zero coefficients so much.

As illustrated in Figure 2.11, there is an intuitive correlation between the reduction in image quality and the consequent decrease in required storage space. Within the scope of our experimental investigations, the most compact file size achieved was 4371 bytes, resulting in an exemplary

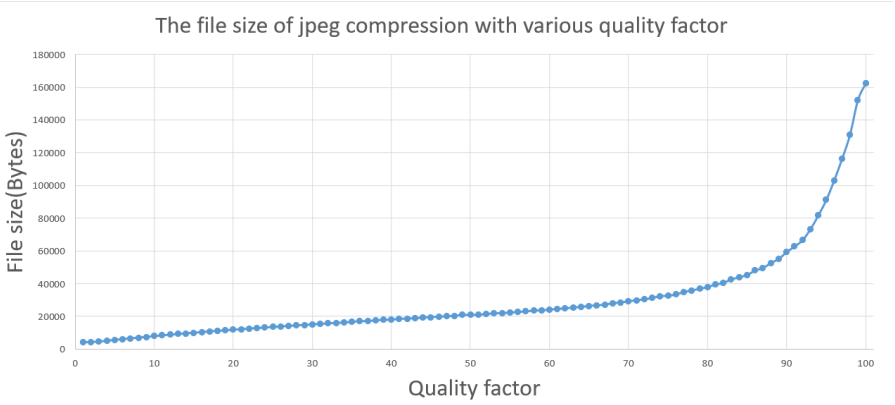


Figure 2.11: File size for JPEG standard with different quality

		512								256								128							
	DL	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
CR	91	2917	2947	2905	2776	2948	2769	2789	2806	2962	2932	2937	2945	2964	2926	2965	2949	2802	2953	2949	2963	2947	2964	2964	2965
	92	2917	2899	2905	2776	2743	2769	2789	2806	2611	2932	2908	2928	2934	2926	2932	2916	2802	2882	2932	2934	2931	2926	2926	2930
	93	2566	2899	2899	2776	2743	2769	2789	2806	2611	2875	2882	2880	2880	2884	2872	2880	2802	2882	2898	2901	2894	2891	2897	2901
	94	2566	2850	2867	2776	2743	2769	2789	2806	2611	2497	2745	2868	2872	2857	2872	2869	2802	2867	2864	2831	2867	2860	2873	2870
	95	2566	2813	2735	2776	2743	2769	2789	2806	2611	2497	2745	2840	2843	2824	2836	2821	2802	2772	2816	2831	2842	2843	2844	2838
	96	2566	2813	2735	2776	2743	2769	2789	2806	2611	2497	2745	2799	2783	2788	2811	2805	2802	2772	2795	2810	2814	2810	2809	2805
	97	2566	2435	2735	2776	2743	2769	2754	2771	2611	2497	2745	2782	2783	2777	2777	2779	2763	2772	2785	2756	2778	2782	2784	2786
	98	2566	2435	2735	2741	2743	2734	2754	2722	2611	2497	2745	2749	2747	2759	2735	2757	2693	2754	2750	2756	2759	2753	2747	2758
	99	2566	2435	2648	2696	2708	2731	2706	2722	2611	2497	2649	2720	2722	2730	2732	2622	2693	2724	2720	2729	2731	2729	2729	2729
	100	2566	2435	2648	2696	2663	2686	2706	2704	2611	2497	2649	2695	2689	2688	2578	2622	2693	2706	2699	2701	2695	2705	2692	2703

Figure 2.12: The file size comparison with various decomposition level and tile size

compression ratio of 61.45, the highest recorded in this study. Intriguingly, the rate at which file size decreases exhibits a diminishing return as the quality factor is further reduced. The underlying rationale for this observed phenomenon is attributable to the finite number of non-zero high-frequency coefficients that can be driven to zero as the quality factor diminishes. Unlike the initial stages of quality reduction, further compression at lower quality factors has a less pronounced impact on the distribution of zero coefficients.

The corresponding diagram of JPEG2000 is shown in figure **??**, **??** and **??**. The tendency is clearly that all the parameters are gradually decreasing as the CR decreases and the decomposition level 1 has the worst performance in all parameters. The tendency of decline is gradually slowing down as the CR gets higher. Except for the decomposition level 1, the PSNR of all other DL are higher than 20 dB. For SSIM, for all combination, the SSIM are higher than 0.65 which is significantly higher than that of JPEG. File size can reach down to 2611 bytes which is not feasible for JPEG. To sum up, the SSIM and file size of the JPEG2000 standard is much better than that of JPEG. For PSNR, cross comparing is still needed to investigate.

If cross-comparing the compression performance for the same JPEG2000 format but with different tile sizes provided in appendix A, it is obvious that as the tile size approaches the original size of the image, the compression performance with the same parameters has a very clear tendency. For example, the cases whose compression ratio ranges from 91 to 100 were picked and compared.

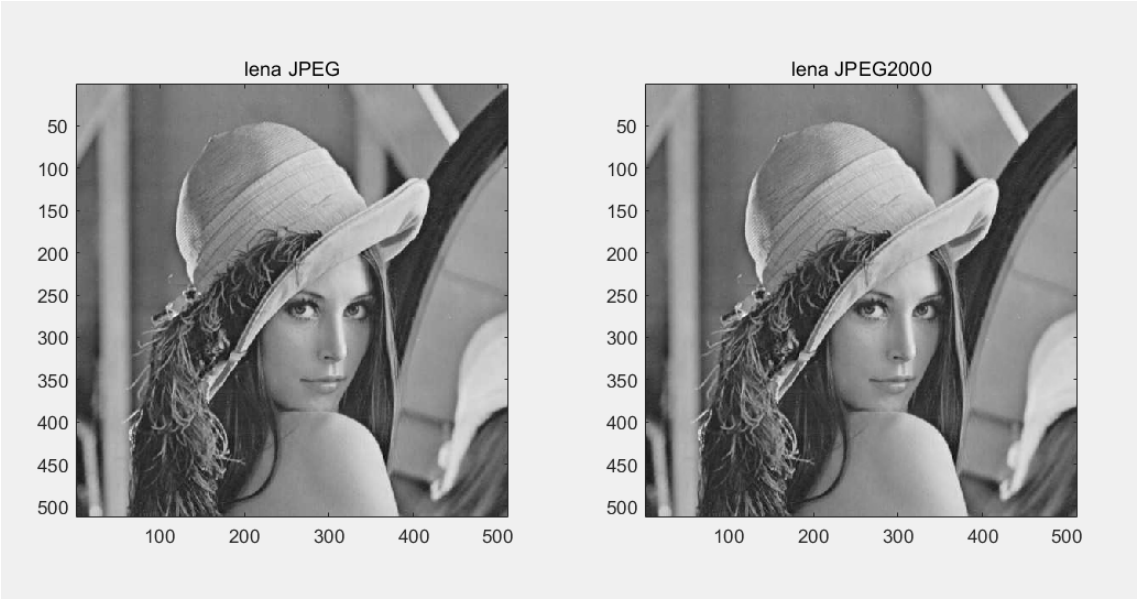


Figure 2.13: The low compression rate comparison

The file size decreases as the tile size increases as demonstrated in fig 2.12. It is intuitive that if the tile size of the image is larger, the better quality and smaller size of the reconstructed image since there is no edge effect produced from tiles and a smaller header is required. The necessary header is more complex for JPEG2000 than JPEG which may add extra markers to specify the information which causes the increase of file size.

To assess the comparative performance of JPEG and JPEG2000 in terms of PSNR and SSIM, images compressed at similar rates for both formats were selected. The compression settings were carefully chosen to minimize the file size difference: a compression rate of 7, a decomposition level of 5, and a tile size of 512×512 were employed for JPEG2000, while a quality level of 79 was applied for JPEG. Both settings resulted in identical file sizes. The obtained PSNR and SSIM values were 40.98 dB and 0.9678 for JPEG2000, and 38.35 dB and 0.7763 for JPEG, respectively. These findings are visually represented in Figure 2.13.

While the PSNR values for both JPEG and JPEG2000 are closely aligned, there is a significant discrepancy in SSIM, with JPEG2000 outperforming JPEG by approximately 0.2. At lower compression levels, JPEG2000 holds a marginal advantage, evidenced by slightly higher PSNR and SSIM values. However, the absence of noticeable artifacts in JPEG-compressed images suggests that both formats offer comparable visual quality under these conditions. Nonetheless, JPEG2000 exhibits a clear advantage in SSIM, indicating a closer resemblance to the original image, despite the PSNR values for both standards being relatively similar.

Figure 2.14: The middle compression rate comparison

For a medium compression rate in the context of the JPEG standard, a quality setting of 57 was selected. The closest matching parameters for JPEG2000 involved a compression ratio of 11, a decomposition level of 1, and a tile size of 512×512. The file size discrepancy between the two was a mere 5 bytes. The JPEG image yielded a PSNR of 36.23 dB and an SSIM of 0.6922, while the JPEG2000 image exhibited a PSNR of 36.98 dB and an SSIM of 0.9200. These results indicate a diminishing PSNR performance gap between JPEG and JPEG2000 as compared to the previous low compression rate scenario. Both images are illustrated in Figure 2.14.

While it may be challenging to discern visual differences based solely on PSNR, closer inspection of the noise regions reveals distinct artifacts in the JPEG format. This is evident when zooming into the specified areas, as shown in Figure 2.15. The JPEG standard displays discernible block patterns at the boundaries of each 8×8 block, causing discontinuities. In contrast, JPEG2000 demonstrates smoother transitions and is devoid of such block effects, corroborating the observed differences in SSIM performance.

If a high compression rate case is selected, for example, the quality of 1 in JPEG. the corresponding JPEG2000 compression parameters have a compression rate of 57, a decomposition level of 4 with a tile size of 512 * 512. The file size difference is 0 bytes. Compared to the PSNR and SSIM difference, the JPEG2000 image has PSNR SSIM 7 dB and 0.6484 more than the JPEG image respectively. The brief compressed image of two standards is given in the fig2.16. The JPEG reconstructed image quality in PSNR and SSIM is much lower than that of JPEG2000 with artifacts for each block. Even with almost the same file size, the image quality can differ

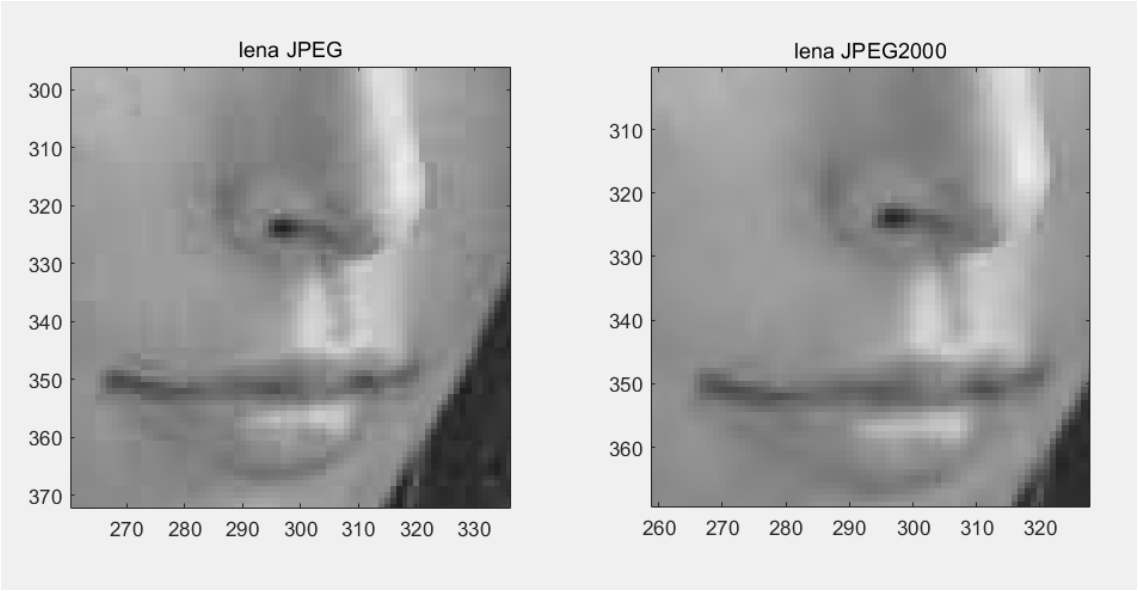


Figure 2.15: The middle compression rate comparison in detail

significantly. That's the advantage of JPEG2000 at a high compression rate. Beyond that, the JPEG2000 can still compress it further which is the ability that JPEG doesn't possess. From this, it is concluded that the JPEG2000 is better than JPEG in the high middle low compression rate that JPEG possesses and the advantages are more obvious in high compression cases.

2.2 Overview of JPEG2000

To fully design the hardware of the JPEG2000 compression standard, it is necessary to understand the principle of JPEG2000. Let the input image be set to 512 times 512 grayscale Lena image as shown in fig 2.4. In JPEG2000, the entire process includes the following parts: 1) Image preprocessing. 2) 2D DWT (Discrete Wavelet Transform). 3) Quantization. 4) EBCOT (Embedded Block Coding with Optimal Truncation) encoding. 5) Bitstream organization, as shown in Figure fig 2.20. Among them, FDWT (Forward Discrete Wavelet Transform) and EBCOT encoding are the two most critical compression steps as both these two steps significantly shorten the bit required to represent the image. In the following chapter, those steps are introduced in detail.

2.3 Preprocess

The preprocessing step has many sub-steps, which include image tiling, DC level shifting, and Multiple component transformation. Among these, image tiling and Multiple component trans-

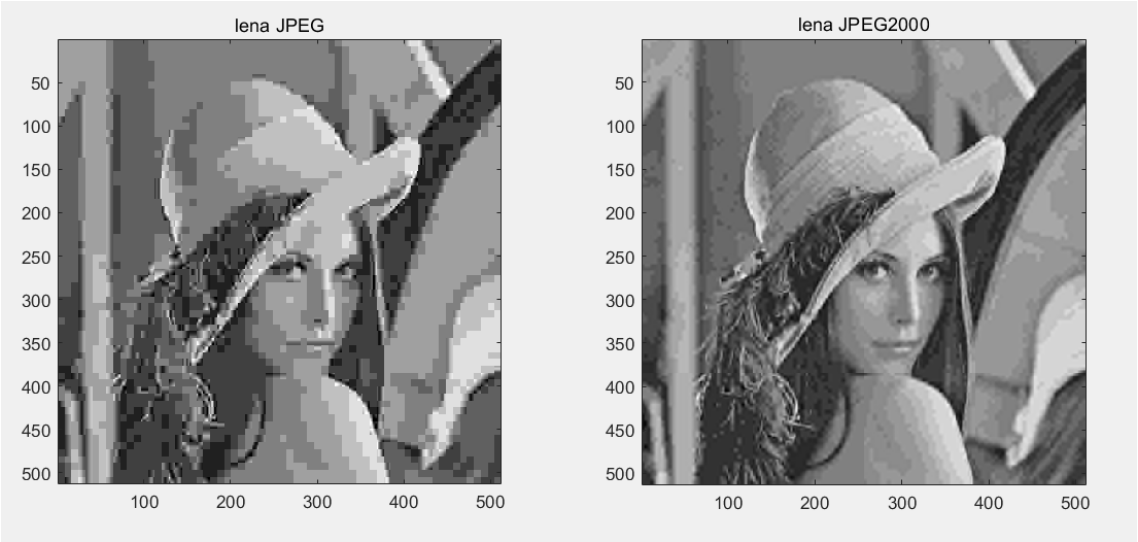


Figure 2.16: The high compression rate comparison

formation are optional, as shown in Figure fig 2.21. The details of each step are described as follows.

Image tiling refers to the partitioning of the original image into multiple processing blocks of the same size, known as tiles. Each tile is processed independently in subsequent steps, and there is no overlap between tiles, ensuring their independence. If an offset exists, the edge tiles may have different sizes to adapt to the reference grid. The size of a tile can range arbitrarily from 1 to $2^{32}-1$. In Figure 2.21, the original image with dimensions 512×512 is divided into four equivalent tiles, each having a size of 256×256. The image can also be partitioned into other sizes, such as 16 tiles of 128×128 or 64 tiles of 64×64. The minimum tile size is 1×1 and the largest is the size of the input picture.

DC level shifting is a method used to make the dynamic range of each component symmetric around 0. This reduces the dynamic range of the coefficients resulting from the Forward Discrete Wavelet Transform, which is beneficial for the encoding phase. The basic principle involves subtracting $2^{(R-1)}$ from each element in the image, where R represents the number of bits per pixel. For instance, if the input data is of the `uint8` data type, then $R = 8$ bits are used to represent a grayscale pixel. If multiple components are present, each one must undergo this process separately. As illustrated in the middle of Figure 2.21, each element in a 4×4 matrix is subtracted by $2^{(8-1)}$, which is 128. This operation ensures that the range of magnitudes for each pixel is symmetric around 0.

Multiple component transformation is optional depending on whether the input image is a

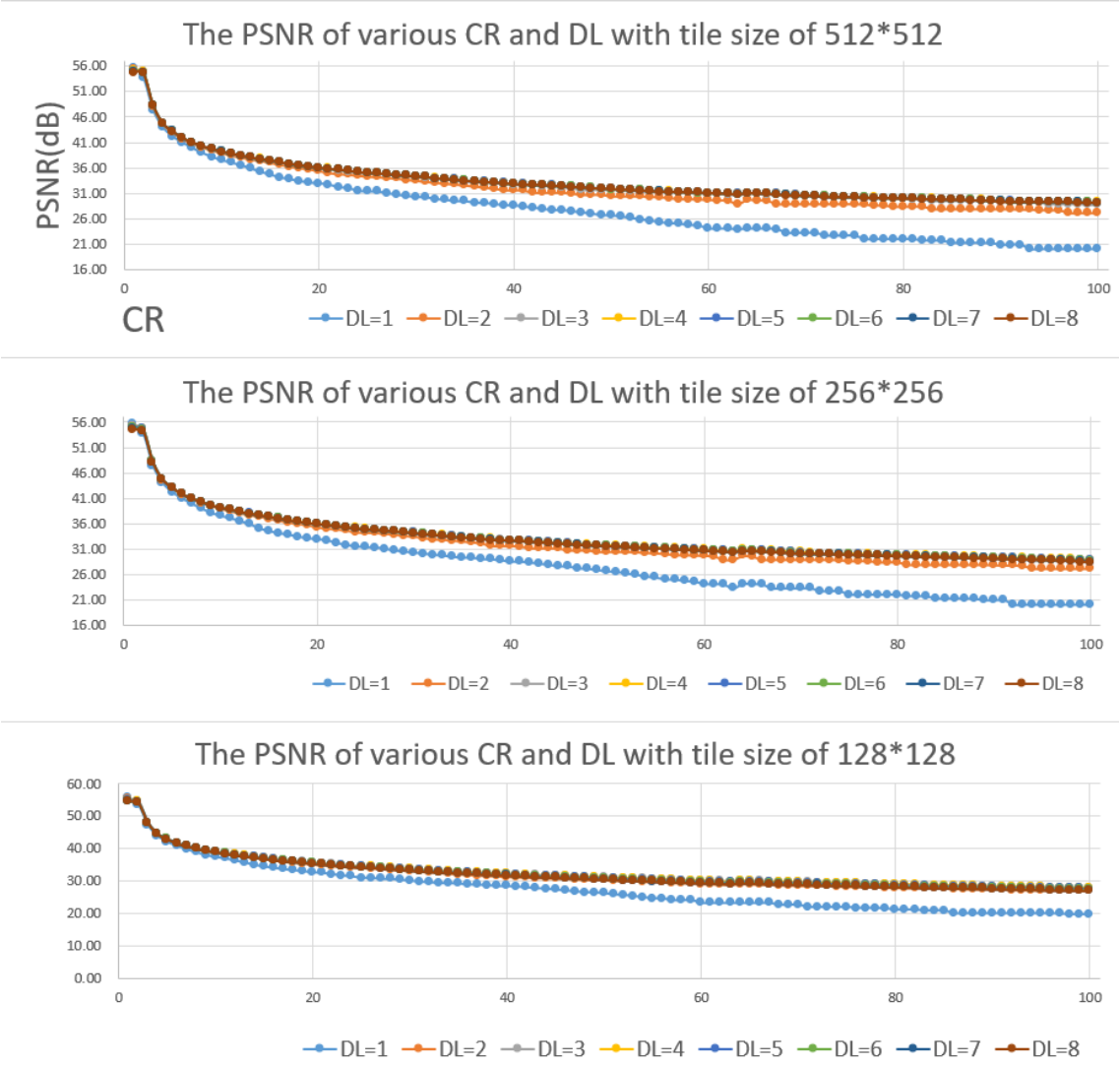


Figure 2.17: The PSNR of various CR and DL for JPEG2000

multiple components image or a grayscale image. It is default irreversible if lossy compression is chosen because the calculation is done in real number and the result is rounded in the quantization step. If lossless compression is used, this step will employ reversible component transformation. This project will focus on irreversible color transformation. If the input image is a grayscale image where there is only a grayscale component in each pixel, there is no need to map it to other color domains. However, if the input image is a multiple-component image, it is necessary to transform it into Y Cb Cr domain as stated in the introduction [6] that human eyes are more sensitive to brightness than color. As presented in the bottom of Figure 2.21, the transformation maps an RGB domain image to the Y Cb Cr domain. The Irreversible Color Transformation (ICT) mapping equation is presented below:

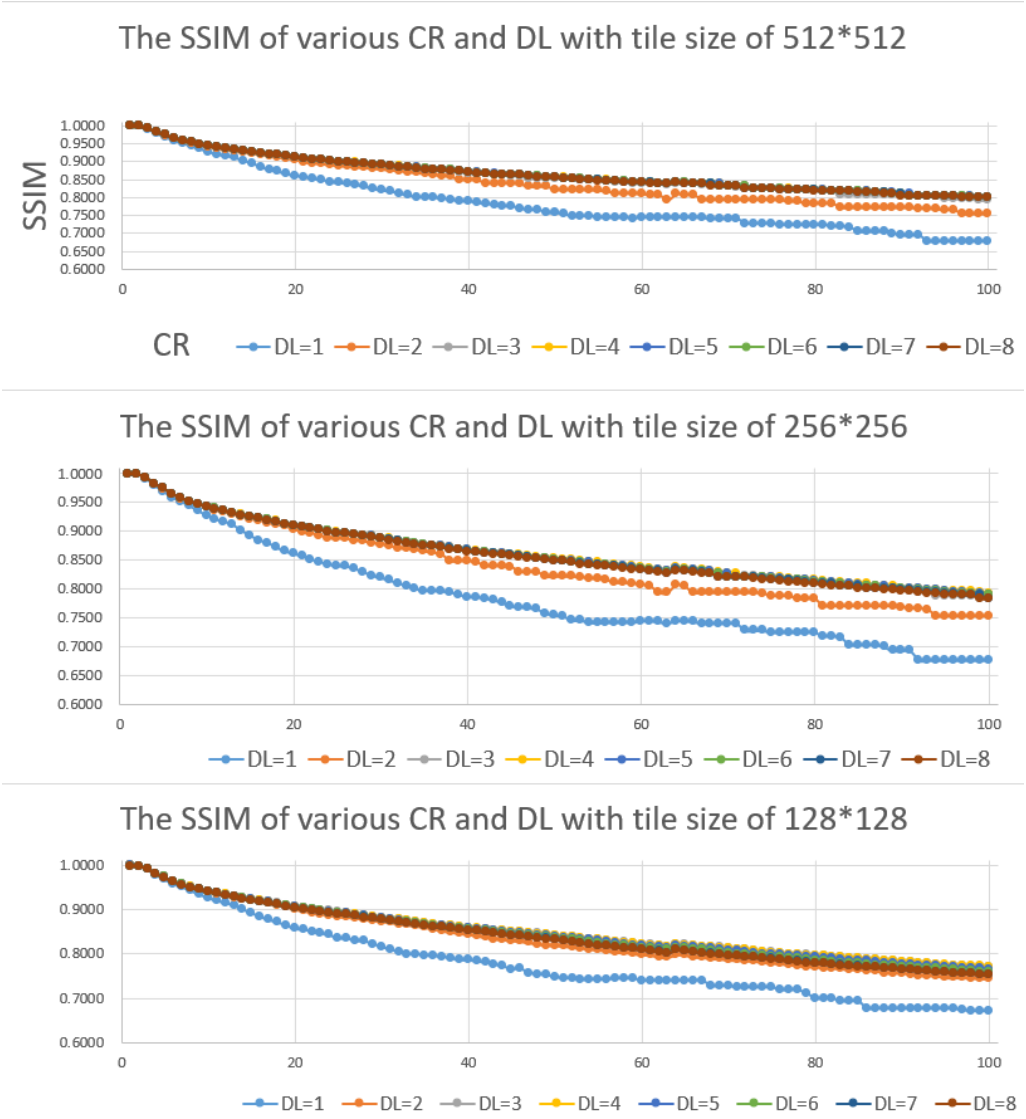


Figure 2.18: The SSIM of various CR and DL for JPEG2000

$$\begin{aligned} Y &= -0.299R - 0.587G + 0.114B \\ C_b &= -0.16875R - 0.33126G + 0.5B \\ C_r &= 0.5R - 0.41869G - 0.08131B \end{aligned} \tag{2.5}$$

Where the R, G, and B represent the RGB component magnitude of the specific pixel. Y, C_b, C_r is the YC_bC_r component magnitude of the corresponding pixel.

The following part is for the software verification of the Pre-processing part. In this experiment, Matlab is used to verify the preprocessing operation. This project is aiming to compress grayscale images in lossy mode. Therefore, the MCT process is not required. The rest of the process is implemented as follows.

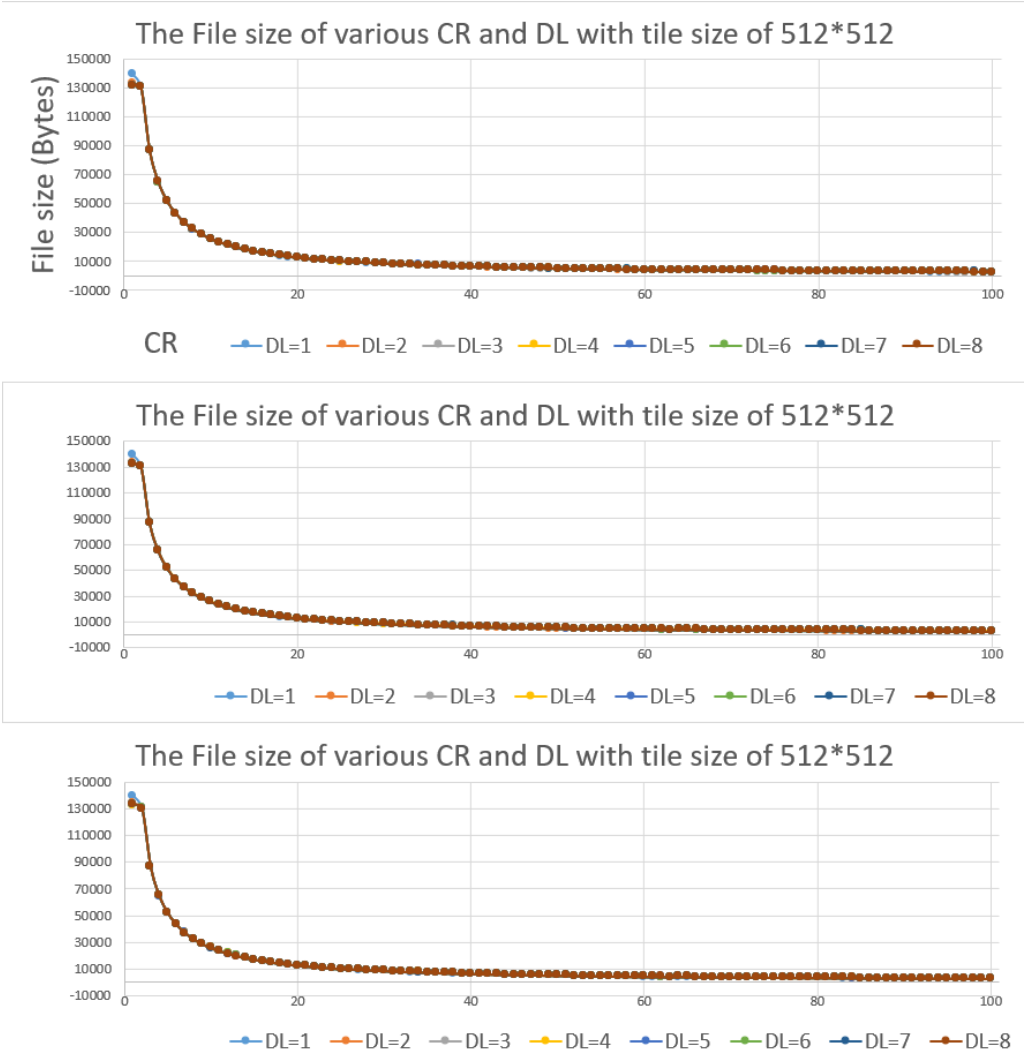


Figure 2.19: The File size of various CR and DL for JPEG2000

The image will be firstly read in by the imread function which is built in the Matlab library. For the choice of testing image, the standard Lena image is chosen which is commonly used in the image processing area as shown in 2.4 previously.

The image is firstly stored in BMP format BMP is an image format called Bitmap or DIB(Device-Independent Device) due to its non-transform-based image storage property. It will store the image information without any modification which is a significant RAW data source. The Lena image with size 512*512 is represented by grayscale color space which uses 8 bits to store the information of each pixel. Image tiling is employed to tile the image into many subblocks for the benefit of efficient memory demand. Besides, the DC level shifting is performed as well. The corresponding code for read-in, image tiling and DC level shifting is shown in figure 2.22, 2.23 and 2.24. In the tiling part, the input is an image called x and the number of the tile in a row or a column called

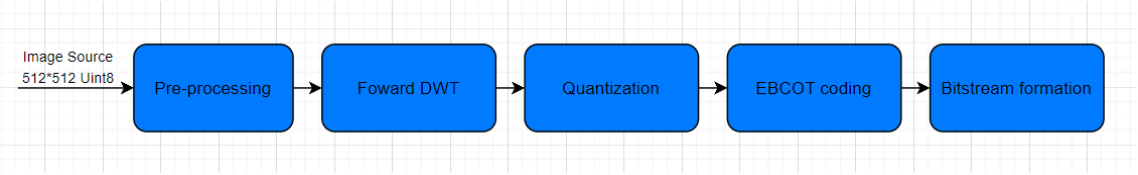


Figure 2.20: The overview of the whole process of JPEG2000

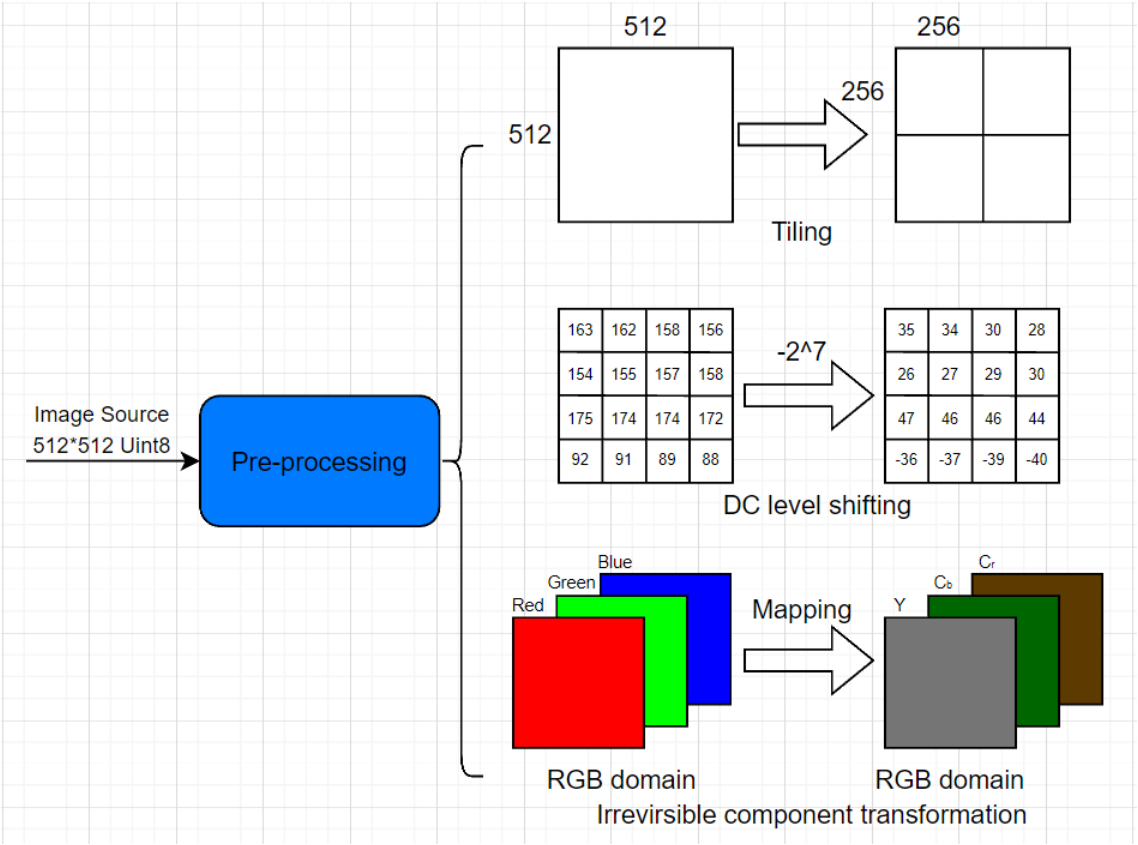


Figure 2.21: The preprocessing step details

num. To simplify the complexity, the input image and tile size are always set to square. Therefore, the total number of tiles is the square of num. Then the tiles are generated as a zero matrix in advance. The loop at the bottom is how data is allocated into each tile. In the DC level shifting part, the 128 is subtracted from the input x. There is also another way above which is used in some other open source program. However, it's not necessary to do the normalization at this stage therefore the bottom one is used. In the main code, the split image and whole image are both shifted and stored in x2 x3 separately.

main.m split.m

```
clear;
clc;

pic = imread('Lena','bmp');%     Image read
num = 2;
[x1]=split(pic,num);%            Image tiling, input image:pic, number of tiles: num^2
[x2]=ZZYnormalize(x1);%          tile normalization,  level shifting
[x3]=ZZYnormalize(pic);%         Whole picture normalization, level shifting

```

Figure 2.22: The image read, tiling and DC level shifting

main.m split.m

```
function [tile]=split(x,num)
% Tile the original image into small tiles
num_tile = num^2;% calculate the number of generated tiles.
[r,c] = size(x);%Find the length of the row and column of input image
R = ceil(r/num);%row number of tiles
C = ceil(c/num);%coloum number of tiles
TILE = zeros(R,C,num_tile);%Generate the tile blocks in advance

for i = 1:num % outer loop for row
    for j = 1:num % inner loop for column
        TILE(:,:, (i-1)*num+j) = x( (i-1)*C+1:i*C , (j-1)*R+1:j*R ); % Allocate the data into each tile
    end
end

tile = TILE;
end

```

Figure 2.23: The tiling module implementation

2.4 DWT Algorithm

This chapter will elaborate on the computational principles and details of DWT (Discrete Wavelet Transform). The foundation for the CDF 9/7 wavelet used in JPEG2000 was first proposed in the paper 'Biorthogonal bases of compactly supported matrix-valued wavelets' by Slavakis, K. and Yamada [24]. This paper laid out a mathematical framework for wavelets with biorthogonal bases and compact support properties. Subsequent research in 'Factoring wavelet transforms into lifting

main.m ZZYnormalize.m

```
function [gray]=ZZYnormalize(x)
% DC level shifting
%  GRAY=(double(x)-128)./256;
  GRAY=(double(x)-128);

gray = GRAY;
end
```

Figure 2.24: The DC level shifting module implementation

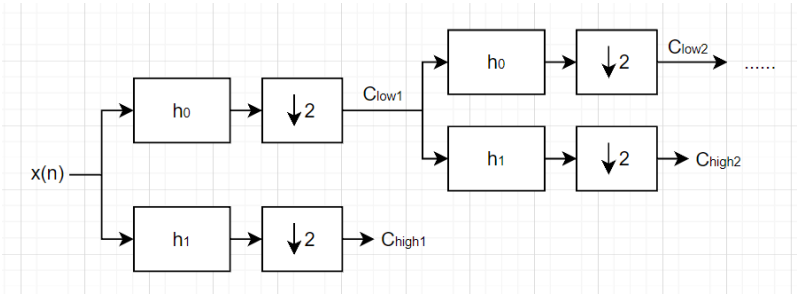


Figure 2.25: The filter bank used in convolutional based DWT implementation

steps' proposed transforming the CDF 9/7 into a lifting-based method, significantly simplifying the hardware implementation's difficulty and complexity. Therefore, there are two implementation methods for DWT: the convolutional-based method and the lifting scheme-based method. The convolutional-based method is intuitive, involving the convolution of the input sequence with the analysis filter bank to obtain the final result. The lifting-based scheme simplifies the required computational complexity by converting it into a series of lifting steps that does not involve convolutional operations, as proposed by Daubechies, Ingrid, and Sweldens, Wim.

Specific convolutional implementation methods are delineated as per the accompanying Figure 2.25. The incoming dataset undergoes a separate filtering process: one via a high-pass filter and another via a low-pass filter. The filter bank is the analysis filter bank from the 9-tap/7-tap Cohen-Daubechies-Feauveau convolutional filter bank mentioned previously. The convolutional based filter bank is presented in fig2.26. Subsequently, down-sampling is performed on both of these filtered datasets to obtain coefficients corresponding to high-frequency and low-frequency components. Each set of coefficients possesses a length equivalent to half that of the original dataset.

The high-frequency coefficients encapsulate the intricate details associated with rapid variations in the data, while the low-frequency coefficients signify the more gradual contour variations within the image. Through the utilization of DWT, a segregation of high-frequency and low-frequency elements in the image is achieved, enabling the concentration of energy within the low-frequency domain. Given that the human visual system exhibits heightened sensitivity to low-frequency contours and decreased sensitivity to high-frequency details, the omission of a subset of high-frequency details is unlikely to significantly compromise the perceptual interpretation of the image's content. This forms the underlying principle by which DWT facilitates image compression. In the diagram, the low-frequency coefficients obtained from the first pass through the low-pass filter are further decomposed on the right side. This yields high-frequency and low-frequency coefficients for the

n	Low-pass filter
0	0.602 949 018 236 360
±1	0. 266 864 118 442 875
±2	–0.078 223 266 528 990
±3	–0.016 864 118 442 875
±4	0.026 748 757 410 810
n	**High-pass filter**
–1	1.115 087 052 457 000
–2, 0	–0.591 271 763 114 250
–3, 1	–0.057 543 526 228 500
–4, 2	0.091 271 763 114 250

Figure 2.26: The filter bank used in convolutional based DWT implementation

second decomposition level. By repeatedly filtering the low-frequency coefficients, low-frequency information can be obtained at different scales. In JPEG2000, the number of decomposition levels can be adjusted to capture low-frequency information at various resolution levels. This allows for various compression performance.

The Convolutional based implementation is intuitive and due to its high computation complexity resulting from the real number addition and multiplication of 9 tap low pass filter and 7 tap high pass filter, it is not hardware friendly. To simplify the complexity of related hardware implementations, Ingrid Daubechies and Wim Sweldens proposed a wavelet transform implementation method based on the lifting scheme in 1998 [25]. In their paper, they provided the complete steps to decompose the convolution method into the lifting scheme. The specific steps for the lifting scheme are as follows.

The basic principle is shown in the figure 2.27. The lifting scheme method transforms the

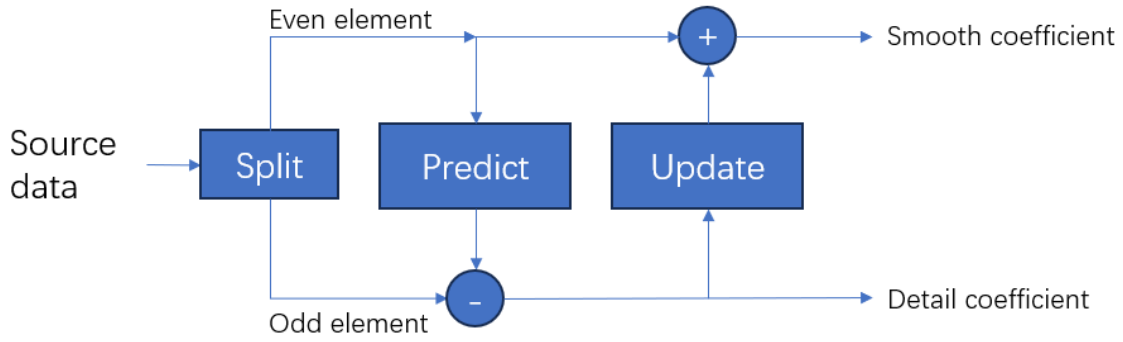


Figure 2.27: The basic principle of lifting based method

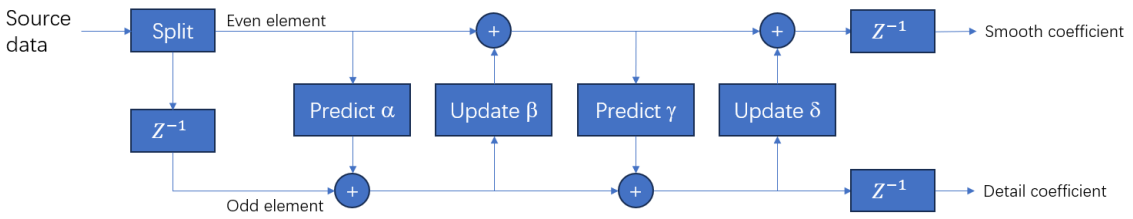


Figure 2.28: The lifting based process of CDF 9/7 wavelet

convolution into a set of predictions and updates. The input data will first be divided into odd and even sequences then, the even sequence is used to predict the odd elements in the early phase, and then the even element is updated using predicted odd elements. Therefore, the process of prediction and update is in order and there exists data dependence. The even sequence represents the low frequency components and the odd sequence represents the high frequency components.

The whole process of CDF 9/7 wavelet in the lifting scheme is shown in fig 2.28. The entire procedure includes two basic lifting processes, but different parameters are used in each process. After undergoing two prediction steps and two update steps, one iteration of the FDWT is com-

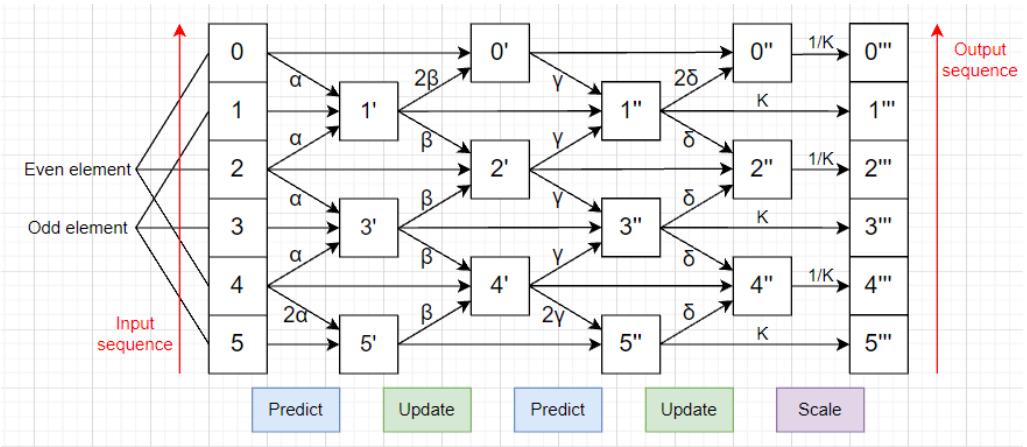


Figure 2.29: The detailed calculation process of CDF 9/7 lifting scheme

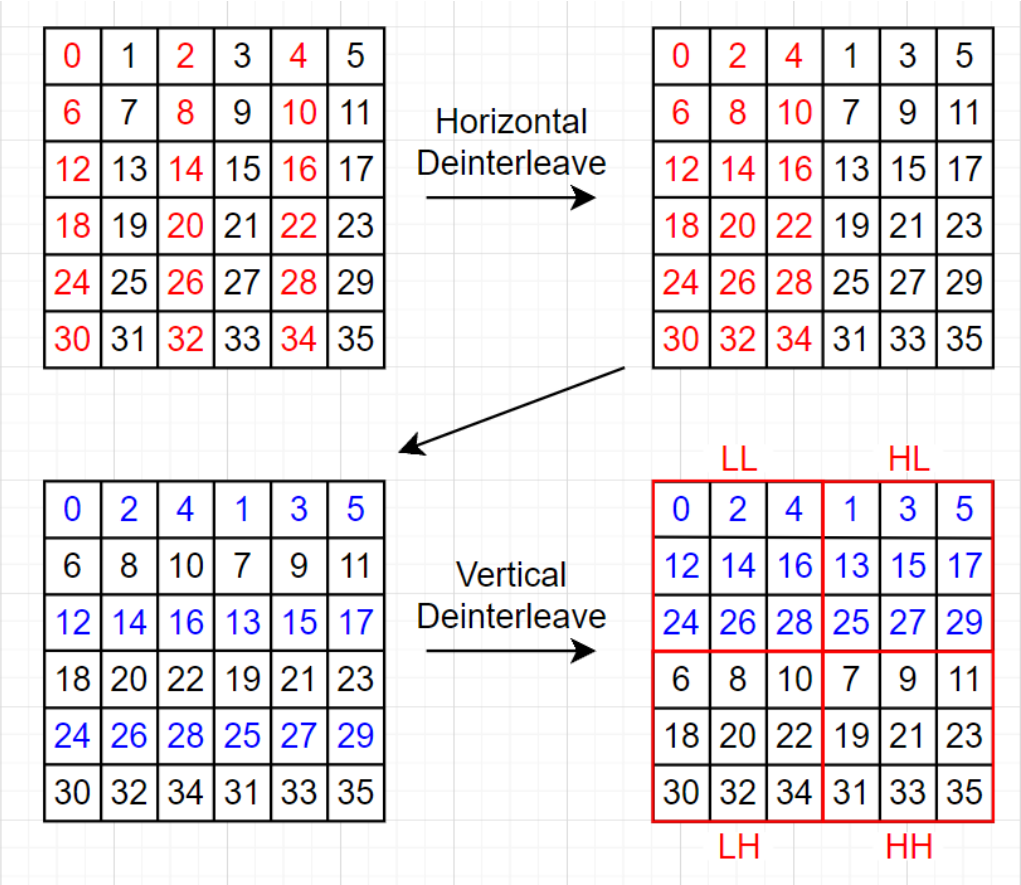


Figure 2.30: The deinterleave process after 2D DWT

pleted. The output includes the updated odd and even sequences. The odd sequence represents high-frequency information, while the even sequence represents low-frequency information. Just like before, the lengths of both the frequency coefficient sequences are half of the original length. The mathematical expression for the lifting scheme is shown below:

$$\begin{aligned} Y(2n+1) &= X_{ext}(2n+1) + \alpha(X_{ext}(2n) + X_{ext}(2n+2)) \\ Y(2n) &= X_{ext}(2n) + \beta(Y(2n-1) + Y(2n+1)) \\ Y(2n+1) &= Y(2n+1) + \gamma(Y(2n) + Y(2n+2)) \\ Y(2n) &= Y(2n) + \delta(Y(2n-1) + Y(2n+1)) \\ Y(2n+1) &= KY(2n+1) \\ Y(2n) &= (1/K)Y(2n) \end{aligned} \tag{2.6}$$

The detailed calculation process is shown in fig 2.29. A length of 6 sequences is used to demonstrate the DWT process. The even element is firstly added together and multiplied with α, the middle odd element between two even element is then added to it and result in 1',3',5'. One thing to notice is at

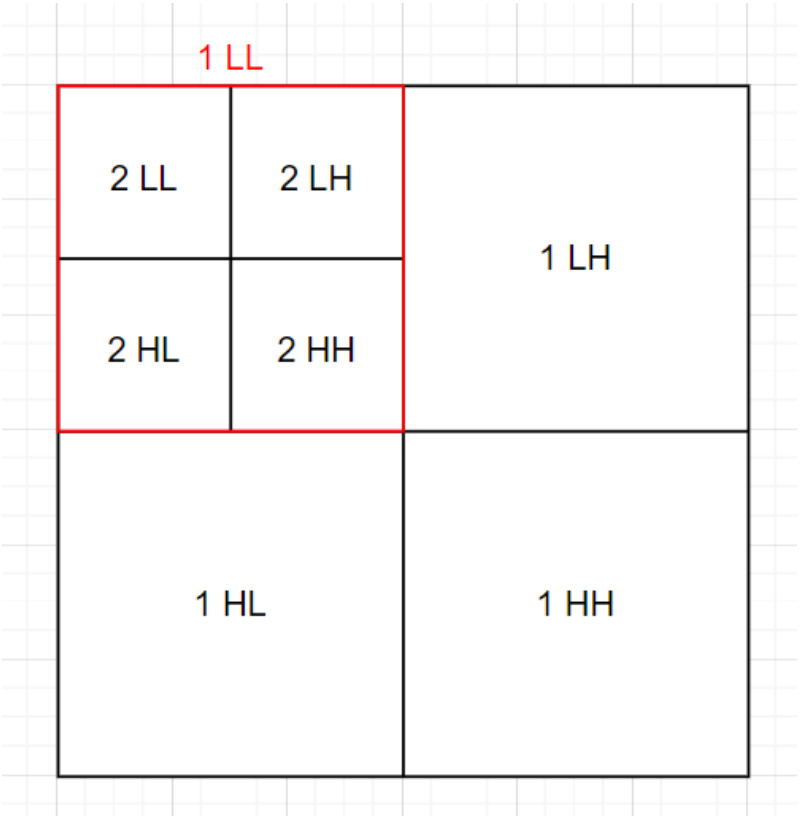


Figure 2.31: The subband allocation after 2D DWT decomposition twice

the end of the prediction step, the last odd element only has one even element around it, therefore, boundary handling is applied. As shown in the figure, the 4th element is multiplied by 2 α to imitate there are two same surrounding elements around the 5th element. Secondly, the predicted odd element is used to update the even elements. The same as the prediction process, two adjacent odd elements are added together and multiplied by β, then the original even elements are added and get the new even elements. The same as the previous case, the first even element only has one surrounding odd element. Therefore, the 1st element is multiplied by 2β to apply the boundary handling. The rest of the prediction and update is the same as previous with parameters changed to $\gamma \quad \delta$. After finishing two consecutive predictions and updates, the updated even elements are scaled by 1/k and odd elements are scaled by K to get normalized coefficients. Each element is also placed in order to form the output sequence as shown on the right of the fig 2.29.

For a still image, the Discrete Wavelet Transform (DWT) needs to be performed on each row and column in both the horizontal and vertical directions to complete the 2D DWT. Since the computed data are still arranged in an alternating sequence of odd and even numbers, Deinterleaving is necessary to gather the low-frequency data from both directions. Figure 2.30 illustrates this deinterleaving process. The number on each pixel is the index of input data. The horizontal

deinterleave puts the even columns on the front and odd columns on the back, the same as the vertical deinterleave, the even rows are put on the top and odd rows on the bottom.

After the deinterleave, the order is changed to the last diagram. The finalized deinterleaved image will have low frequency components in both horizontal and vertical directions on the top left corner. Therefore, it is called the LL subband. The bottom right corner is the HH subband because it contains high frequency components in both directions. The top right corner and bottom left corner are the HL and LH subbands separately. As mentioned previously, the low frequency component can be further decomposed into second level decomposition with level 2 LL, HL, LH, HH subband. If two decomposition level is employed, there are 7 subbands in total where 4 second level subband and 3 first level subband as shown in the fig 2.31. The more it decomposes, the more frequency scale it has, with which JPEG2000 gives more flexibility in compressing images.

Details on folders hierarchy

- src
  - lib
    - openjp2: contains the sources of the openjp2 library (Part 1 & 2)
    - openjpip: complete client-server architecture for remote browsing of jpeg 2000 images.
  - bin: contains all applications that use the openjpeg library
    - common: common files to all applications
    - jp2: a basic codec
    - jpip: OpenJPIP applications (server and dec server)
      - java: a Java client viewer for JPIP
    - wx
      - OPJViewer: gui for displaying j2k files (based on wxWidget)
- wrapping
  - java: java jni to use openjpeg in a java program
- thirdparty: thirdparty libraries used by some applications. These libraries will be built only if there are not found on the system. Note that libopenjpeg itself does not have any dependency.
- doc: doxygen documentation setup file and man pages
- tests: configuration files and utilities for the openjpeg test suite. All test images are located in openjpeg-data repository.
- cmake: cmake related files
- scripts: scripts for developers

Figure 2.32: The hierarchy of the openjpeg project

The software verification of DWT is presented as follows. As described above, The lifting based DWT is more friendly to hardware implementation and cost less computational resource. Therefore, the lifting scheme is implemented and compared with the open source library OPENJPEG, posted on Git Hub, to validate the correctness of the transform function. The function implementation is shown in appendix B. The beginning of the function is the definition of interface and parameters where the parameters for prediction and update are α, β, γ and Δ. The parameters for scaling are defined as K1 and K2 where K1 is for the even number and K2 is for the odd number.

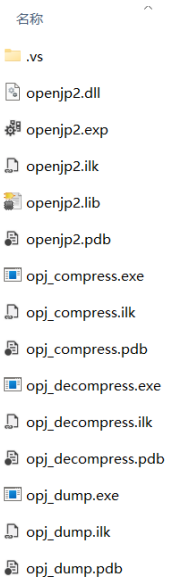


Figure 2.33: The executable file after build

The size of the input image and output matrix is obtained and generated in the following code. The R and C stores the length and width of the input image. y1 and z are the intermediate matrix for calculation and the output matrix. The lifting scheme as discussed before, will decompose into two 1D DWT in two directions. The first half code does horizontal 1D DWT. First, predict odd elements and fo the boundary handling, then update the even elements with boundary handling. The same flow is conducted again with different parameters. Finally, the odd and even elements are scaled by different scale factors. To simplify the deinterleaving operation, the data is firstly deinterleaved horizontally which looks like the top half of the operation shown in figure 2.30. The second half of the code is for vertical 1D DWT. The same as the horizontal 1D DWT, but the index is in the first place of the matrix which controls the row-wise operation. Finally, after scaling and the second half deinterleave. The output is given to output matrix z.

To validate my implementation, the most widely used open source project is employed. The source code contains many features and the hierarchy is shown in figure 2.32. The function of the DWT part is stored at src\lib\openjp2. The files dwt.c and dwt.h store the whole process of the DWT algorithm. Its corresponding codec is stored in bin\jp2. The environment used here is Visual Studio 2022. After downloading the source code, the codec application is complied with and tested to ensure it works correctly. Using the cmake built in the Integrated Development Environment (IDE) to generate the executable file as shown in the figure 2.33. The obj_compress.exe is the executable document. However, it is not executable if you double click it. Because it requires parameters when it starts. CMD or Power shell is used to add starting parameters. The official page

```
D:\Imperial_study\spring_term\Individual_Project_jpeg2000\OPENjpeg\openjpeg-
master\out\build\x64-Debug\bin>opj_compress.exe -ImgDir D:\Imperial_study\sp
ring_term\Individual_Project_jpeg2000\OPENjpeg -OutFor JP2 -n 2 -mct 0
Folder opened successfully

File Number 0 "data.csv"
skipping file...

File Number 1 "DATA_comparision.csv"
skipping file...

File Number 2 "data_Original.csv"
skipping file...

File Number 3 "data_PSNR.csv"
skipping file...

File Number 4 "openjpeg-master"
skipping file...

File Number 5 "openjpeg-master.zip"
skipping file...

File Number 6 "opj_脚本.txt"
skipping file...

File Number 7 "Origin.bmp"
[INFO] tile number 1 / 1
[INFO] Generated outfile D:\Imperial_study\spring_term\Individual_Project_jp
eg2000\OPENjpeg/Origin.JP2
encode time: 533 ms
```

Figure 2.34: The command line for ruining the program

of OPENJPEG provides the parameters needed to run the program. It contains the format of the path where the image to be compressed is stored, the input image characteristic, the compression ratio, the output quality in dB, the number of decomposition levels and so on. In the validation, only path and MCT and decomposition level are set just for simplicity. The command line and result are shown in figure 2.34. The output jp2 file is viewed using an online viewer as shown in figure 2.35 which validates the functionality of the program.

To test whether my own function is the same as the one implemented by OPENJPEG. The parameters are set the same as my own implementation. To simplify, the tile size is set to be the whole image which means there is only one tile and the default set in OPENJPEG is not tiling either. Also, the decomposition level is all set to 2 for simplicity. The command line is shown in figure 2.36. Where the -OutFor specifies where the image to be compressed is stored. -n is the decomposition level which is set to be 2. -mct is the configuration of MCT, 0 means there is no need to do color transformation. -I mean use lossy compression mode which is the same as my own function.

However, the output JP2 file does not contain the actual coefficient after 2D DWT, it is necessary to insert some code into the source code to obtain the data and compare it with the data generated in Matlab. Therefore, A CSV file output code is inserted after the DWT operation. The data flow is shown in the figure 2.37. This flow is stored in tcd.c in the library file. The way to obtain the data is to output the data after DWT into a CSV file and read it into Matlab. The code is shown in appendix C. The structure of the program is quite complex, therefore only the necessary steps are discussed here. On the front of the code, there is a file opening action that

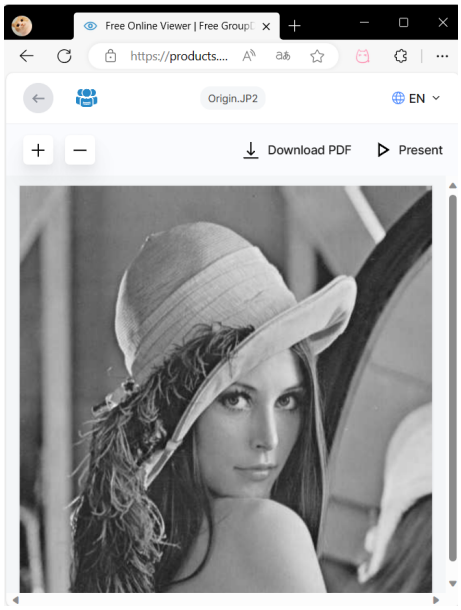


Figure 2.35: The JP2 file viewed by online viewer

creates a file for writing the output data. In the end, there is a nested for loop to write data after DWT into the file where the float_ptr is the pointer points to the modified sequence tiledp. After building and running the program again, the output data is stored in a CSV file. The CSV file is then read into Matlab and the corresponding PSNR is calculated. The PSNR function uses the previously implemented function. The result is shown in the figure 2.38. The PSNR between the reference result and the self-implemented method is over 148 dB which means the result is almost the same. The built-in PSNR function also presents over 100 dB in PSNR which fully validates the implementation of the DWT algorithm. The DWT on a tiling image is the same as the DWT applied to a whole image but with a smaller size.

2.5 Quantization

This section will discuss the quantization process, which is one of the sources of information loss in JPEG2000's lossy compression. JPEG2000 employs a uniform scalar (fixed-size) dead-zone quantization method as mentioned in [26]. It quantizes different sub-bands at various decomposition levels separately. The quantization formula is as follows:

$$q_b(u,v) = \text{sign}(a_b(u,v)) \cdot \left\lfloor \frac{|a_b(u,v)|}{\Delta_b} \right\rfloor$$

```
D:\Imperial_study\spring_term\Individual_Project_jpeg2000\OPENjpeg\openjpeg-
master\out\build\x64-Debug\bin>opj_compress.exe -ImgDir D:\Imperial_study\sp
ring_term\Individual_Project_jpeg2000\OPENjpeg -OutFor JP2 -n 2 -mct 0 -I
Folder opened successfully

File Number 0 "data.csv"
skipping file...

File Number 1 "DATA_comparision.csv"
skipping file...

File Number 2 "data_Original.csv"
skipping file...

File Number 3 "data_PSNR.csv"
skipping file...

File Number 4 "openjpeg-master"
skipping file...

File Number 5 "openjpeg-master.zip"
skipping file...

File Number 6 "opj_脚本.txt"
skipping file...

File Number 7 "Origin.bmp"
[INFO] tile number 1 / 1
[INFO] Generated outfile D:\Imperial_study\spring_term\Individual_Project_jp
eg2000\OPENjpeg/Origin.JP2

File Number 8 "Origin.JP2"
skipping file...
encode time: 566 ms
```

Figure 2.36: The command line for validating the self implemented function

Here, $a_b(u, v)$ represents the coefficients after performing the 2D DWT transformation, Δ_b represents the quantization step size, $\text{sign}(a_b(u, v))$ represents the sign of the transformation coefficient, and $q_b(u, v)$ represents the final quantized coefficient.

The quantization step size Δ_b is determined by equations as follow:

$$\Delta_b = 2^{R_b - \epsilon_b} \left(1 + \frac{\mu_b}{2^1 1}\right)$$

where R_b is the dynamic range, ϵ_b and μ_b are the exponent and mantissa stored in the QCD QCC maker segment. The nominal dynamic range R_b is defined in below equation:

$$R_b = R_I + log_2(gain_b)$$

where R_I is the number of bits used to represent the original tile-component samples which can be extracted from the SIZ marker, $log_2^{gain_b}$ is the base 2 exponent of the sub-band gain (gainb) of the current sub-band b, which varies with the type of sub-band b(levLL, levLH or levHL, levHH) and the corresponding table is shown below in table 2.2. This means that for each different subband, the corresponding quantization step size is different.

Normally, the Δ_b can be arbitrary and highly dependent on the application. To simplify the implementation, The ϵ_b and μ_b is fixed to a constant which is easy for calculation. The quantized pixels in same subband has same bit length which is necessary for later EBCOT encoding.

```
    /* FIXME _ProfStart(PGROUP_DC_SHIFT); */
    /*---------------TILE-------------------*/
    if (! opj_tcd_dc_level_shift_encode(p_tcd)) {
        return OPJ_FALSE;
    }
    /* FIXME _ProfStop(PGROUP_DC_SHIFT); */

    /* FIXME _ProfStart(PGROUP_MCT); */
    if (! opj_tcd_mct_encode(p_tcd)) {
        return OPJ_FALSE;
    }
    /* FIXME _ProfStop(PGROUP_MCT); */

    /* FIXME _ProfStart(PGROUP_DWT); */
    if (! opj_tcd_dwt_encode(p_tcd)) {
        return OPJ_FALSE;
    }
    /* FIXME  _ProfStop(PGROUP_DWT); */

    /* FIXME  _ProfStart(PGROUP_T1); */
    if (! opj_tcd_t1_encode(p_tcd)) {
        return OPJ_FALSE;
    }
    /* FIXME _ProfStop(PGROUP_T1); */

    /* FIXME _ProfStart(PGROUP_RATE); */
    if (! opj_tcd_rate_allocate_encode(p_tcd, p_dest, p_max_length,
                                       p_cstr_info, p_manager)) {
        return OPJ_FALSE;
    }
    /* FIXME _ProfStop(PGROUP_RATE); */
}
/*---------------TIER2------------------*/
/* INDEX */
if (p_cstr_info) {
    p_cstr_info->index_write = 1;
}
/* FIXME _ProfStart(PGROUP_T2); */

if (! opj_tcd_t2_encode(p_tcd, p_dest, p_data_written, p_max_length,
                        p_cstr_info, p_marker_info, p_manager)) {
    return OPJ_FALSE;
}
/* FIXME _ProfStop(PGROUP_T2); */
```

Figure 2.37: The JPEG2000 encoding data flow

2.6 EBCOT

This chapter will introduce the specific implementation process and methods of the EBCOT algorithm, which stands for Embedded Block Coding with Optimal Truncation. The EBCOT process is divided into two stages: Tier 1 and Tier 2. In Tier 1, the quantized image matrix is first divided into equally sized computational units known as code blocks. Then, each code block undergoes bitplane encoding, and the resulting output bits and contextual information are fed to the MQ arithmetic encoder for arithmetic encoding. A series of encoding operations ultimately yield the encoded data stream. Due to the limited time and personal expertise, this part of hardware design is not implemented. Several papers of hardware design for this module is referred.

```
pic = imread('Origin','bmp');%     Image read
data= csvread('data_Original.csv',0,0,[0,0,511,511]);% the reference program result
num = 2;
[x3]=ZZYnormalize(pic);%         Whole picture normalization, level shifting
y2 = DWTwhole(x3);
PSNR_val1 = PSNR(y2,data);
PSNR_val2 = psnr(y2,data);
```

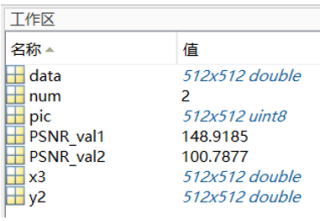


Figure 2.38: The result comparison between own function and open source program

Table 2.2: Table for the subband gain and its base 2 exponent

Sub-band b	$gain_b$	$log_2(gain_b)$
levLL	1	0
levLH	2	1
levHL	2	1
levHH	4	2

2.6.1 Tier 1 coding

Bitplane coding

The process is shown in the figure 2.39. The input code block is firstly transformed from IEEE 754 format to the binary format with a sign bit. The same bit from each coefficient in a code block forms a bit plane. If the input data consists of 20 bits. Then there are 20 bitplanes starting from MSB to LSB as shown on the left. The scan pattern of the bitplane coding starts from MSB to LSB and in each bitplane, The scan pattern is shown on the top right of the figure. Four vertical bit is called a strip and after scanning a row of strips, it will go back to the front and start from the second row of strips till the end of this bitplane. If the last row does not satisfy a strip then the rest of the rows will be regarded as a strip and the encoding will continue.

Every bitplane will be scanned three times for allocating the three passes to each bit individually and each bit can only be allocated with one pass. The neighbors are the surrounding 8 bits of scanned bit X as shown in the fig2.40. The bit named H_n, V_n and D_n are the horizontal, vertical and diagonal neighbors of middle bit X where n represents the index of the neighbors. Each coefficient has a significance identifier. If the coefficient in a bit plane first turns its bit to 1 during the process of biplane scan, the significance identifier of this coefficient will become 1 from this point to the last bit plane. In the scan of each bit, it will be allocated with a pass from three

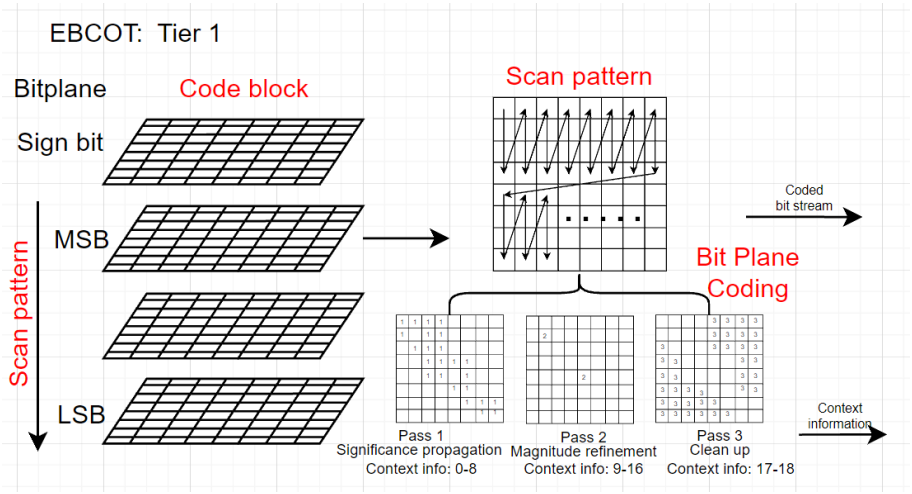


Figure 2.39: The subband allocation after 2D DWT decomposition twice

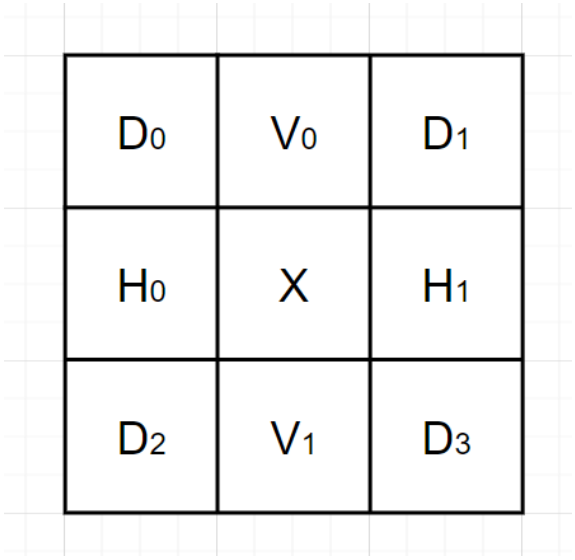


Figure 2.40: The neighbour information of a bit X

passes. The first pass is the Significance propagation pass. This pass is allocated to the bit whose significance identifier is not 1 but its neighbors have no zero bit. Significance coding is applied. If the bit is turning into 1 from 0 after it, then additional sign coding is applied. Second, Magnitude Refinement Pass is allocated to those bits whose significance identifier is already 1. Magnitude coding is applied. Thirdly, if there are still bits in this bitplane that are left to be encoded where all of them are not significant with no neighbor bits that are significant, then all of them are allocated with Clean-up Pass, and cleanup coding is applied. To sum up, there are three passes and 4 coding passes and each bit will be allocated with context information through the bitplane coding.

During the coding of each bit in the bitplane, each bit will be allocated with context information according to its subband, sign, and conditions of the neighbor's significant identifier. Figure 2.41

Table D.1 – Contexts for the significance propagation and cleanup coding passes

LL and LH sub-bands (vertical high-pass)			HL sub-band (horizontal high-pass)			HH sub-band (diagonally high-pass)		Context label[a]
$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum (H_i + V_i)$	$\sum D_i$	
2	x[b]	x	x	2	x	x	≥3	8
1	≥1	x	≥1	1	x	≥1	2	7
1	0	≥1	0	1	≥1	0	2	6
1	0	0	0	1	0	≥2	1	5
0	2	x	2	0	x	1	1	4
0	1	x	1	0	x	0	1	3
0	0	≥2	0	0	≥2	≥2	0	2
0	0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0

a) Note that the context labels are indexed only for identification convenience in this Recommendation | International Standard. The actual identifiers used is a matter of implementation.

b) "x" indicates a "don't care" state.

Figure 2.41: The context information table for Significance Propagation and Cleanup coding pass

shows the context information for significance propagation and cleanup coding pass. The context information for these two coding passes depends on the conditions of the neighbor's significance states. If the coded bit is at the edge of the code block, then the nearest neighbor lying outside the current coefficient's code block is regarded as insignificant for the independence of each code block.

For sign coding passes, it will calculate the contributions of the vertical (and the horizontal) neighbors to the sign context then through another look-up table to find the context information. The contribution table and the context table are shown in figure 2.42 and 2.43. After gathering the information from the neighbors, the contributions are figured out and the corresponding context label is determined after that.

Magnitude refinement pass will use the information of whether it's the first time going through the refinement and the conditions for the neighbor's information to determine the context because the same coefficient can go through refinement pass multiple times in different bitplanes, therefore it's necessary to store the history of the refinement. the context table is presented in fig 2.44.

Finally, the Cleanup Pass is used to clean up all bit which is not significant, therefore a Run-length context mode is applied. The brief principle is that If four bits in a column are not significant, then a single binary bit is used to represent this column, However, if at least one of the four bits is significant, then the position of the first significant bit is labeled and the Run-length mode is stopped and the rest bits of the column will be decoded in previous passes rather than clean up pass. The details are as follows: If the four contiguous coefficients in a column remaining to be

2

Table D.2 – Contributions of the vertical (and the horizontal) neighbors to the sign context

V0 (or H0)	V1 (or H1)	V (or H) contribution
significant, positive	significant, positive	1
significant, negative	significant, positive	0
insignificant	significant, positive	1
significant, positive	significant, negative	0
significant, negative	significant, negative	–1
insignificant	significant, negative	–1
significant, positive	insignificant	1
significant, negative	insignificant	–1
insignificant	insignificant	0

Figure 2.42: The horizontal and vertical contribution information for Sign coding pass

decoded and each currently has the 0 context then the unique run-length context is given to the arithmetic decoder with the bit stream, if the returning symbol is 0, then all four coefficients will remain insignificant and set to be 0. If the returning symbol is 1, which means at least one of the four contiguous coefficients in the column is significant, the two-bit symbol will record the position of the significant bit stop the run-length coding, and transfer to previous passes for coding. The logical table for the cleanup pass is shown in figure2.45. If there are fewer than four rows remaining in a code block then run-length coding will not be applied. Once again, the significance state of any coefficient is changed immediately after decoding the first 1 magnitude bit.

MQ arithmetic encoder

Arithmetic encoding is based on the idea of representing a sequence of symbols as a single floating-point number in the interval [0, 1). It focuses on encoding an entire sequence, rather than individual symbols, therefore there is no one-to-one correspondence between source symbols and code words. Here is how it works in detail:

Before the encoding begins, the interval representing the sequence of symbols is set to zero to one. For each symbol in the sequence, the [0, 1) interval is subdivided based on the probability of that symbol occurring. Every time a new symbol is coming, the interval then shrinks to the sub-interval corresponding to that symbol. As more symbols are processed, the length of the interval representing them gets smaller. Consequently, more bits are needed to represent this interval accurately.

Table D.3 – Sign contexts from the vertical and horizontal contributions

Horizontal contribution	Vertical contribution	Context label	XORbit
1	1	13	0
1	0	12	0
1	–1	11	0
0	1	10	0
0	0	9	0
0	–1	10	1
–1	1	11	1
–1	0	12	1
–1	–1	13	1

Figure 2.43: The context information table for Sign coding pass

Table D.4 – Contexts for the magnitude refinement coding passes

$\sum H_i + \sum V_i + \sum D_i$	First refinement for this coefficient	Context label
x[a]	false	16
≥1	true	15
0	true	14

a) "x" indicates a "don't care" state.

Figure 2.44: The context information table for magnitude refinement pass

For MQ arithmetic encoder, a kind of adaptive binary arithmetic encoder, is a specific form of arithmetic encoder where the source sequence consists only of binary symbols which are 0 and 1. The output is a bitstream that represents a binary fraction, and its precision improves as more bits are added during the encoding process. This kind of encoder will take the changing possibility of the symbols into account which can predict the probabilities of further subdividing the interval. Also, the probabilities will be adaptively updated to the input sequence as the new symbols are added. One of the key advantages of arithmetic encoding is its efficiency. It allows the representation of the source entropy in fractional bits, breaking the limitation of Huffman encoding, which can only approximate source entropy in whole bits.

The basic flow of the arithmetic coder is shown in the fig 2.46. In the beginning, the INITECN module will initialize the encoder and feed the bit stream (D) and context information (CX) into

Table D.5 – Run-length decoder for cleanup passes

Four contiguous coefficients in a column remaining to be decoded and each currently has the 0 context	Symbols with run-length context	Four contiguous bits to be decoded are zero	Symbols decoded with UNIFORM[a] context	Number of coefficients to decode
true	0	true	none	none
true	1	false	MSB LSB	
		skip to first coefficient sign	00	3
		skip to second coefficient sign	01	2
		skip to third coefficient sign	10	1
		skip to fourth coefficient sign	11	0
false	none	x	none	rest of column
[a] See Annex C.				

Figure 2.45: The logic table for Cleanup pass

the encoder. The ENCODE module does the probability prediction process and the FLUSH will set the output register for outputting or stop the encoding process and generate the stop flag.

The algorithm of encoding is as follow: Two register A and C are defined in figure 2.47, where a and x are the fractional bit of register A and C. s bit are the spacer bits and b bits represents where the completed bytes are moved out. c bit is a carry bit. In the process of encoding, fixed-precision integer arithmetic is used for addition and subtraction, thereby replacing decimals with integers. For example, the hexadecimal integer 0x8000 represents the decimal fraction 0.750. The formula for converting between integers and decimals is as follows.

$$Q_{e(DEC)} = \frac{Q_{e(DEC)}}{\frac{4}{3} \times 0X8000} \tag{2.7}$$

The value of interval register A should be between 0.75 and 1.5. Every time A is smaller than 0.75, register A and C should left shift 1 bit. To prevent the overflow of register C, The higher 8-bit should be moved out periodically and written into a data buffer and the whole operation is done by the RENORME module. Since the register has a value between 0.75 to 1.5. The interval of MPS (Most probable symbol) and LPS(less probable symbol) can be approximated to $A - Q_e$ and Q_e, where A is the current length of the interval and Q_e is the probability of LPS. When MPS is coded, Q_e is placed in register C and the current length of the interval (A) is reduced to $A - Q_e$. If LPS is coded, register C will remain the same, and register A will be reduced to Q_e. If the length of LPS (Q_e) is larger than the length of MPS, they should exchange with each other to ensure MPS always has the longer length.

As shown in figure 2.48The flow of the ENCODE module will compare whether the decision

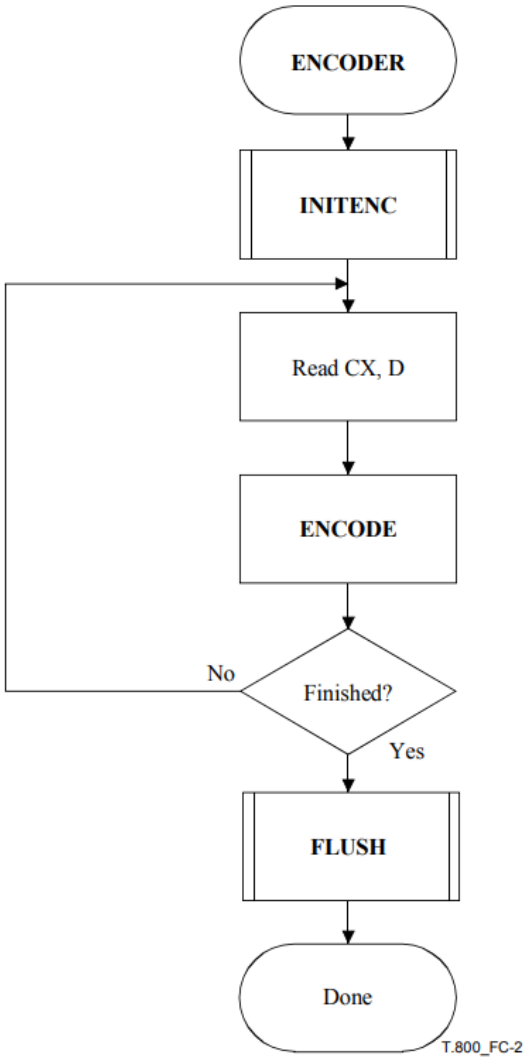


Figure 2.46: The data flow of MQ coder

bit(D) is 0 or 1. Then it will compare the MPS(CX). If the D is equal to MPS(CX), the MPS is set to D and does CODEMPS operation, otherwise, LPS is set to D and CODELPS is applied. The flow chart of CODELPS and CODEMPS is shown in fig 2.49. The process of CODELPS will firstly calculate the interval length of MPS and compare it with

$$Q_e(I(CX))$$

to ensure LPS is the shorter one. Otherwise, change the value. If SWITCH(CX) is equal to 1, then MPS(CX) will become its reverse and The new I(CX) can be found in the table in figure 2.50. CODEMPS will first calculate the MPS interval length $A - Q_e(I(CX))$ and then adjust the register C to point to the bottom of MPS. Then the same as the previous case, compare the length

	MSB			LSB
C-register	0000 cbbb	bbbb bsss	xxxx xxxx	xxxx xxxx
A-register	0000 0000	0000 0000	1aaa aaaa	aaaa aaaa

Figure 2.47: The Register for MQ encoder

of LPS and MPS and give the longer value to register A. The new I(CX) also can be found in fig 2.50.

The table in fig 2.50 shows the Finite-State Probability Estimation where NMPS and NLPS can be looked up immediately.

Both the CODEMPS and CODELPS process will apply the RENORME module as shown in fig 2.51. It's a process of renormalization. Every time it normalizes, registers A and C will left shift 1 bit and a counter CT is used to count down the number of shifts. If the counter is equal to 0, the compressed data will be moved out from register C using the BYTEOUT module.

The Byteout module will output the compressed data. This module utilizes the Bit stuffing Technique where if the compressed data happens to be OxFF, the MSB of the next byte will be placed with 0. The fig 2.52 shows the data flow of the BYTEOUT. As shown in the figure, if the current Byte B is 0xFF, the bit stuffing applies as the B will add a carry bit and be checked again whether it is 0xFF. If one of these two check is 1, the next byte will be stuffed with a zero. If both of these two check is not satisfied, no bit is stuffed into the next byte. The compressed image data buffer pointer BP points to the current byte. If the next byte is instead a bit, the counter CT will be set to 7 rather than 8 to ensure the correctness of the data length.

Finally, the Flush procedure is used to terminate the encoding operations and generate the required terminating marker and the procedure is shown in figure 2.53 and figure 2.54. The first part of FLUSH is setbit which is to set as many bits in register C as possible to 1. There is an upper bound for C which is the sum of the C register and interval register. After forcing the lower 16-bit to 1, If the C is larger than $C + A$, then the leading 1-bit is removed which places the value of C back into the boundary. The second part of FLUSH is to shift out two-byte form register C. Lastly, it will check whether the byte in buffer, B, is an 0xFF. If so, then it is discarded. Otherwise, buffer B is output to the bit stream.

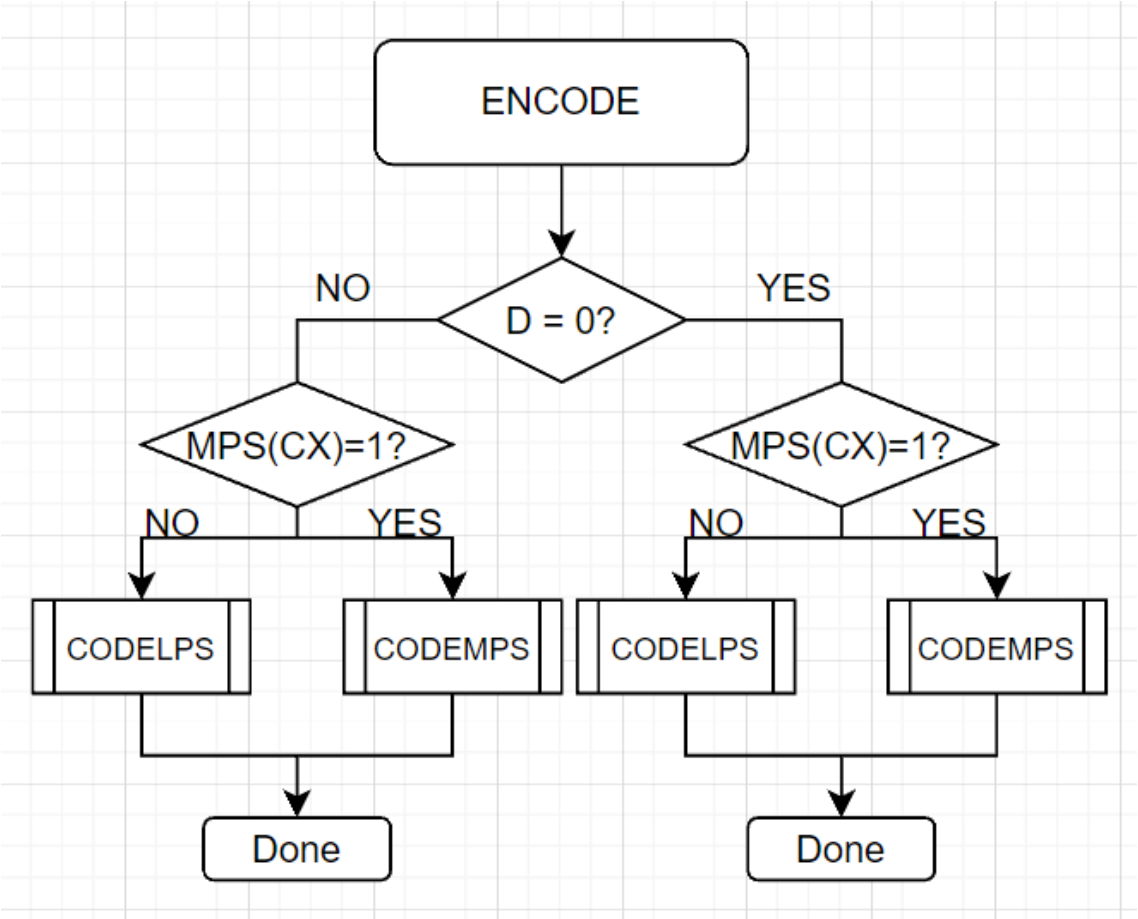
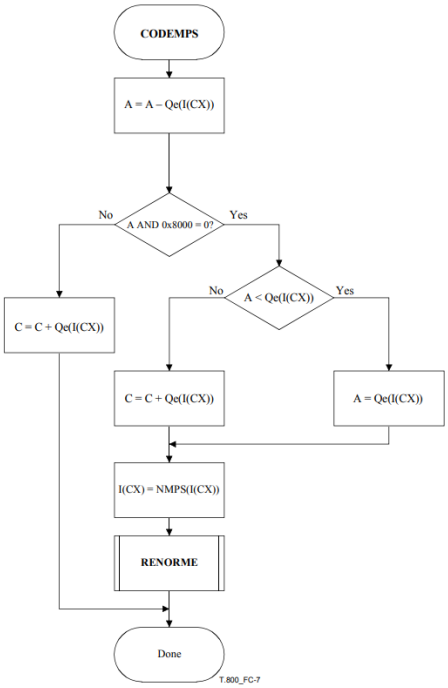


Figure 2.48: The encode workflow

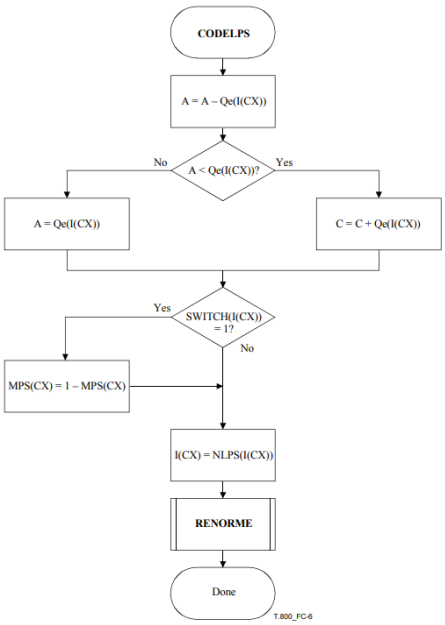
2.6.2 Tier 2 coding

Tier 2 takes the bitstreams of each code block from tier 1, which have undergone bitplane encoding and arithmetic encoding, and applies a post-compression rate-distortion optimization algorithm to calculate the optimal truncation set for each code block at a given bit rate that minimizes the overall distortion. If a set of truncation points can be found that minimizes the total distortion under a specified bit rate, it is determined to be the optimal truncation point at that bit rate. This optimization algorithm is known as post-compression rate-distortion optimization.

In the simplest way to obtain the final compressed data, where each encoded block is at a fixed resolution, this packetization method exhibits resolution scalability. It also has spatial scalability because each encoding block only affects a specific region of the image. Therefore, if there are regions of interest, the encoding blocks corresponding to these regions can be processed first. EBCOT also introduces the concept of quality layers to support multiple levels of distortion. Each quality layer is made up of contributions from various encoding blocks; of course, some encoding



(a) Flow chart of CODEMPS



(b) Flow chart of CODELPS

Figure 2.49: The process of module CODEMPS and module CODELPS

blocks may not contribute to a certain layer. The resulting layered bitstream has distortion-scalable characteristics. During decoding, the image can be progressively transmitted.

The hardware deign of EBCOT is complex and there are some investigation on the design as shown in [27].Besides the design of EBCOT, It proposed two methods in speeding up the EBCOT operations called sample skipping (SS) and group-of-column skipping (GOCS).

Index	Qe_Value			NMPS	NLPS	SWITCH
	(hexadecimal)	(binary)	(decimal)			
0	0x5601	0101 0110 0000 0001	0,503 937	1	1	1
1	0x3401	0011 0100 0000 0001	0,304 715	2	6	0
2	0x1801	0001 1000 0000 0001	0,140 650	3	9	0
3	0x0AC1	0000 1010 1100 0001	0,063 012	4	12	0
4	0x0521	0000 0101 0010 0001	0,030 053	5	29	0
5	0x0221	0000 0010 0010 0001	0,012 474	38	33	0
6	0x5601	0101 0110 0000 0001	0,503 937	7	6	1
7	0x5401	0101 0100 0000 0001	0,492 218	8	14	0
8	0x4801	0100 1000 0000 0001	0,421 904	9	14	0
9	0x3801	0011 1000 0000 0001	0,328 153	10	14	0
10	0x3001	0011 0000 0000 0001	0,281 277	11	17	0
11	0x2401	0010 0100 0000 0001	0,210 964	12	18	0
12	0x1C01	0001 1100 0000 0001	0,164 088	13	20	0
13	0x1601	0001 0110 0000 0001	0,128 931	29	21	0
14	0x5601	0101 0110 0000 0001	0,503 937	15	14	1
15	0x5401	0101 0100 0000 0001	0,492 218	16	14	0
16	0x5101	0101 0001 0000 0001	0,474 640	17	15	0
17	0x4801	0100 1000 0000 0001	0,421 904	18	16	0
18	0x3801	0011 1000 0000 0001	0,328 153	19	17	0
19	0x3401	0011 0100 0000 0001	0,304 715	20	18	0
20	0x3001	0011 0000 0000 0001	0,281 277	21	19	0
21	0x2801	0010 1000 0000 0001	0,234 401	22	19	0
22	0x2401	0010 0100 0000 0001	0,210 964	23	20	0
23	0x2201	0010 0010 0000 0001	0,199 245	24	21	0
24	0x1C01	0001 1100 0000 0001	0,164 088	25	22	0
25	0x1801	0001 1000 0000 0001	0,140 650	26	23	0
26	0x1601	0001 0110 0000 0001	0,128 931	27	24	0
27	0x1401	0001 0100 0000 0001	0,117 212	28	25	0
28	0x1201	0001 0010 0000 0001	0,105 493	29	26	0
29	0x1101	0001 0001 0000 0001	0,099 634	30	27	0
30	0x0AC1	0000 1010 1100 0001	0,063 012	31	28	0
31	0x09C1	0000 1001 1100 0001	0,057 153	32	29	0
32	0x08A1	0000 1000 1010 0001	0,050 561	33	30	0
33	0x0521	0000 0101 0010 0001	0,030 053	34	31	0
34	0x0441	0000 0100 0100 0001	0,024 926	35	32	0
35	0x02A1	0000 0010 1010 0001	0,015 404	36	33	0
36	0x0221	0000 0010 0010 0001	0,012 474	37	34	0
37	0x0141	0000 0001 0100 0001	0,007 347	38	35	0
38	0x0111	0000 0001 0001 0001	0,006 249	39	36	0
39	0x0085	0000 0000 1000 0101	0,003 044	40	37	0
40	0x0049	0000 0000 0100 1001	0,001 671	41	38	0
41	0x0025	0000 0000 0010 0101	0,000 847	42	39	0
42	0x0015	0000 0000 0001 0101	0,000 481	43	40	0
43	0x0009	0000 0000 0000 1001	0,000 206	44	41	0
44	0x0005	0000 0000 0000 0101	0,000 114	45	42	0
45	0x0001	0000 0000 0000 0001	0,000 023	45	43	0
46	0x5601	0101 0110 0000 0001	0,503 937	46	46	0

Figure 2.50: The probability estimation look up table

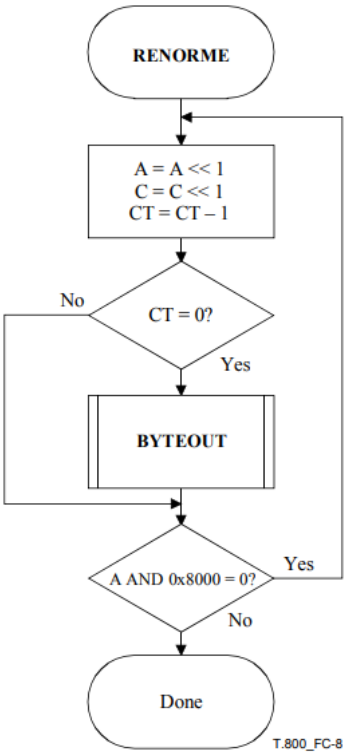


Figure 2.51: The renormalization procedure

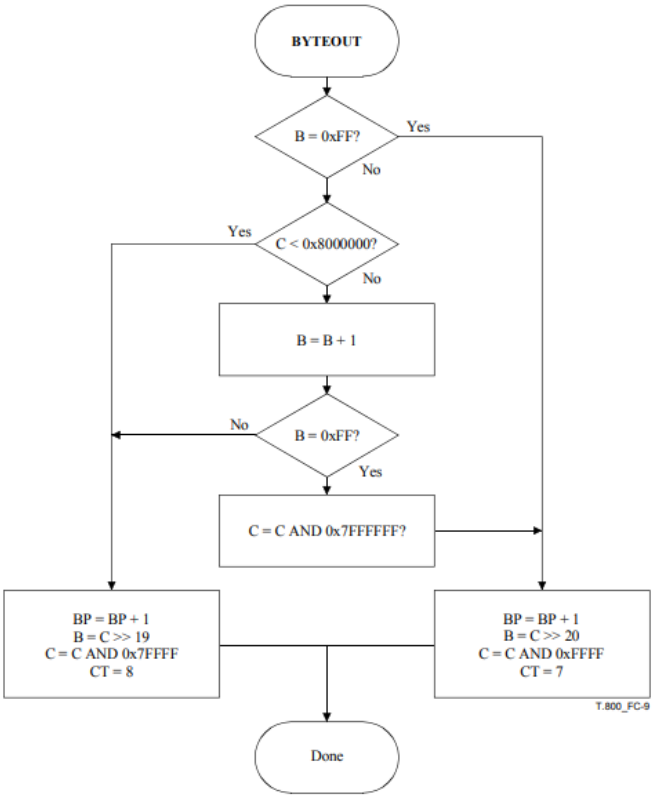


Figure 2.52: The BYTEOUT data flow

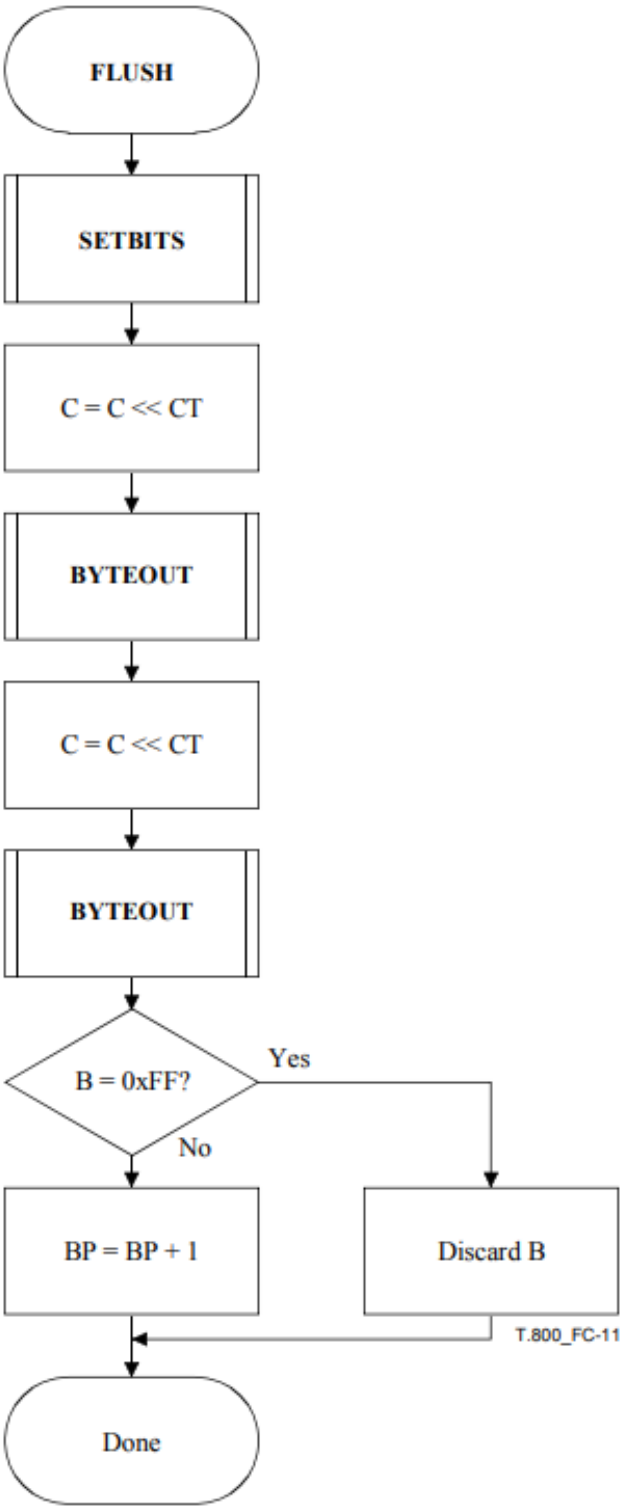


Figure 2.53: The FLUSH operation

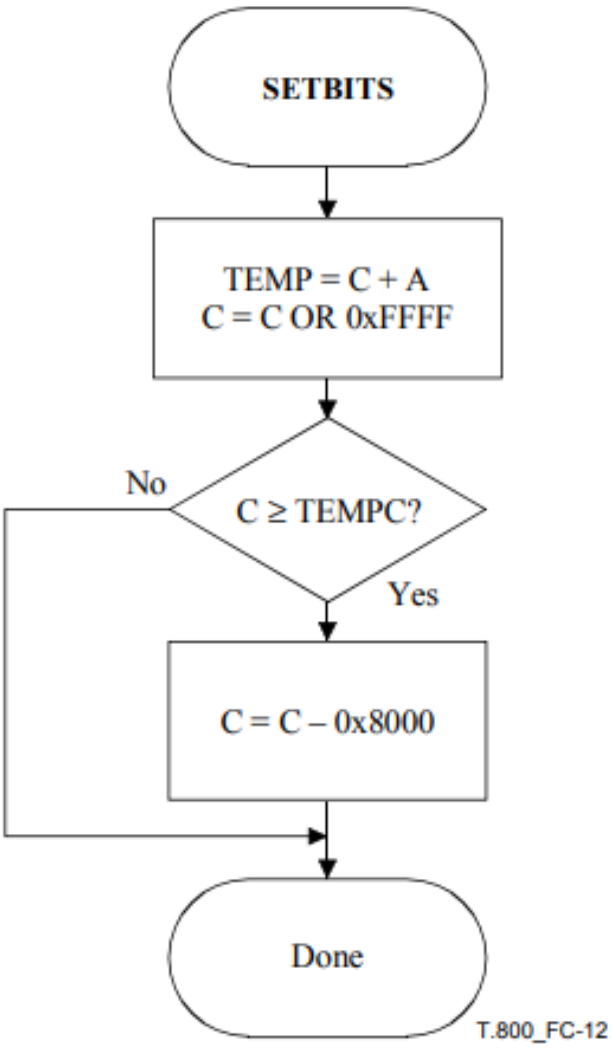


Figure 2.54: The setbit operation

3 Hardware implementation

.

Contents

THIS chapter discusses the hardware implementation of the JPEG2000 image compression algorithm and its simulation results. Due to time constraints and limited personal capabilities, this paper only replicates the preprocessing and DWT (Discrete Wavelet Transform) algorithm modules. The subsequent parts only provide relevant hardware design ideas.

3.1 Pre processing

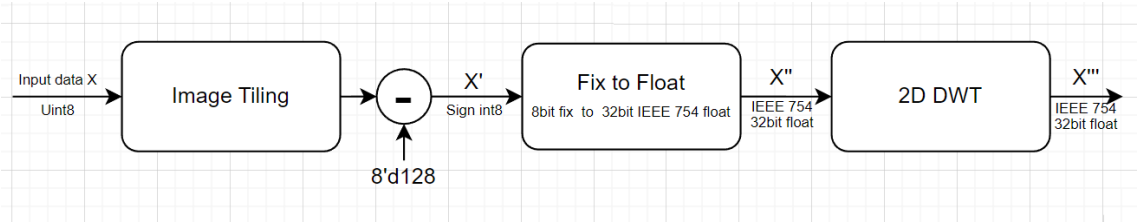


Figure 3.1: The data flow of hardware implementation

This hardware accepts 8 bits unsigned int data one by one and the input go through tiling, DC level shifting and 2D DWT. One thing to notice is that the DSP IP used in DWT algorithm

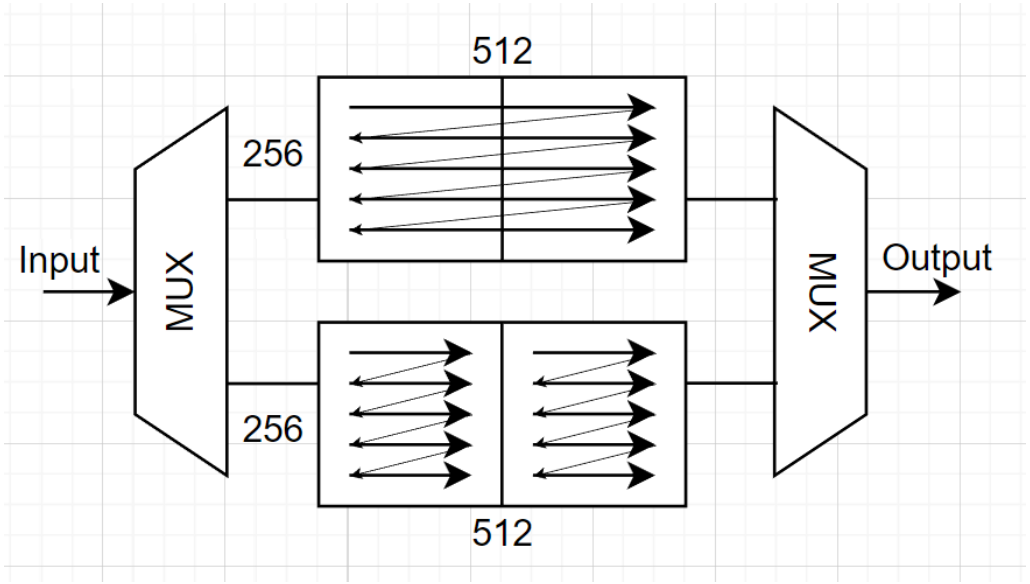


Figure 3.2: The tiling method using ppbuffer

is 32 bit long. Therefore, the input should be also change into 32 IEEE format at the end of preprocessing module for the purpose of maintaining a consistent format.

The flow diagram is shown in figure 3.1. The input Uint8 data X is firstly tiled into specific number of tiles and then subtract 128 in binary form to get a Signed data X'. The signed data is then translated into IEEE 754 single size (32bit) data X" using IP provided. 2D DWT is then applied on the result and the final result is X"'.

For DC level shifting part, the module is quite simple and the code is shown below:

```
module SUB(
    input wire clk,
    input wire nrst,
    input wire [7:0] data_in,
    input wire data_in_valid,
    output reg [7:0] data_out,
    output reg data_out_valid
    );

    always @(posedge clk or negedge nrst )begin
        if(~nrst) begin
            data_out <= 8'b0;
            data_out_valid <=1'b0;
        end
        else if(data_in_valid) begin
            data_out <= data_in - 8'd128;
            data_out_valid <= 1'b1;
        end
    end

endmodule
```

Image tiling is the part that split the original data into several small blocks. The idea is shown in figure 3.2The verilog code is shown in below:

```
module Tiling(
    input clk,
    input nrst,
    input [31:0] din ,
    input din_valid,
    output [31:0] dout,
    output dout_valid
    );
    reg dout_valid;
    wire [31:0] dout;

// din_counter
reg [16:0] din_cnt; // 0-131071
```

```
14 wire [16:0] dout_cnt; //
15 reg [8:0]din_cnt_col;  //0-511
16 reg [7:0]din_cnt_row;  // 0-255
17 assign dout_cnt = {din_cnt[15:8],din_cnt[16],din_cnt[7:0]};
18
19 always @(posedge clk or negedge nrst) begin
20     if (~nrst) begin
21         din_cnt_col <=0;
22         din_cnt_row <=0;
23         din_cnt <=0;
24     end if (din_valid) begin
25         din_cnt_col <= din_cnt_col +1;
26         if (din_cnt_col == 9'd511)
27             din_cnt_row <= din_cnt_row+1;
28         din_cnt <= din_cnt+1;
29     end
30 end
31 // RAM
32 reg wea;//wirte enable a
33 reg web;// write enable b
34 wire [31:0] douta;//output from ram0
35 wire [31:0] doutb;//output from ram1
36 wire rsta_busy0;
37 wire rsta_busy1;
38
39 //write enable Control.  write ram0 read ram1 first, then change
40 always @ (posedge clk, negedge nrst) begin
41     if(~nrst) begin
42         wea <=1;
43         web <=0;
44     end else begin
45         if(din_cnt_col==9'd511 & din_cnt_row == 9'd255 & din_valid) begin
46             wea <=~wea;
47             web <=~web;
48         end
49     end
50 end
```

```
51
52 //buffer raedy signal
53 reg ready;
54 reg ready_delay[1:0];
55 always @ (posedge clk, negedge nrst) begin
56     if(~nrst) begin
57         ready <= 0;
58         ready_delay[0] <=0;
59         ready_delay[1] <=0;
60     end
61     else if(din_valid & din_cnt == 17'd131071)begin
62         ready <= ~ready;
63         ready_delay[0] <= ready;
64         ready_delay[1] <= ready_delay[0];
65     end
66 end
67
68 // dout valid control
69 always @ (posedge clk, negedge nrst) begin
70     if(~nrst)
71         dout_valid <= 0;
72     else
73         if(din_valid & ready==1)
74             dout_valid <= 1'd1;
75 end
76 //dout control
77 assign dout = (ready_delay[1])?douta:doutb;
78 //ram control
79 blk_mem_gen_1 inst1 (
80   .clka(clk),             // input wire clka
81   .rsta(~nrst),             // input wire rsta
82   .wea(wea),               // input wire [0 : 0] wea
83   .addra(wea?din_cnt:dout_cnt),          // input wire [16 : 0] addra
84   .dina(din),             // input wire [31 : 0] dina
85   .douta(douta),           // output wire [31 : 0] douta
86   .rsta_busy(rsta_busy0)  // output wire rsta_busy
87 );
```

```
88 blk_mem_gen_1 inst2 (
89   .clka(clk),             // input wire clka
90   .rsta(~nrst),            // input wire rsta
91   .wea(web),              // input wire [0 : 0] wea
92   .addra(web?din_cnt:dout_cnt),          // input wire [16 : 0] addra
93   .dina(din),             // input wire [31 : 0] dina
94   .douta(doutb),           // output wire [31 : 0] douta
95   .rsta_busy(rsta_busy1)  // output wire rsta_busy
96 );
97
98 endmodule
```

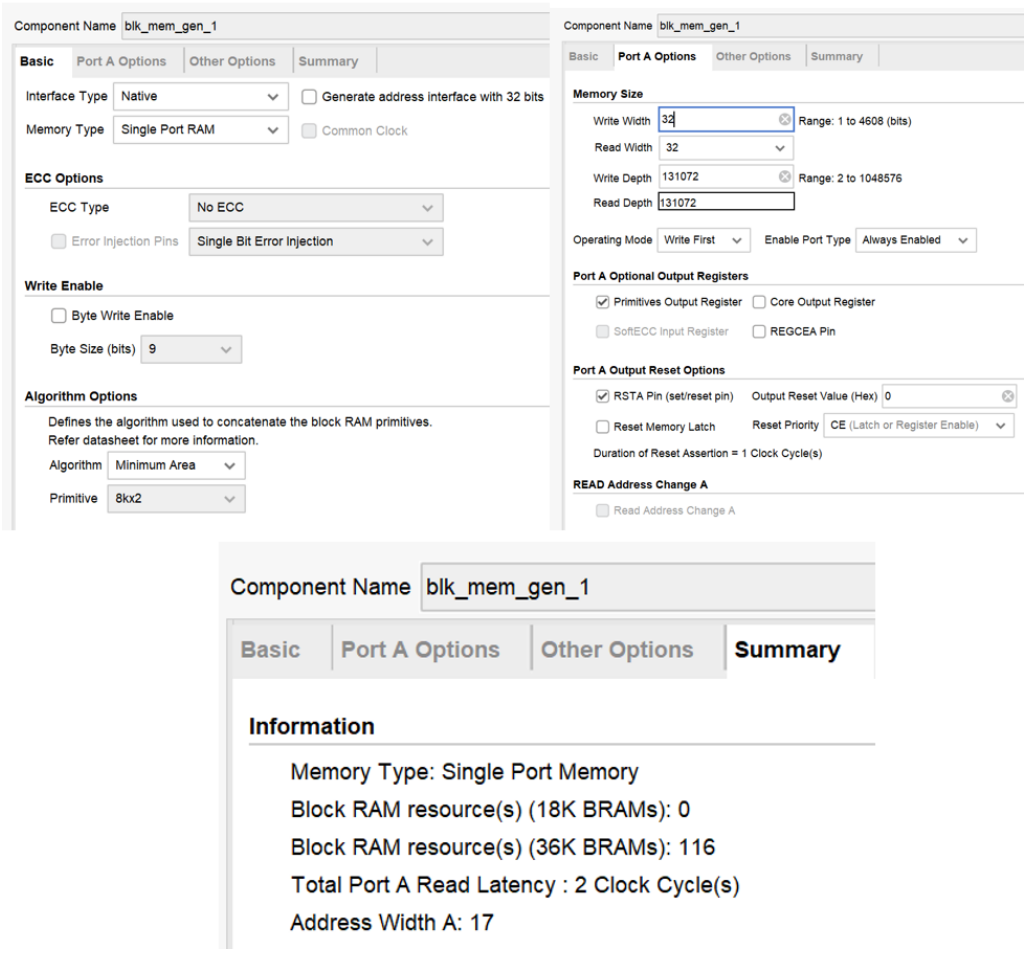


Figure 3.3: The configuration of PPbuffer

In this module, ping pang buffer is used to tile the input image into 4 small blocks. Ping Pang buffer is a modified double buffer, it utilises two buffer and two multiplexers. As one of the buffer is writing new data in, the other buffer is reading data out. By changing the write or read mode,

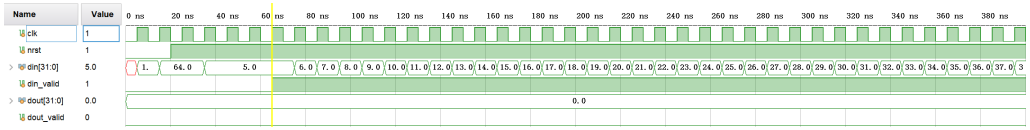


Figure 3.4: The start of the simulation of tiling module

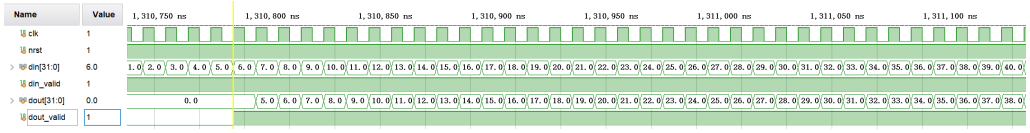


Figure 3.5: The transition of buffer of tiling module

a smooth non stoping data flow is achieved. Without ppbuffer, the new image data will have to stop to wait for cycles that whole image data needed to be transmitted.

The number of tiles are alternative and depend on the application, for demonstration, 4 tiles are used. The ping pong buffer uses block memory generator provided in IP as shown in figure 3.3. The ppbuffer is controlled by several counters, din_cnt for writing into buffer, dout_counter for reading out from buffer. Ready signal is set high periodically to indicate the buffer is full and change the buffer to write and read. It is delayed two cycles as the latency for buffer is two cycles mentioned in 3.3. There is trick which simplify the complexity of control. To form the reading pattern shown in the bottom of the figure 3.2, the MSB of the counter is inserted between ninth and eighth bit which gives the demand reading pattern. A simple exchange of the wire gives the logic needed which optimize the control process. In the first phase, buffer a is set to write mode and buffer b is in read mode where write enable a (wea) is high and write enable b (web) is low. Therefore the output is 0 which is the content of buffer b at the beginning. After the buffer is full at phase 2, the write enable will exchange where wea and web is set to 0 and 1. This time, the output is from buffer a where the content is already written in phase 1.

The testbench is shown in appendix D. The signal nrst is set high at 20ns and the input data is an auto increment counter starts from 5 for simplicity. It go through a fix to float module for the transition and the result is fed into the module. The simulation is shown in figure 3.4, The input starts from 5 and the increase by 1 every time. At the transition of two buffer, the output will also start output the result and the simulation is shown in figure 3.5. As the din_cnt reaches the end of he buffer, the two buffer's mode is changed and due to the latency of the IP, the output will start giving the content two cycles after the mode transition as illustrated in the simulation.

To extend the tiling module, the way to split ping pang buffer have to change. If the image is tiled into 4*4 tiles, The logical control of the address counter, which control the output address,

will have to change. In this example, the counter MSB is inserted between ninth and eighth bit, if the requirement is changed to 16 tiles, the MSB will have to be asserted between eighth and seventh bit. similarly to other number of tiles.

Utilization **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	51	1182240	0.01
FF	40	2364480	0.01
IO	68	832	8.17
BUFG	1	1800	0.06

Figure 3.6: The post synthesis resource used for tiling module

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1813	1182240	0.15
LUTRAM	212	591840	0.04
FF	1144	2364480	0.05
BRAM	232	2160	10.74
IO	68	832	8.17
BUFG	1	1800	0.06

Figure 3.7: The post implementation resource used for tiling module

The synthesis and implementation is shown in figure 3.6 and 3.7. Through the figure, it shows that 232 bram is used. The buffer size is calculate as $32 \times 512 \times 256 \times 2 = 8,388,608$ bits. $\frac{8,388,608}{18,000}$ = 466.0337 brams should be used. The reason why there is only half of the number is that it uses another type of bram which consists of 36k bits per bram. Therefore, the number of bram used is half of calculated. The timing report for the tiling module is shown in figure 3.8 and all requirement is met.

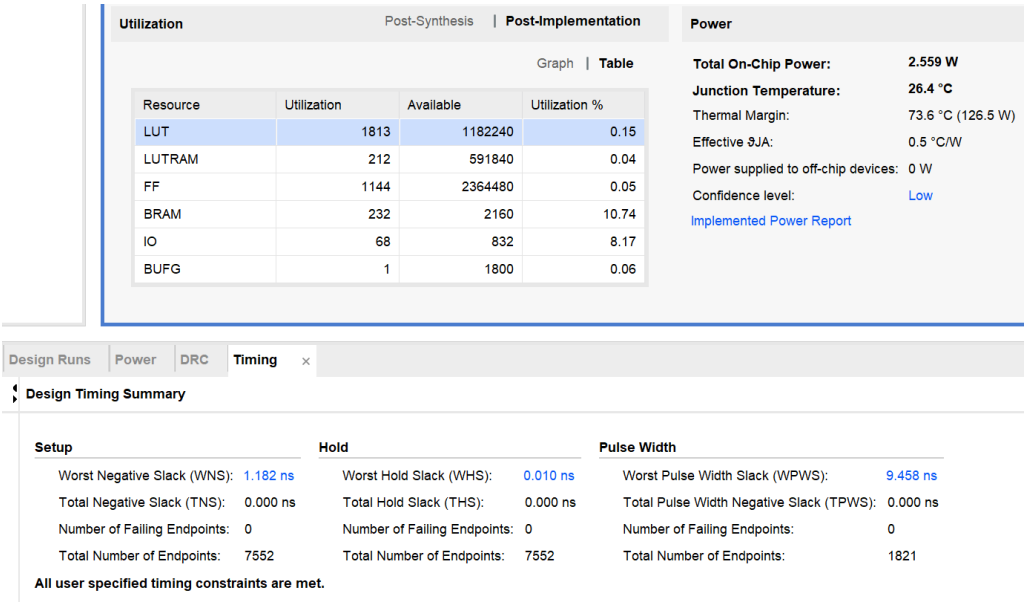


Figure 3.8: The timing report for tiling module

3.2 2D DWT

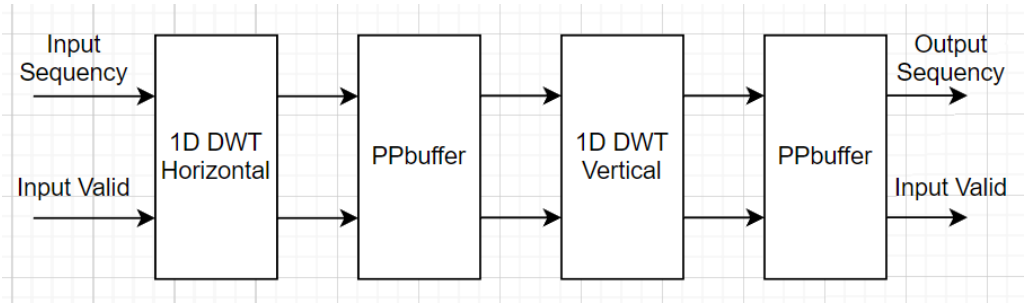


Figure 3.9: The general hardware data flow of 2D DWT

2D DWT is the following processing part after preprocessing, Due to the processing requirement on both directions, it is splited into two 1D DWT operation. The whole process is shown in figure 3.9. The input is firstly fed into the horizontal 1D DWT module which performs the operation and result fills the PPbuffer to ensure no stalls happen. After then, the buffer the result is going through vertical 1D DWT process and PPbuffer similarly. The details of each block are described below.

3.2.1 1D DWT

For the 1D DWT module it includes horizontal and vertical operations and they are further decomposed into first half and second half. The first half do prediction and update operations once.

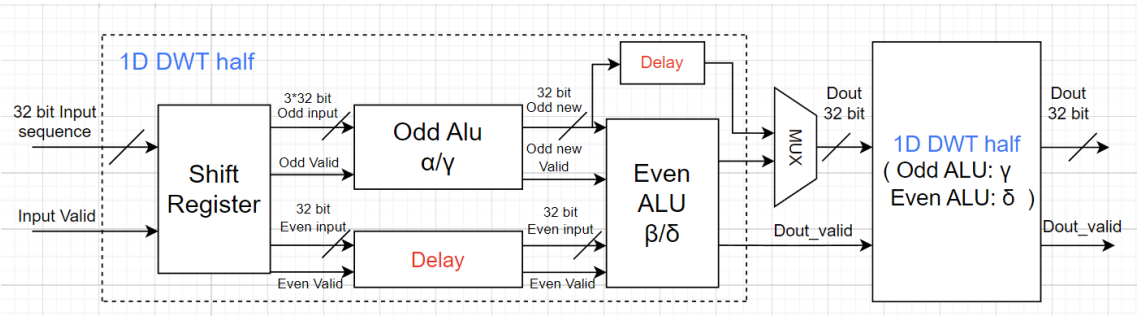


Figure 3.10: The detail of 1D DWT hardware implementation

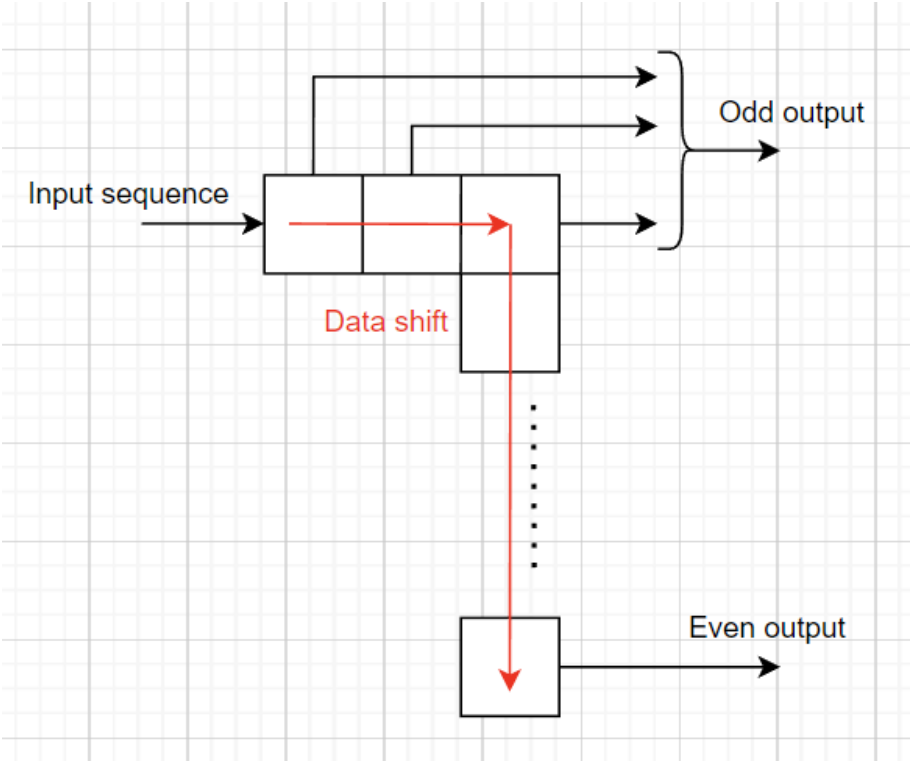


Figure 3.11: The data flow of shift register

Second half do the prediction and update with other parameters and do the scaling as well. The detailed diagram is shown in figure 3.10. The input sequence is firstly stored in shift register, and the odd elements with it adjacent two even elements are sent to odd ALU. The even elements are delayed to match with predicted odd elements odd new. Then the delayed even elements and new odd elements are send to even ALU which update the old even elements. The odd new and even new are then separately scaled with factor K and 1/K. The result is finally aligned in order and output in sequence. This is thw whole process of 1D DWT. The verilog code for necessary module is in appendix E and F. The module used in 1D DWT includes shift register, odd and even ALU, delay module. Originally, a fully ram based design is proposed. However, it takes too much resource and the complexity is not reduced much. Therefore, a new shift register based

design is come up. To simplify the verification, DWT uses whole images to do the calculation and calculation for tiled image is the same. If extra functions such as dynamically choosing the tile size is needed, the solution is implementing the the different DWT module for each tile size and using multiplexers to select which one to output which may cost extra resources. The following chapter will discuss the DWT operations in detail.

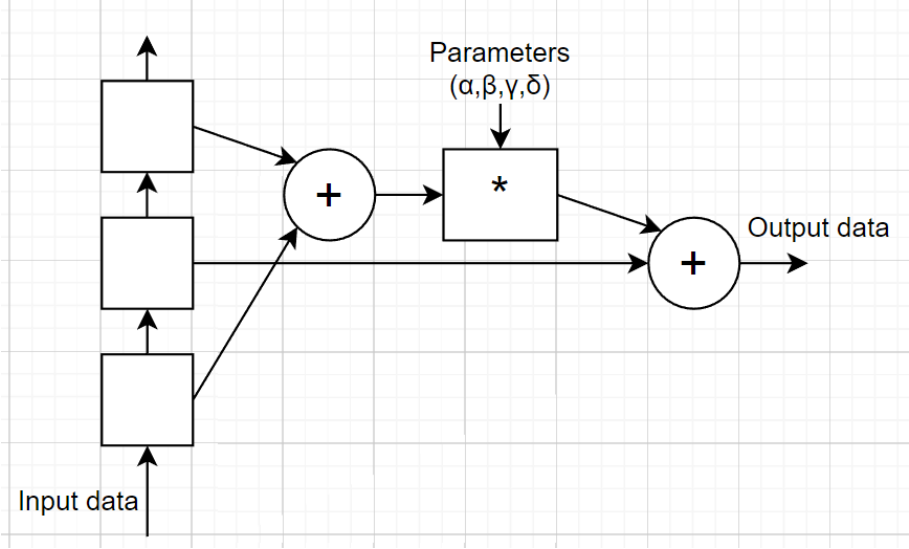


Figure 3.12: The data flow of ALU

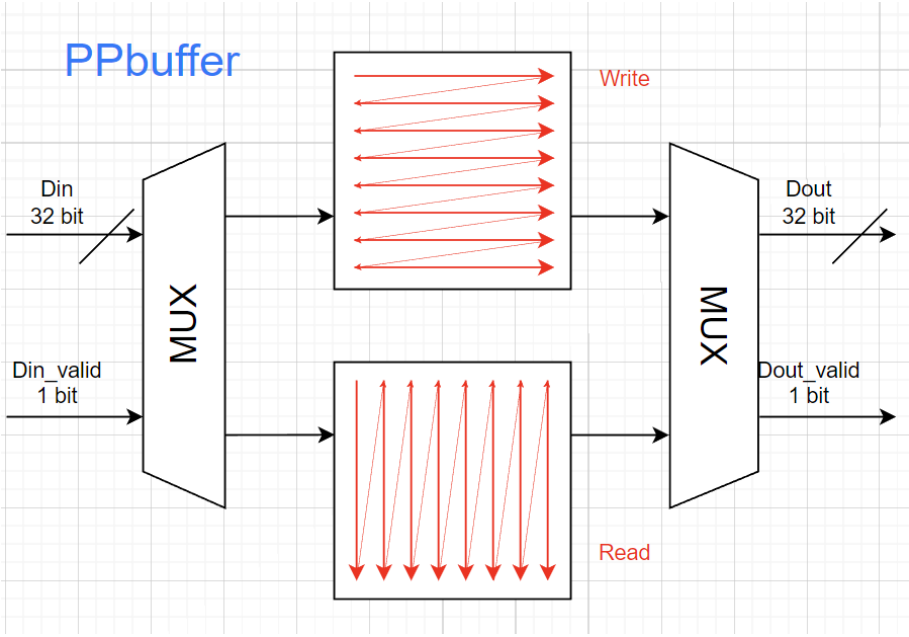


Figure 3.13: The details of PPbuffer

Shift register accepts input sequence and output three value for predictions and one value for updating even element as shown in figure 3.11. The input is firstly shift in three value, the three registers on the top is the three value for odd output. The last value is then delayed till the odd

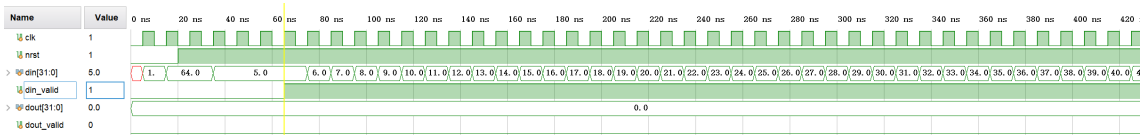


Figure 3.14: The 2D DWT module simulation at the beginning

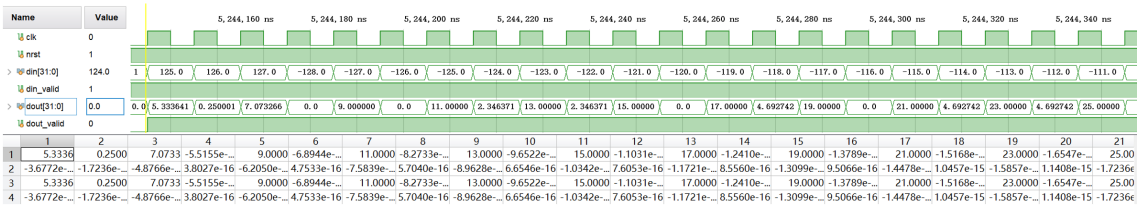


Figure 3.15: The 2D DWT module simulation result

new was calculated. The even value is then output to match with modified odd value. ALU is designed to include the most commonly used calculation in prediction and update. It is shown in figure 3.12. The three square are registers in shift register. An addition, multiplication and an addition can represents both predict and update. It is abstracted out for simplicity. Although there are different types of ALU, the basic idea is the same. Delay module uses several register and non blocking assignment to delay the date which is quite simple.

The simulation of the 2D DWT is shown in the figure3.14 and 3.15. The input is a signed 8 bit number which starts from 5 and increase by one every cycle, Till it reach 127, it will becomes -128 and starts increasing back to 5 and starts a new cycle. The output is also simulated by Matlab and the result is shown in the figure. The result of simulation and Matlab simulation shows that the calculation is correct. Due to the precision of 32 bit, some coefficients that close to 0 may not express correctly. If higher precision is needed, the data type can be extended to double for more accuracy.

3.2.2 Ping Pang Buffer

This section explains the working principle of Ping Pang Buffer (PPbuffer) used in the 2D DWT process. The code for PPbuffer is shown in appendix G. The DWT PPbuffer employed here is different from the one used in preprocessing. The detail implementation diagram is shown in figure 3.13. The input is selected to be written into buffer A at phase 1. At the mean time, Data reading out from the buffer B and the selections of reading is also controlled by a multiplexer. Different from the normal PPbuffer, the reading and writing order is sequential, the reading order of ping pang buffer used here is horizontal. For the configuration of ip which is used here, a change

of write depth from 131072 to 262143 is enough. The reason why to use this order is that for horizontal DWT, sequential order is intuitive, However, for vertical 1D DWT, the input order should be changed to horizontal column as shown in the figure. Otherwise, it's impossible to apply the vertical 1D DWT. By doing so, the order of input data in next stage is changed to vertical direction which after applying the same DWT procedure as horizontal step achieve the vertical DWT without implementing new module and reuse the existing module. The way to change the order back to row wise is to perform the PPbuffer again.

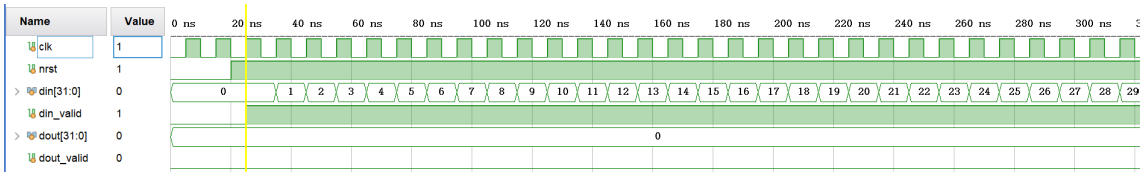


Figure 3.16: The start of the ppbuffer simulation

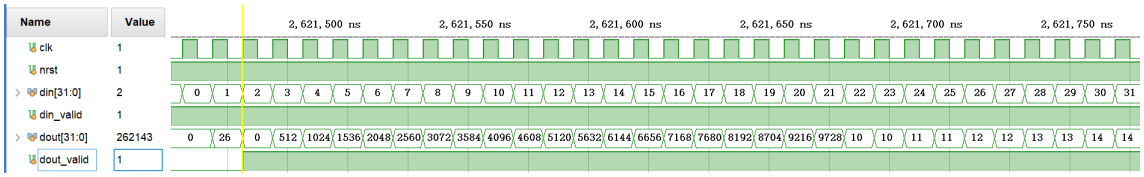


Figure 3.17: The middle of PPbuffer simulation

The simulation is shown in the figure 3.16 and 3.17. In first figure, the ppbuffer start write buffer A and read from buffer B where wea is high and web is low. Dout keep outputting zero at the beginning where the output from buffer b is constantly zero. At the transition point where the wea and web is toggled, the output will begin to give the output from buffer a which starts from 0 to 262143.

The final 2D DWT module is shown below: It composes both horizontal 1D DWT, vertical 1D DWT and two intermediate PPbuffer. The input counter din_cnt starts from 0 to 262143. The output counter dout_cnt is auto increment by 512 to 512*511 and then go back to 1 to $512 \times 511 + 1$ and so on. Till the last column which starts from 511 to $512 \times 511 + 511$ which is the pattern shown in the bottom buffer in figure 3.13.

```
1    module DWT_2D(
2    input clk,
3    input nrst,
4    input [31:0] din,
5    input din_valid,
6    output [31:0] dout,
7    output dout_valid
```

```
    );

//1d_row
wire [31:0] dout_1d_row;
wire dout_1d_row_valid;
DWT_1D dwt_1d_inst1(
    .clk(clk),
    .nrst(nrst),
    .din(din),
    .din_valid(din_valid),
    .dout(dout_1d_row),
    .dout_valid(dout_1d_row_valid)
);

//ppb1
wire [31:0] dout_ppb_1;
wire dout_ppb_1_valid;
pp_buf
 ppb_1(
    .clk(clk),
    .nrst(nrst),
    .din(dout_1d_row),
    .din_valid(dout_1d_row_valid),
    .dout(dout_ppb_1),
    .dout_valid(dout_ppb_1_valid)
);

//1d_col
wire [31:0] dout_1d_col;
wire dout_1d_col_valid;
DWT_1D dwt_1d_inst2(
    .clk(clk),
    .nrst(nrst),
    .din(dout_ppb_1),
    .din_valid(dout_ppb_1_valid),
    .dout(dout_1d_col),
    .dout_valid(dout_1d_col_valid)
```

```
45 );
46
47 //ppb2
48 pp_buf ppb_2(
49     .clk(clk),
50     .nrst(nrst),
51     .din(dout_1d_col),
52     .din_valid(dout_1d_col_valid),
53     .dout(dout),
54     .dout_valid(dout_valid)
55 );
56
57 endmodule
```


3.3 Synthesis and implementation

The module is synthesised report and the resource used is shown in the figure. The FPGA board used to synthesis and implement is AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit which has abundant resource which is suitable for investigation on the resource and performance.

Resource	Estimation	Available	Utilization %
LUT	1172	1182240	0.10
LUTRAM	400	591840	0.07
FF	1561	2364480	0.07
IO	68	832	8.17
BUFG	1	1800	0.06

Figure 3.18: The resource used after synthesis

The resource used in both post synthesis and post implementation for 2D DWT is shown in figure 3.18 and 3.19. The bram is where the Ping Pang buffer utilizes. The two buffer in PPbuffer uses $32bit \times 2^{18} \times 2 = 16,777,216$ bits to store the data. The size of bram used in the board is 18k bits per bram. It uses $\frac{16,777,216}{18,000} = 932$ brams to fully store the whole data. There are 928 Brams used which is similar to the expectation. Some of the brams is optimized to other form of resource.

Resource	Utilization	Available	Utilization %
LUT	12808	1182240	1.08
LUTRAM	1248	591840	0.21
FF	10427	2364480	0.44
BRAM	928	2160	42.96
DSP	28	6840	0.41
IO	68	832	8.17
BUFG	1	1800	0.06

Figure 3.19: The resource used after implementation

There are 28 DSP used which is consumed by the calculation of DWT. For each 1D DWT, there are 2 half 1D DWT. In each 1D DWT, there are 3 DSP for odd ALU, 3 DSP for Even ALU, 12 in total for whole prediction and update and 2 more DSP for the scaling part. Therefore, 14 DSP are used in 1D DWT. 28 DSP for 2D DWT which is the same as the report. The timing report is shown in figure 3.20. All the requirement are met.

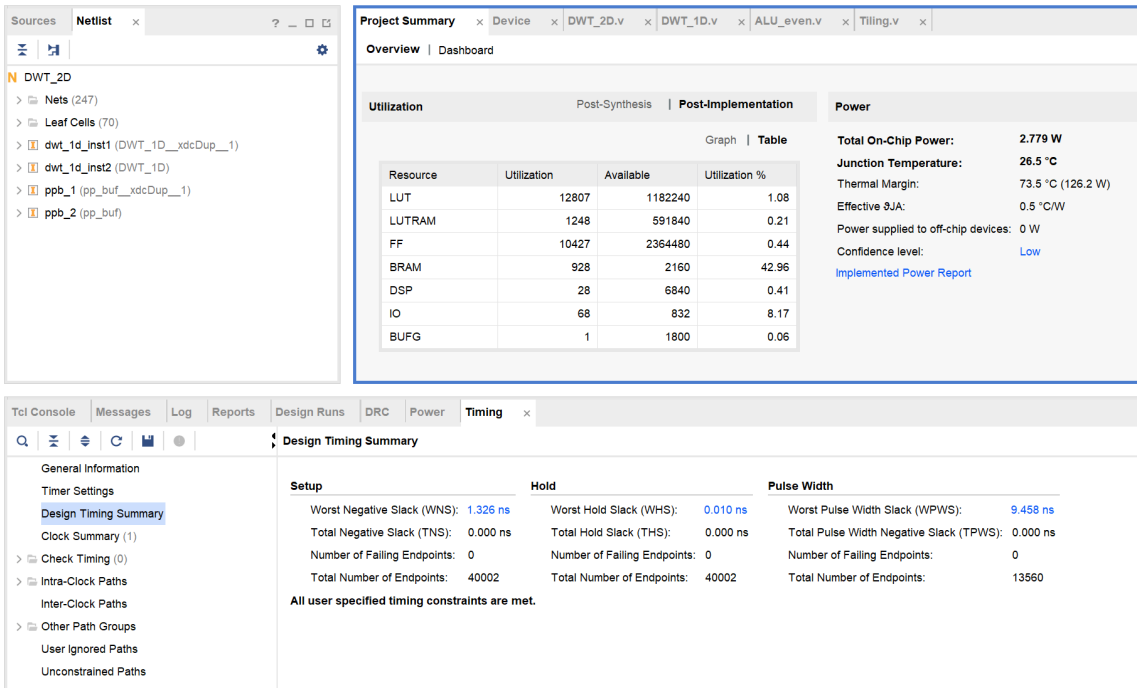


Figure 3.20: The timing report of the 2D DWT design

Conclusions

This chapter concludes the results of this project and reflects on the improvement that can be made if the project starts again.

This project first investigates the performance of the JPEG and JPEG2000 standards in many aspects including their PSNR and SSIM in low, middle, and high compression ratios. The intuitive perception of human eyes is also investigated and compared to illustrate the effect of the human visual system in image compression.

Before comparing the performance, the compared parameters are explained in detail and a corresponding method of calculation is given. Besides, The comparison results show that if the file size is limited and the same, JPEG2000 has slightly better results in PSNR and SSIM in low compression ratio when the file size is the same. By observing the pictures using eyes, there are no obvious artifacts shown in JPEG. At the middle compression ratio, JPEG2000 has a smaller advantage over JPEG2000 in PSNR but a huge advantage in SSIM. Observation shows that it turns out some noticeable artifacts on the edge of each block. However, the JPEG2000 standard image has no artifacts with higher similarity to the original image. At a high compression ratio, the advantages of JPEG2000 stand over JPEG. The PSNR and SSIM of JPEG2000 both exceed the JPEG and from the human visual perspective, the image quality of JPEG2000 is much better than JPEG from the general view of point. From this point, it can be concluded that JPEG2000 has much better image quality in PSNR and SSIM than JPEG when the compression rate is high. Besides, it doesn't have artifacts which is obvious in JPEG at a high compression rate. Therefore, JPEG2000 has better overall image quality at any compression rate.

Secondly, the algorithm of JPEG2000 is illustrated in the algorithm and software verification chapter in detail, which includes four main parts which are preprocessing, DWT, Quantization and EBCOT coding. The preprocessing is the necessary operation required before performing DWT. DWT is a mathematical transformation that can analyze energy in the spatial frequency domain. Due to the energy compaction property of DWT, it will gather the energy into low-frequency components which can reduce the bit rate needed to store the image information. Quantization is the lossy step to further reduce the stored information at a certain level. Finally, the EBCOT

algorithm is another coding algorithm to compact the entropy stored in each coefficient to reduce the storage space to the theoretical minimum. It consists of two parts, tier 1 and tier 2. Tier 1 does the bitplane coding and MQ arithmetic encoder. Tier2 does the bitstream formation which forms the final JP2 file. The DWT part is verified using the open source platform OPENJPEG and the difference of transformed images between self-decomposed and the one from open source is over 148 dB in PSNR which convinces the correctness.

Finally, the hardware implementation of the preprocessing, 2D DWT is introduced which includes tiling, DC level shifting, 2D DWT, and PPbuffer. The detailed implementation of each module is introduced and its corresponding testbench is given. The simulation result of each module is also shown.

For the synthesis and implementation of the design, the timing and resource report are shown and the resource usage is similar or the same as calculated.

In terms of future improvement, the hardware design of EBCOT needs to be proposed and carefully designed. Regarding the image used in this project, an extension on the size of the image can be carried out such as adding the support of arbitrary image shape because the image is not always in square. According to the resource report, optimization of processing space is needed otherwise Bram usage will be the biggest barrier in deploying the system on resource-limited devices. If the whole design of JPEG2000 is achieved, further comparison of the ease of implementation and its corresponding compression performance can be done with the JPEG design proposed in the [1]. Due to the fact that JPEG2000 is more complex than JPEG, it will take a longer time and it's meaningful to investigate the latency difference and there is the possibility of tailing some unnecessary functions to improve the processing speed which can adapt to the specific application. Another point is that the processing space requirement of JPEG2000 is quite high which makes JPEG2000 not a suitable portable application. Further improvement can be done on the reducing of processing space that it requires.

A

The data of file size in bytes for different compression rate, decomposition level and tile size for JPEG2000

A

Table A.1: The File size of JPEG2000 for various size of CR and DL with tile size of 128 * 128

File size (Bytes)	DL							
CR	1	2	3	4	5	6	7	8
1	140067	134119	133084	133121	133312	133494	133687	133702
2	130520	130281	130618	130644	130631	130621	130397	130412
3	86831	86898	86942	86845	86908	86825	86948	86952
4	64888	65056	64915	64961	64707	64895	65072	65083
5	51865	51978	51898	51982	51973	51981	51911	51922
6	43242	43080	43214	43240	43087	43206	43143	43154
7	36965	37011	36979	37020	37019	36911	36901	36912
8	32283	32336	32318	32311	32293	32165	32327	32338
9	28519	28443	28691	28686	28558	28647	28692	28619
10	25730	25771	25696	25733	25771	25786	25719	25729
11	23382	23360	23377	23387	23369	23393	23345	23361
12	21315	21393	21368	21402	21274	21396	21326	21344
13	19668	19582	19657	19706	19659	19722	19589	19607
14	18159	18289	18281	18250	18290	18285	18207	18225
15	16928	17014	17046	17023	17044	17015	17037	17020
16	15897	15915	15939	15915	15940	15876	15742	15764
17	14863	14973	14935	14940	14972	14978	14843	14864
18	14071	14131	14089	14078	14127	14033	14090	14112
19	13310	13360	13303	13288	13304	13295	13353	13360
20	12362	12566	12675	12666	12664	12677	12676	12678
21	11994	12055	12041	12056	12051	12027	12045	12048
22	11379	11402	11435	11439	11455	11487	11386	11402
23	10791	10970	10968	10873	10955	10961	10927	10943
24	10364	10467	10484	10489	10403	10388	10473	10489
25	9689	10045	10047	10015	10057	10057	10038	10052
26	9689	9885	9882	9896	9911	9883	9896	9910
27	9187	9534	9461	9516	9455	9534	9529	9538
28	9187	9149	9154	9178	9190	9190	9180	9145
29	8842	8849	8862	8854	8821	8868	8865	8863
30	8496	8562	8563	8547	8557	8561	8529	8544
31	8022	8278	8274	8282	8282	8260	8281	8276
32	7860	7996	8021	8017	8012	7999	8011	8008
33	7510	7766	7765	7764	7765	7733	7703	7717
34	7510	7530	7460	7535	7523	7539	7519	7533
35	7282	7310	7279	7298	7318	7301	7312	7309
36	6931	7101	7100	7053	7108	7109	7105	7109
37	6861	6829	6913	6912	6909	6907	6901	6892
38	6649	6711	6718	6719	6726	6723	6684	6700
39	6318	6461	6547	6498	6530	6532	6549	6542
40	6318	6360	6363	6375	6371	6376	6380	6377
41	6190	6144	6211	6221	6214	6211	6221	6200
42	6035	6037	6036	6068	6065	6068	6065	6052
43	5878	5851	5906	5898	5923	5916	5925	5902
44	5782	5773	5762	5781	5774	5780	5783	5763
45	5614	5633	5627	5646	5653	5640	5632	5647
46	5495	5498	5516	5524	5524	5521	5521	5525
47	5305	5376	5400	5401	5405	5403	5398	5405
48	5183	5195	5289	5268	5290	5290	5285	5284
49	5148	5094	5174	5178	5174	5174	5173	5178
50	5064	5060	5066	5063	5067	5068	5063	5053

File size (Bytes)	DL							
CR	1	2	3	4	5	6	7	8
51	4936	4960	4955	4964	4968	4968	4969	4963
52	4869	4855	4862	4870	4866	4862	4861	4842
53	4754	4772	4771	4773	4771	4775	4771	4770
54	4610	4678	4679	4672	4682	4674	4683	4682
55	4489	4585	4587	4580	4592	4593	4563	4579
56	4489	4507	4500	4503	4509	4494	4510	4507
57	4262	4348	4400	4412	4419	4411	4428	4427
58	4262	4348	4311	4322	4333	4343	4348	4345
59	4262	4272	4255	4266	4251	4267	4259	4270
60	3969	4142	4195	4194	4198	4198	4189	4194
61	3969	4123	4121	4120	4123	4107	4116	4125
62	3969	3950	4055	4055	4054	4055	4057	4057
63	3969	3950	3975	3989	3990	3990	3977	3990
64	3969	4142	4177	4176	4179	4177	4178	4169
65	3969	4116	4107	4106	4117	4107	4116	4103
66	3969	3950	4055	4055	4054	4055	4044	4055
67	3969	3950	3975	3995	3990	3997	3977	3993
68	3778	3836	3933	3938	3940	3935	3938	3927
69	3778	3836	3877	3880	3880	3883	3879	3881
70	3778	3732	3824	3828	3810	3829	3829	3823
71	3543	3732	3774	3761	3775	3763	3777	3773
72	3543	3699	3721	3724	3716	3717	3723	3724
73	3543	3673	3667	3662	3675	3659	3671	3674
74	3543	3593	3615	3620	3609	3616	3619	3627
75	3543	3563	3578	3566	3577	3580	3576	3574
76	3443	3529	3531	3530	3533	3533	3534	3528
77	3443	3486	3489	3471	3488	3469	3482	3460
78	3443	3427	3443	3436	3444	3426	3443	3444
79	3388	3352	3388	3403	3398	3400	3399	3391
80	3248	3352	3361	3352	3360	3332	3361	3354
81	3248	3262	3320	3321	3318	3298	3317	3321
82	3248	3262	3268	3277	3273	3256	3263	3273
83	3153	3214	3240	3242	3233	3228	3242	3236
84	3153	3166	3194	3203	3179	3205	3191	3201
85	3153	3166	3164	3167	3145	3168	3164	3169
86	2802	3117	3116	3129	3132	3114	3132	3131
87	2802	3094	3088	3096	3096	3093	3093	3091
88	2802	2984	3058	3062	3063	3042	3062	3062
89	2802	2984	3024	3028	3030	3028	3024	3030
90	2802	2984	2990	2988	2968	2992	2988	2997
91	2802	2953	2949	2963	2947	2964	2964	2965
92	2802	2882	2932	2934	2931	2926	2926	2930
93	2802	2882	2898	2901	2894	2891	2897	2901
94	2802	2867	2864	2831	2867	2860	2873	2870
95	2802	2772	2816	2831	2842	2843	2844	2838
96	2802	2772	2795	2810	2814	2810	2809	2805
97	2763	2772	2785	2756	2778	2782	2784	2786
98	2693	2754	2750	2756	2759	2753	2747	2758
99	2693	2724	2720	2729	2731	2729	2729	2729
100	2693	2706	2699	2701	2695	2705	2692	2703

A

Table A.2: The PSNR of JPEG2000 for various size of CR and DL with tile size of 128 * 128

PSNR(dB)	DL							
CR	1	2	3	4	5	6	7	8
1	55.63	55.23	55.11	54.99	54.89	54.73	54.65	54.65
2	53.65	54.40	54.60	54.48	54.36	54.20	54.05	54.05
3	47.29	47.97	48.09	48.05	48.00	47.95	47.92	47.92
4	43.96	44.49	44.56	44.55	44.48	44.47	44.47	44.47
5	42.09	42.86	42.98	42.98	42.95	42.91	42.86	42.86
6	40.89	41.56	41.68	41.68	41.62	41.61	41.56	41.56
7	39.81	40.71	40.81	40.80	40.77	40.73	40.69	40.69
8	38.84	39.93	40.06	40.04	40.00	39.94	39.94	39.94
9	37.99	39.27	39.44	39.43	39.37	39.35	39.32	39.31
10	37.43	38.82	38.95	38.95	38.91	38.89	38.84	38.84
11	36.96	38.30	38.55	38.54	38.50	38.46	38.40	38.40
12	36.34	37.88	38.10	38.10	38.02	38.01	37.95	37.95
13	35.67	37.48	37.71	37.71	37.65	37.63	37.55	37.55
14	34.98	37.18	37.41	37.38	37.34	37.30	37.24	37.24
15	34.46	36.75	37.09	37.06	37.00	36.92	36.87	36.86
16	34.04	36.38	36.71	36.67	36.61	36.53	36.42	36.42
17	33.63	36.05	36.37	36.34	36.29	36.23	36.12	36.12
18	33.33	35.76	36.08	36.05	36.00	35.92	35.87	35.88
19	33.03	35.42	35.83	35.80	35.74	35.68	35.64	35.63
20	32.69	35.08	35.61	35.59	35.52	35.45	35.38	35.37
21	32.49	34.87	35.35	35.35	35.26	35.18	35.12	35.11
22	32.09	34.60	35.10	35.08	35.00	34.94	34.82	34.82
23	31.69	34.42	34.90	34.82	34.78	34.70	34.60	34.60
24	31.41	34.18	34.67	34.64	34.51	34.41	34.38	34.38
25	30.97	33.99	34.46	34.41	34.34	34.26	34.18	34.18
26	30.97	33.91	34.38	34.35	34.27	34.18	34.11	34.11
27	30.62	33.71	34.18	34.17	34.05	34.00	33.92	33.91
28	30.62	33.46	34.03	34.00	33.91	33.80	33.69	33.65
29	30.37	33.26	33.86	33.82	33.66	33.58	33.48	33.46
30	30.12	33.06	33.65	33.60	33.48	33.37	33.25	33.25
31	29.76	32.87	33.45	33.42	33.29	33.17	33.09	33.08
32	29.62	32.67	33.28	33.24	33.12	33.00	32.91	32.90
33	29.36	32.49	33.11	33.08	32.95	32.82	32.71	32.71
34	29.36	32.28	32.91	32.93	32.79	32.70	32.59	32.59
35	29.20	32.09	32.79	32.77	32.66	32.55	32.46	32.45
36	28.92	31.92	32.67	32.61	32.53	32.43	32.31	32.31
37	28.86	31.69	32.55	32.53	32.40	32.28	32.15	32.14
38	28.69	31.60	32.40	32.39	32.26	32.13	31.99	31.99
39	28.43	31.40	32.26	32.21	32.10	31.98	31.88	31.86
40	28.43	31.31	32.11	32.12	31.98	31.85	31.73	31.72
41	28.25	31.15	31.99	32.00	31.85	31.71	31.60	31.57
42	28.04	31.06	31.84	31.88	31.73	31.59	31.46	31.43
43	27.78	30.92	31.74	31.73	31.61	31.45	31.34	31.31
44	27.61	30.86	31.63	31.64	31.47	31.33	31.21	31.17
45	27.31	30.73	31.52	31.51	31.37	31.21	31.06	31.06
46	27.09	30.63	31.44	31.41	31.26	31.09	30.96	30.95
47	26.74	30.52	31.33	31.30	31.14	30.98	30.83	30.83
48	26.52	30.37	31.24	31.18	31.04	30.87	30.72	30.70
49	26.45	30.28	31.14	31.09	30.93	30.76	30.60	30.59
50	26.21	30.25	31.03	30.99	30.82	30.64	30.47	30.44

PSNR(dB)	DL							
CR	1	2	3	4	5	6	7	8
51	25.85	30.14	30.93	30.89	30.71	30.53	30.36	30.34
52	25.65	30.02	30.84	30.79	30.60	30.41	30.22	30.18
53	25.33	29.93	30.75	30.68	30.49	30.31	30.12	30.10
54	24.93	29.82	30.65	30.57	30.39	30.18	30.02	30.00
55	24.62	29.69	30.55	30.46	30.28	30.09	29.87	29.87
56	24.62	29.58	30.45	30.37	30.17	29.97	29.80	29.78
57	24.04	29.39	30.31	30.25	30.07	29.87	29.70	29.68
58	24.04	29.39	30.21	30.15	29.96	29.78	29.60	29.58
59	24.04	29.27	30.14	30.09	29.87	29.69	29.49	29.48
60	23.19	29.10	30.07	30.00	29.80	29.60	29.39	29.38
61	23.19	29.07	29.98	29.91	29.70	29.49	29.30	29.30
62	23.19	28.83	29.90	29.83	29.62	29.41	29.23	29.20
63	23.19	28.83	29.79	29.74	29.54	29.33	29.11	29.11
64	23.19	29.10	30.04	29.97	29.78	29.57	29.38	29.35
65	23.19	29.06	29.96	29.89	29.69	29.49	29.30	29.26
66	23.19	28.83	29.90	29.83	29.62	29.41	29.21	29.20
67	23.19	28.83	29.79	29.75	29.54	29.34	29.11	29.11
68	22.63	28.67	29.73	29.68	29.47	29.26	29.06	29.03
69	22.63	28.67	29.66	29.61	29.39	29.19	28.97	28.95
70	22.63	28.53	29.59	29.55	29.31	29.11	28.91	28.89
71	21.94	28.53	29.53	29.45	29.25	29.02	28.84	28.81
72	21.94	28.47	29.45	29.40	29.18	28.95	28.76	28.75
73	21.94	28.43	29.36	29.32	29.12	28.88	28.69	28.68
74	21.94	28.31	29.29	29.26	29.03	28.82	28.60	28.60
75	21.94	28.26	29.23	29.18	28.97	28.77	28.53	28.52
76	21.67	28.20	29.16	29.14	28.92	28.71	28.46	28.43
77	21.67	28.13	29.11	29.04	28.87	28.59	28.36	28.31
78	21.67	28.04	29.04	28.99	28.80	28.52	28.30	28.28
79	21.50	27.94	28.95	28.96	28.74	28.47	28.22	28.19
80	21.13	27.94	28.91	28.89	28.69	28.35	28.17	28.14
81	21.13	27.81	28.83	28.84	28.61	28.30	28.11	28.10
82	21.13	27.81	28.75	28.78	28.53	28.23	28.01	28.01
83	20.85	27.73	28.70	28.73	28.47	28.17	27.98	27.96
84	20.85	27.66	28.63	28.66	28.37	28.15	27.90	27.90
85	20.85	27.66	28.59	28.60	28.31	28.10	27.86	27.86
86	19.97	27.59	28.51	28.53	28.29	28.00	27.82	27.80
87	19.97	27.55	28.47	28.47	28.22	27.97	27.76	27.74
88	19.97	27.38	28.42	28.41	28.17	27.89	27.71	27.70
89	19.97	27.38	28.36	28.35	28.12	27.87	27.65	27.65
90	19.97	27.38	28.30	28.28	28.02	27.81	27.59	27.59
91	19.97	27.32	28.23	28.23	27.99	27.77	27.54	27.53
92	19.97	27.21	28.21	28.19	27.96	27.71	27.47	27.47
93	19.97	27.21	28.16	28.15	27.90	27.66	27.43	27.42
94	19.97	27.18	28.11	28.03	27.86	27.60	27.36	27.34
95	19.97	27.03	28.02	28.03	27.82	27.57	27.30	27.27
96	19.97	27.03	27.99	28.00	27.77	27.52	27.24	27.21
97	19.87	27.03	27.98	27.91	27.72	27.46	27.19	27.18
98	19.71	27.00	27.92	27.91	27.69	27.42	27.13	27.13
99	19.71	26.95	27.87	27.87	27.64	27.35	27.09	27.07
100	19.71	26.92	27.84	27.83	27.58	27.30	27.01	27.01

A

Table A.3: The SSIM of JPEG2000 for various size of CR and DL with tile size of 128 * 128

SSIM	DL							
CR	1	2	3	4	5	6	7	8
1	0.9985	0.9983	0.9983	0.9983	0.9982	0.9982	0.9982	0.9982
2	0.9978	0.9981	0.9981	0.9981	0.9981	0.9981	0.9980	0.9980
3	0.9903	0.9919	0.9921	0.9921	0.9920	0.9919	0.9919	0.9919
4	0.9781	0.9802	0.9805	0.9805	0.9803	0.9803	0.9803	0.9803
5	0.9680	0.9722	0.9730	0.9731	0.9729	0.9728	0.9726	0.9726
6	0.9579	0.9628	0.9638	0.9638	0.9631	0.9631	0.9629	0.9629
7	0.9512	0.9556	0.9564	0.9564	0.9560	0.9557	0.9555	0.9555
8	0.9426	0.9497	0.9502	0.9501	0.9499	0.9496	0.9495	0.9495
9	0.9340	0.9445	0.9457	0.9455	0.9451	0.9449	0.9447	0.9445
10	0.9264	0.9401	0.9414	0.9414	0.9411	0.9409	0.9403	0.9404
11	0.9198	0.9351	0.9365	0.9367	0.9362	0.9360	0.9358	0.9358
12	0.9155	0.9313	0.9332	0.9332	0.9327	0.9325	0.9322	0.9322
13	0.9092	0.9269	0.9298	0.9295	0.9289	0.9286	0.9278	0.9278
14	0.9001	0.9235	0.9264	0.9261	0.9258	0.9253	0.9243	0.9243
15	0.8921	0.9202	0.9229	0.9226	0.9221	0.9214	0.9210	0.9208
16	0.8837	0.9168	0.9194	0.9192	0.9185	0.9178	0.9172	0.9173
17	0.8764	0.9131	0.9165	0.9163	0.9160	0.9150	0.9142	0.9142
18	0.8712	0.9091	0.9130	0.9128	0.9121	0.9113	0.9107	0.9107
19	0.8642	0.9059	0.9095	0.9093	0.9080	0.9074	0.9069	0.9069
20	0.8563	0.9003	0.9062	0.9060	0.9052	0.9045	0.9040	0.9040
21	0.8548	0.8968	0.9035	0.9034	0.9029	0.9022	0.9014	0.9013
22	0.8503	0.8927	0.9011	0.9009	0.8998	0.8991	0.8978	0.8978
23	0.8459	0.8899	0.8984	0.8977	0.8971	0.8962	0.8946	0.8946
24	0.8421	0.8874	0.8960	0.8950	0.8939	0.8930	0.8925	0.8925
25	0.8346	0.8838	0.8934	0.8930	0.8920	0.8909	0.8888	0.8888
26	0.8346	0.8826	0.8925	0.8922	0.8910	0.8888	0.8880	0.8880
27	0.8287	0.8807	0.8887	0.8888	0.8870	0.8862	0.8853	0.8852
28	0.8287	0.8774	0.8863	0.8863	0.8850	0.8840	0.8820	0.8814
29	0.8221	0.8754	0.8844	0.8841	0.8814	0.8803	0.8793	0.8791
30	0.8160	0.8730	0.8810	0.8806	0.8793	0.8776	0.8764	0.8763
31	0.8090	0.8707	0.8784	0.8783	0.8767	0.8754	0.8742	0.8739
32	0.8049	0.8671	0.8764	0.8762	0.8744	0.8726	0.8717	0.8713
33	0.7992	0.8649	0.8740	0.8738	0.8723	0.8705	0.8683	0.8683
34	0.7992	0.8625	0.8713	0.8718	0.8701	0.8683	0.8662	0.8662
35	0.7962	0.8594	0.8691	0.8693	0.8675	0.8650	0.8631	0.8630
36	0.7950	0.8562	0.8670	0.8664	0.8645	0.8627	0.8614	0.8614
37	0.7935	0.8512	0.8642	0.8642	0.8623	0.8611	0.8592	0.8591
38	0.7896	0.8492	0.8620	0.8619	0.8606	0.8590	0.8570	0.8571
39	0.7852	0.8457	0.8607	0.8596	0.8588	0.8570	0.8551	0.8547
40	0.7852	0.8440	0.8586	0.8586	0.8570	0.8545	0.8534	0.8532
41	0.7826	0.8401	0.8561	0.8570	0.8546	0.8531	0.8512	0.8506
42	0.7800	0.8373	0.8544	0.8549	0.8534	0.8511	0.8488	0.8483
43	0.7754	0.8334	0.8530	0.8533	0.8514	0.8487	0.8468	0.8460
44	0.7718	0.8318	0.8513	0.8518	0.8492	0.8468	0.8442	0.8439
45	0.7647	0.8296	0.8496	0.8500	0.8474	0.8443	0.8416	0.8416
46	0.7660	0.8280	0.8483	0.8479	0.8448	0.8425	0.8398	0.8398
47	0.7567	0.8250	0.8465	0.8466	0.8436	0.8403	0.8376	0.8376
48	0.7532	0.8217	0.8450	0.8442	0.8416	0.8385	0.8359	0.8357
49	0.7520	0.8190	0.8423	0.8427	0.8395	0.8367	0.8336	0.8334
50	0.7467	0.8186	0.8408	0.8404	0.8376	0.8344	0.8318	0.8314

SSIM	DL							
CR	1	2	3	4	5	6	7	8
51	0.7449	0.8171	0.8385	0.8386	0.8360	0.8325	0.8300	0.8296
52	0.7443	0.8149	0.8365	0.8370	0.8338	0.8311	0.8270	0.8260
53	0.7403	0.8126	0.8353	0.8349	0.8321	0.8287	0.8245	0.8243
54	0.7411	0.8106	0.8335	0.8327	0.8309	0.8260	0.8216	0.8213
55	0.7416	0.8083	0.8320	0.8315	0.8283	0.8241	0.8188	0.8188
56	0.7416	0.8066	0.8305	0.8299	0.8259	0.8206	0.8174	0.8169
57	0.7439	0.8029	0.8276	0.8272	0.8238	0.8187	0.8155	0.8151
58	0.7439	0.8029	0.8251	0.8248	0.8207	0.8168	0.8134	0.8130
59	0.7439	0.8003	0.8239	0.8236	0.8189	0.8152	0.8112	0.8110
60	0.7396	0.7976	0.8214	0.8207	0.8175	0.8134	0.8092	0.8090
61	0.7396	0.7971	0.8198	0.8192	0.8157	0.8110	0.8069	0.8068
62	0.7396	0.7921	0.8181	0.8174	0.8139	0.8099	0.8054	0.8046
63	0.7396	0.7921	0.8162	0.8160	0.8124	0.8080	0.8015	0.8015
64	0.7396	0.7976	0.8208	0.8201	0.8170	0.8129	0.8090	0.8084
65	0.7396	0.7969	0.8194	0.8187	0.8156	0.8110	0.8069	0.8059
66	0.7396	0.7921	0.8181	0.8174	0.8139	0.8099	0.8047	0.8045
67	0.7396	0.7921	0.8162	0.8161	0.8124	0.8082	0.8015	0.8016
68	0.7270	0.7886	0.8152	0.8146	0.8108	0.8057	0.8000	0.7991
69	0.7270	0.7886	0.8138	0.8133	0.8093	0.8040	0.7981	0.7975
70	0.7270	0.7854	0.8123	0.8118	0.8070	0.8014	0.7964	0.7953
71	0.7237	0.7854	0.8105	0.8098	0.8057	0.7991	0.7944	0.7937
72	0.7237	0.7840	0.8083	0.8089	0.8033	0.7974	0.7931	0.7929
73	0.7237	0.7833	0.8057	0.8064	0.8016	0.7953	0.7915	0.7913
74	0.7237	0.7801	0.8038	0.8048	0.7992	0.7939	0.7897	0.7896
75	0.7237	0.7789	0.8028	0.8022	0.7983	0.7931	0.7885	0.7880
76	0.7175	0.7774	0.8012	0.8007	0.7970	0.7918	0.7866	0.7857
77	0.7175	0.7752	0.7993	0.7989	0.7952	0.7896	0.7844	0.7827
78	0.7175	0.7731	0.7980	0.7977	0.7939	0.7884	0.7825	0.7821
79	0.7108	0.7708	0.7966	0.7961	0.7930	0.7869	0.7804	0.7795
80	0.6983	0.7708	0.7955	0.7946	0.7915	0.7842	0.7790	0.7782
81	0.6983	0.7677	0.7938	0.7940	0.7900	0.7825	0.7778	0.7776
82	0.6983	0.7677	0.7923	0.7926	0.7888	0.7805	0.7756	0.7756
83	0.6921	0.7659	0.7917	0.7918	0.7869	0.7794	0.7738	0.7735
84	0.6921	0.7643	0.7897	0.7902	0.7846	0.7787	0.7719	0.7719
85	0.6921	0.7643	0.7887	0.7893	0.7829	0.7777	0.7711	0.7710
86	0.6754	0.7624	0.7868	0.7873	0.7825	0.7755	0.7700	0.7695
87	0.6754	0.7613	0.7863	0.7863	0.7807	0.7737	0.7689	0.7684
88	0.6754	0.7566	0.7850	0.7848	0.7793	0.7719	0.7676	0.7674
89	0.6754	0.7566	0.7834	0.7832	0.7782	0.7717	0.7665	0.7664
90	0.6754	0.7566	0.7815	0.7811	0.7758	0.7701	0.7647	0.7647
91	0.6754	0.7539	0.7799	0.7800	0.7740	0.7692	0.7636	0.7632
92	0.6754	0.7508	0.7792	0.7793	0.7737	0.7678	0.7619	0.7617
93	0.6754	0.7508	0.7769	0.7781	0.7721	0.7668	0.7604	0.7602
94	0.6754	0.7503	0.7756	0.7755	0.7713	0.7654	0.7595	0.7588
95	0.6754	0.7459	0.7736	0.7755	0.7703	0.7646	0.7580	0.7575
96	0.6754	0.7459	0.7718	0.7738	0.7694	0.7631	0.7566	0.7560
97	0.6717	0.7459	0.7716	0.7718	0.7680	0.7617	0.7557	0.7552
98	0.6704	0.7452	0.7700	0.7718	0.7676	0.7602	0.7546	0.7546
99	0.6704	0.7439	0.7691	0.7710	0.7666	0.7593	0.7535	0.7529
100	0.6704	0.7433	0.7682	0.7700	0.7648	0.7580	0.7517	0.7517

A

Table A.4: The File size of JPEG2000 for various size of CR and DL with tile size of 256 * 256

File size (Bytes)	DL							
CR	1	2	3	4	5	6	7	8
1	139860	133535	132225	132122	132167	132228	132280	132334
2	130338	130351	130421	130620	130548	130601	130538	130588
3	86937	86905	86865	86807	86865	86931	86704	86754
4	65020	64963	64984	64888	64935	64986	65035	65087
5	51835	51785	51945	51808	51837	51880	51930	51982
6	43073	43251	43229	43245	43219	43067	43119	43171
7	36897	36837	36831	37014	36821	36881	36935	36987
8	32313	32296	32177	32328	32165	32221	32270	32320
9	28591	28691	28510	28446	28496	28552	28600	28650
10	25532	25618	25706	25606	25656	25712	25759	25722
11	23206	23256	23343	23361	23395	23389	23378	23361
12	21345	21279	21380	21359	21413	21379	21282	21333
13	19726	19711	19601	19619	19726	19544	19593	19644
14	17936	18251	18280	18256	18119	18190	18239	18289
15	16653	16983	17044	17039	17045	17044	17049	17045
16	15675	15877	15886	15930	15853	15911	15934	15903
17	14980	14856	14966	14889	14938	14987	14825	14875
18	13857	14025	14092	14133	14094	14090	14070	14120
19	13085	13338	13317	13207	13256	13313	13363	13230
20	12567	12389	12653	12577	12628	12656	12641	12675
21	12028	11908	12044	12001	12049	12046	12040	12046
22	11198	11422	11459	11488	11464	11470	11477	11456
23	10601	10941	10929	10899	10951	10921	10968	10951
24	10171	10081	10442	10486	10487	10493	10453	10426
25	9921	9919	10036	10050	9963	10019	10049	10047
26	9878	9785	9841	9855	9893	9813	9862	9911
27	9453	9492	9379	9393	9486	9523	9530	9531
28	8994	9174	9111	9078	9152	9086	9137	9186
29	8677	8833	8848	8863	8851	8748	8799	8848
30	8422	8563	8543	8472	8541	8539	8559	8559
31	8155	8222	8265	8240	8257	8285	8242	8253
32	7845	7815	7917	7988	7908	7970	8020	8011
33	7684	7726	7723	7763	7698	7760	7737	7766
34	7335	7531	7536	7469	7531	7513	7410	7458
35	7107	7280	7295	7236	7298	7316	7286	7294
36	7107	7102	7037	7072	7061	7110	7085	7050
37	6755	6883	6906	6831	6890	6899	6879	6906
38	6686	6293	6514	6719	6707	6703	6726	6523
39	6475	6293	6514	6314	6363	6425	6473	6523
40	6145	6293	6372	6314	6363	6308	6356	6342
41	6145	6192	6207	6200	6185	6195	6219	6221
42	6017	5721	6055	6043	6062	6055	6069	6056
43	5863	5721	5915	5912	5909	5914	5924	5885
44	5703	5721	5781	5734	5740	5778	5768	5775
45	5565	5654	5609	5626	5632	5615	5648	5569
46	5516	5228	5494	5520	5518	5473	5520	5517
47	5311	5228	5370	5360	5378	5379	5400	5348
48	5275	5228	5185	5266	5209	5252	5259	5257
49	5119	4838	5084	5106	5173	5161	5168	5178
50	4997	4838	5052	5070	5071	5062	4988	5039

File size (Bytes)	DL							
CR	1	2	3	4	5	6	7	8
51	4960	4838	4961	4953	4894	4944	4955	4968
52	4830	4838	4839	4869	4861	4847	4859	4869
53	4746	4767	4741	4748	4770	4774	4736	4696
54	4629	4645	4670	4682	4676	4601	4645	4660
55	4562	4585	4594	4595	4531	4565	4588	4575
56	4418	4497	4438	4453	4495	4505	4509	4506
57	4418	4307	4402	4417	4412	4413	4425	4423
58	4299	4307	4319	4336	4344	4337	4338	4324
59	4268	4261	4267	4263	4267	4229	4241	4239
60	4071	4194	4165	4195	4170	4195	4190	4185
61	4071	4122	4117	4093	4085	4090	4113	4102
62	4032	3639	4040	4045	4031	4006	4053	4044
63	3777	3639	3980	3968	3948	3969	3958	3982
64	4071	4163	4165	4178	4170	4144	4136	4162
65	4071	4078	4117	4093	4085	4090	4113	4102
66	4032	3639	4040	4045	4031	4006	4053	4044
67	3777	3639	3980	3991	3948	3991	3995	3982
68	3777	3639	3937	3908	3933	3911	3933	3937
69	3777	3639	3878	3871	3853	3841	3698	3747
70	3777	3639	3643	3816	3828	3651	3698	3747
71	3777	3639	3643	3746	3593	3651	3698	3747
72	3589	3639	3643	3556	3593	3651	3698	3712
73	3589	3639	3643	3556	3593	3651	3663	3645
74	3589	3606	3573	3556	3593	3616	3596	3586
75	3355	3484	3573	3556	3580	3549	3537	3553
76	3355	3484	3520	3521	3491	3491	3504	3518
77	3355	3484	3487	3464	3438	3458	3487	3442
78	3355	3350	3444	3411	3438	3441	3443	3442
79	3355	3350	3394	3378	3388	3397	3393	3397
80	3355	3350	3335	3361	3344	3347	3348	3356
81	3255	3018	3288	3317	3294	3318	3307	3305
82	3255	3018	3247	3267	3265	3261	3276	3255
83	3231	3018	3170	3239	3233	3230	3206	3228
84	3089	3018	3170	3185	3180	3160	3202	3187
85	3089	3018	3146	3154	3160	3160	3138	3109
86	3089	3018	3102	3131	3110	3133	3060	3109
87	3089	3018	3083	3084	3082	3092	3060	3087
88	3059	3018	3058	3063	3041	3015	3060	3025
89	2962	3018	3013	3022	2964	3015	3030	3025
90	2962	2970	2966	2945	2964	2990	2976	2979
91	2962	2932	2937	2945	2964	2926	2965	2949
92	2611	2932	2908	2928	2934	2926	2932	2916
93	2611	2875	2882	2880	2880	2884	2872	2880
94	2611	2497	2745	2868	2872	2857	2872	2869
95	2611	2497	2745	2840	2843	2824	2836	2821
96	2611	2497	2745	2799	2783	2788	2811	2805
97	2611	2497	2745	2782	2783	2777	2777	2779
98	2611	2497	2745	2749	2747	2759	2735	2757
99	2611	2497	2649	2720	2722	2730	2732	2622
100	2611	2497	2649	2695	2689	2688	2578	2622

A

Table A.5: The PSNR of JPEG2000 for various size of CR and DL with tile size of 256 * 256

PSNR(dB)	DL							
CR	1	2	3	4	5	6	7	8
1	55.63	55.24	55.12	55.02	54.93	54.81	54.73	54.49
2	53.67	54.56	54.71	54.69	54.59	54.51	54.27	54.24
3	47.35	48.08	48.23	48.22	48.21	48.20	48.12	48.09
4	44.01	44.56	44.69	44.68	44.68	44.67	44.66	44.64
5	42.12	42.93	43.11	43.10	43.10	43.09	43.08	43.07
6	40.90	41.67	41.81	41.83	41.81	41.77	41.77	41.76
7	39.85	40.78	40.91	40.95	40.90	40.90	40.90	40.89
8	38.91	40.02	40.18	40.22	40.17	40.17	40.17	40.16
9	38.05	39.40	39.55	39.55	39.54	39.54	39.53	39.53
10	37.45	38.90	39.09	39.08	39.08	39.08	39.07	39.06
11	36.98	38.38	38.71	38.72	38.72	38.70	38.69	38.68
12	36.45	37.96	38.27	38.29	38.29	38.27	38.24	38.23
13	35.81	37.63	37.87	37.90	37.91	37.85	37.85	37.84
14	34.99	37.31	37.58	37.60	37.56	37.56	37.56	37.55
15	34.44	36.93	37.32	37.34	37.32	37.31	37.30	37.28
16	34.05	36.54	36.96	37.01	36.96	36.96	36.95	36.91
17	33.77	36.20	36.63	36.64	36.63	36.63	36.55	36.55
18	33.35	35.92	36.34	36.39	36.35	36.33	36.30	36.30
19	33.05	35.65	36.09	36.08	36.07	36.08	36.07	36.01
20	32.86	35.21	35.87	35.88	35.87	35.86	35.84	35.83
21	32.65	35.01	35.67	35.69	35.68	35.66	35.64	35.62
22	32.12	34.82	35.43	35.50	35.46	35.44	35.42	35.38
23	31.71	34.60	35.19	35.23	35.22	35.18	35.18	35.15
24	31.43	34.24	34.98	35.05	35.02	34.99	34.95	34.91
25	31.27	34.17	34.80	34.84	34.76	34.77	34.76	34.74
26	31.24	34.11	34.71	34.75	34.74	34.67	34.68	34.67
27	30.96	33.99	34.51	34.54	34.55	34.55	34.53	34.50
28	30.64	33.81	34.38	34.40	34.40	34.34	34.34	34.33
29	30.41	33.57	34.26	34.30	34.26	34.19	34.19	34.18
30	30.22	33.38	34.11	34.12	34.11	34.07	34.06	34.03
31	30.02	33.14	33.93	33.97	33.93	33.90	33.83	33.81
32	29.78	32.87	33.69	33.79	33.68	33.68	33.68	33.64
33	29.64	32.80	33.55	33.63	33.53	33.53	33.48	33.46
34	29.38	32.64	33.42	33.42	33.41	33.36	33.25	33.25
35	29.21	32.43	33.25	33.26	33.25	33.22	33.17	33.14
36	29.21	32.27	33.07	33.14	33.09	33.08	33.03	32.98
37	28.93	32.07	32.99	32.98	32.98	32.94	32.90	32.88
38	28.88	31.55	32.73	32.91	32.86	32.82	32.80	32.64
39	28.71	31.55	32.73	32.65	32.64	32.64	32.64	32.64
40	28.44	31.55	32.64	32.65	32.64	32.57	32.57	32.53
41	28.44	31.46	32.51	32.58	32.53	32.50	32.48	32.45
42	28.26	31.09	32.39	32.48	32.45	32.39	32.36	32.31
43	28.04	31.09	32.28	32.37	32.32	32.28	32.24	32.18
44	27.78	31.09	32.14	32.23	32.20	32.18	32.12	32.09
45	27.54	31.03	31.99	32.14	32.11	32.03	32.01	31.90
46	27.46	30.69	31.90	32.04	32.00	31.90	31.90	31.85
47	27.10	30.69	31.80	31.90	31.87	31.82	31.80	31.70
48	27.03	30.69	31.66	31.82	31.71	31.70	31.66	31.61
49	26.74	30.40	31.56	31.67	31.68	31.61	31.58	31.54
50	26.52	30.40	31.53	31.64	31.59	31.53	31.42	31.42

PSNR(dB)	DL							
CR	1	2	3	4	5	6	7	8
51	26.45	30.40	31.45	31.53	31.43	31.42	31.39	31.36
52	26.08	30.40	31.37	31.47	31.41	31.33	31.30	31.26
53	25.84	30.33	31.28	31.36	31.32	31.27	31.18	31.09
54	25.49	30.19	31.22	31.30	31.23	31.10	31.09	31.06
55	25.31	30.12	31.15	31.22	31.09	31.06	31.04	30.99
56	24.92	30.00	31.00	31.08	31.06	31.01	30.97	30.92
57	24.92	29.78	30.97	31.04	30.99	30.92	30.88	30.82
58	24.61	29.78	30.89	30.98	30.91	30.83	30.77	30.69
59	24.53	29.72	30.82	30.90	30.82	30.69	30.65	30.59
60	24.03	29.63	30.69	30.81	30.70	30.65	30.59	30.53
61	24.03	29.52	30.64	30.68	30.60	30.53	30.50	30.42
62	23.92	28.85	30.54	30.62	30.53	30.42	30.42	30.33
63	23.18	28.85	30.46	30.53	30.42	30.36	30.29	30.25
64	24.03	29.59	30.69	30.79	30.70	30.59	30.53	30.50
65	24.03	29.44	30.64	30.68	30.60	30.53	30.50	30.42
66	23.92	28.85	30.54	30.62	30.53	30.42	30.42	30.33
67	23.18	28.85	30.46	30.55	30.42	30.39	30.33	30.25
68	23.18	28.85	30.40	30.45	30.40	30.28	30.25	30.18
69	23.18	28.85	30.32	30.39	30.29	30.18	29.95	29.95
70	23.18	28.85	30.02	30.32	30.25	29.95	29.95	29.95
71	23.18	28.85	30.02	30.21	29.96	29.95	29.95	29.95
72	22.63	28.85	30.02	29.98	29.96	29.95	29.95	29.91
73	22.63	28.85	30.02	29.98	29.96	29.95	29.91	29.83
74	22.63	28.79	29.93	29.98	29.96	29.91	29.83	29.76
75	21.95	28.60	29.93	29.98	29.94	29.83	29.76	29.72
76	21.95	28.60	29.86	29.94	29.83	29.76	29.72	29.67
77	21.95	28.60	29.82	29.87	29.77	29.72	29.69	29.58
78	21.95	28.36	29.76	29.81	29.77	29.69	29.64	29.58
79	21.95	28.36	29.70	29.76	29.70	29.64	29.58	29.52
80	21.95	28.36	29.62	29.74	29.65	29.58	29.52	29.47
81	21.67	27.87	29.55	29.69	29.59	29.54	29.47	29.41
82	21.67	27.87	29.48	29.62	29.55	29.47	29.43	29.34
83	21.59	27.87	29.39	29.59	29.51	29.43	29.34	29.30
84	21.20	27.87	29.39	29.52	29.44	29.34	29.33	29.24
85	21.20	27.87	29.35	29.48	29.42	29.34	29.24	29.14
86	21.20	27.87	29.28	29.45	29.35	29.30	29.14	29.14
87	21.20	27.87	29.26	29.38	29.31	29.24	29.14	29.12
88	21.13	27.87	29.21	29.36	29.25	29.15	29.14	29.02
89	20.85	27.87	29.13	29.29	29.15	29.15	29.11	29.02
90	20.85	27.78	29.04	29.20	29.15	29.12	29.02	28.94
91	20.85	27.72	29.00	29.20	29.15	29.02	29.00	28.89
92	19.97	27.72	28.95	29.18	29.12	29.02	28.94	28.83
93	19.97	27.63	28.90	29.10	29.03	28.94	28.83	28.77
94	19.97	27.10	28.64	29.08	29.02	28.89	28.83	28.75
95	19.97	27.10	28.64	29.03	28.96	28.83	28.78	28.68
96	19.97	27.10	28.64	28.96	28.85	28.78	28.74	28.65
97	19.97	27.10	28.64	28.93	28.85	28.76	28.68	28.61
98	19.97	27.10	28.64	28.87	28.79	28.73	28.61	28.58
99	19.97	27.10	28.49	28.82	28.75	28.68	28.61	28.34
100	19.97	27.10	28.49	28.78	28.70	28.61	28.34	28.34

A

Table A.6: The SSIM of JPEG2000 for various size of CR and DL with tile size of 256 * 256

SSIM	DL							
CR	1	2	3	4	5	6	7	8
1	0.9985	0.9983	0.9983	0.9983	0.9983	0.9982	0.9982	0.9982
2	0.9978	0.9981	0.9982	0.9982	0.9982	0.9982	0.9981	0.9981
3	0.9904	0.9920	0.9922	0.9922	0.9922	0.9922	0.9922	0.9921
4	0.9783	0.9805	0.9812	0.9812	0.9812	0.9812	0.9811	0.9811
5	0.9683	0.9726	0.9738	0.9738	0.9738	0.9738	0.9738	0.9738
6	0.9579	0.9639	0.9647	0.9648	0.9647	0.9646	0.9646	0.9645
7	0.9512	0.9559	0.9571	0.9575	0.9571	0.9571	0.9571	0.9571
8	0.9433	0.9505	0.9515	0.9518	0.9515	0.9515	0.9516	0.9515
9	0.9350	0.9453	0.9464	0.9464	0.9464	0.9464	0.9464	0.9464
10	0.9266	0.9410	0.9425	0.9425	0.9425	0.9425	0.9425	0.9424
11	0.9199	0.9358	0.9386	0.9389	0.9388	0.9387	0.9386	0.9385
12	0.9162	0.9323	0.9349	0.9350	0.9350	0.9348	0.9346	0.9346
13	0.9106	0.9284	0.9309	0.9311	0.9312	0.9307	0.9307	0.9306
14	0.9003	0.9250	0.9280	0.9281	0.9277	0.9277	0.9277	0.9277
15	0.8920	0.9212	0.9254	0.9255	0.9255	0.9254	0.9254	0.9253
16	0.8840	0.9190	0.9215	0.9224	0.9217	0.9217	0.9217	0.9212
17	0.8796	0.9142	0.9193	0.9194	0.9194	0.9194	0.9188	0.9188
18	0.8715	0.9113	0.9167	0.9175	0.9168	0.9167	0.9161	0.9161
19	0.8646	0.9084	0.9123	0.9122	0.9122	0.9123	0.9122	0.9118
20	0.8605	0.9025	0.9097	0.9098	0.9098	0.9097	0.9094	0.9094
21	0.8562	0.8986	0.9069	0.9070	0.9070	0.9069	0.9067	0.9065
22	0.8507	0.8963	0.9045	0.9053	0.9051	0.9050	0.9048	0.9045
23	0.8462	0.8926	0.9021	0.9025	0.9025	0.9022	0.9021	0.9018
24	0.8424	0.8876	0.8995	0.9008	0.9000	0.9000	0.8993	0.8983
25	0.8391	0.8870	0.8969	0.8975	0.8966	0.8967	0.8966	0.8961
26	0.8388	0.8864	0.8960	0.8966	0.8961	0.8955	0.8954	0.8954
27	0.8347	0.8838	0.8933	0.8938	0.8938	0.8938	0.8936	0.8934
28	0.8291	0.8821	0.8921	0.8925	0.8925	0.8918	0.8918	0.8918
29	0.8226	0.8797	0.8909	0.8914	0.8911	0.8890	0.8890	0.8890
30	0.8187	0.8769	0.8879	0.8881	0.8881	0.8877	0.8876	0.8875
31	0.8145	0.8744	0.8855	0.8864	0.8857	0.8851	0.8841	0.8840
32	0.8094	0.8704	0.8823	0.8834	0.8825	0.8825	0.8824	0.8817
33	0.8053	0.8688	0.8804	0.8817	0.8803	0.8803	0.8793	0.8791
34	0.7997	0.8675	0.8780	0.8786	0.8786	0.8776	0.8767	0.8767
35	0.7966	0.8646	0.8762	0.8768	0.8768	0.8759	0.8751	0.8746
36	0.7966	0.8625	0.8741	0.8747	0.8742	0.8740	0.8737	0.8732
37	0.7953	0.8596	0.8734	0.8733	0.8733	0.8730	0.8720	0.8719
38	0.7938	0.8478	0.8682	0.8721	0.8715	0.8704	0.8703	0.8670
39	0.7899	0.8478	0.8682	0.8671	0.8671	0.8670	0.8670	0.8670
40	0.7855	0.8478	0.8657	0.8671	0.8671	0.8653	0.8653	0.8644
41	0.7855	0.8465	0.8639	0.8653	0.8644	0.8635	0.8632	0.8628
42	0.7828	0.8393	0.8628	0.8629	0.8628	0.8623	0.8622	0.8617
43	0.7802	0.8393	0.8612	0.8620	0.8618	0.8613	0.8611	0.8600
44	0.7755	0.8393	0.8597	0.8603	0.8601	0.8598	0.8593	0.8591
45	0.7691	0.8382	0.8572	0.8593	0.8591	0.8583	0.8582	0.8567
46	0.7682	0.8294	0.8559	0.8584	0.8575	0.8568	0.8567	0.8556
47	0.7662	0.8294	0.8549	0.8568	0.8556	0.8551	0.8546	0.8535
48	0.7645	0.8294	0.8526	0.8551	0.8535	0.8535	0.8527	0.8515
49	0.7570	0.8208	0.8514	0.8533	0.8527	0.8514	0.8508	0.8500
50	0.7535	0.8208	0.8506	0.8531	0.8509	0.8499	0.8486	0.8486

SSIM	DL							
CR	1	2	3	4	5	6	7	8
51	0.7522	0.8208	0.8485	0.8505	0.8486	0.8486	0.8481	0.8474
52	0.7448	0.8208	0.8473	0.8495	0.8481	0.8467	0.8461	0.8453
53	0.7451	0.8201	0.8461	0.8478	0.8465	0.8453	0.8440	0.8416
54	0.7410	0.8177	0.8449	0.8464	0.8452	0.8416	0.8416	0.8410
55	0.7404	0.8167	0.8430	0.8450	0.8416	0.8411	0.8407	0.8403
56	0.7411	0.8150	0.8395	0.8415	0.8410	0.8405	0.8402	0.8391
57	0.7411	0.8100	0.8389	0.8409	0.8403	0.8390	0.8386	0.8372
58	0.7416	0.8100	0.8374	0.8402	0.8389	0.8374	0.8365	0.8355
59	0.7412	0.8091	0.8365	0.8386	0.8373	0.8356	0.8344	0.8335
60	0.7439	0.8069	0.8342	0.8368	0.8356	0.8344	0.8335	0.8322
61	0.7439	0.8051	0.8334	0.8345	0.8336	0.8323	0.8315	0.8304
62	0.7423	0.7927	0.8313	0.8337	0.8323	0.8305	0.8305	0.8289
63	0.7396	0.7927	0.8303	0.8317	0.8305	0.8294	0.8270	0.8262
64	0.7439	0.8061	0.8342	0.8364	0.8356	0.8336	0.8322	0.8314
65	0.7439	0.8038	0.8334	0.8345	0.8336	0.8323	0.8315	0.8304
66	0.7423	0.7927	0.8313	0.8337	0.8323	0.8305	0.8305	0.8289
67	0.7396	0.7927	0.8303	0.8324	0.8305	0.8299	0.8289	0.8262
68	0.7396	0.7927	0.8294	0.8306	0.8299	0.8270	0.8262	0.8254
69	0.7396	0.7927	0.8266	0.8295	0.8270	0.8254	0.8204	0.8204
70	0.7396	0.7927	0.8208	0.8272	0.8262	0.8204	0.8204	0.8204
71	0.7396	0.7927	0.8208	0.8256	0.8204	0.8204	0.8204	0.8204
72	0.7271	0.7927	0.8208	0.8206	0.8204	0.8204	0.8204	0.8197
73	0.7271	0.7927	0.8208	0.8206	0.8204	0.8204	0.8197	0.8182
74	0.7271	0.7919	0.8192	0.8206	0.8204	0.8197	0.8182	0.8159
75	0.7240	0.7877	0.8192	0.8206	0.8203	0.8182	0.8159	0.8152
76	0.7240	0.7877	0.8169	0.8199	0.8182	0.8160	0.8153	0.8140
77	0.7240	0.7877	0.8162	0.8184	0.8160	0.8153	0.8146	0.8111
78	0.7240	0.7819	0.8144	0.8162	0.8160	0.8147	0.8128	0.8111
79	0.7240	0.7819	0.8127	0.8155	0.8147	0.8129	0.8111	0.8096
80	0.7240	0.7819	0.8110	0.8149	0.8129	0.8112	0.8096	0.8088
81	0.7178	0.7701	0.8088	0.8131	0.8112	0.8102	0.8088	0.8074
82	0.7178	0.7701	0.8077	0.8114	0.8102	0.8088	0.8081	0.8051
83	0.7154	0.7701	0.8044	0.8105	0.8092	0.8081	0.8051	0.8044
84	0.7028	0.7701	0.8044	0.8091	0.8082	0.8051	0.8051	0.8034
85	0.7028	0.7701	0.8038	0.8084	0.8075	0.8051	0.8035	0.8000
86	0.7028	0.7701	0.8025	0.8077	0.8052	0.8044	0.8000	0.8000
87	0.7028	0.7701	0.8023	0.8054	0.8045	0.8034	0.8000	0.7995
88	0.6985	0.7701	0.8010	0.8049	0.8035	0.8000	0.8000	0.7980
89	0.6922	0.7701	0.7994	0.8038	0.8001	0.8000	0.7995	0.7980
90	0.6922	0.7679	0.7981	0.8004	0.8001	0.7995	0.7980	0.7961
91	0.6922	0.7661	0.7960	0.8004	0.8001	0.7980	0.7978	0.7952
92	0.6756	0.7661	0.7945	0.7999	0.7995	0.7980	0.7962	0.7944
93	0.6756	0.7637	0.7935	0.7986	0.7980	0.7962	0.7944	0.7922
94	0.6756	0.7525	0.7873	0.7980	0.7978	0.7952	0.7944	0.7918
95	0.6756	0.7525	0.7873	0.7967	0.7963	0.7944	0.7923	0.7900
96	0.6756	0.7525	0.7873	0.7962	0.7945	0.7922	0.7915	0.7894
97	0.6756	0.7525	0.7873	0.7955	0.7945	0.7918	0.7900	0.7884
98	0.6756	0.7525	0.7873	0.7946	0.7923	0.7914	0.7884	0.7880
99	0.6756	0.7525	0.7832	0.7925	0.7916	0.7900	0.7884	0.7818
100	0.6756	0.7525	0.7832	0.7917	0.7901	0.7884	0.7818	0.7818

A

Table A.7: The file size of JPEG2000 for various size of CR and DL with tile size of 512 * 512

File size (Bytes)	DL							
CR	1	2	3	4	5	6	7	8
1	139696	133311	131885	131707	131712	131736	131755	131772
2	130513	130440	130513	130609	130548	130589	130611	130627
3	86850	86877	86786	86876	86886	86912	86933	86953
4	64944	64765	65094	64940	64947	64973	64990	65009
5	51720	51821	51624	51820	51825	51851	51868	51887
6	43256	43048	43229	43094	43105	43131	43154	43170
7	36859	36897	36975	36840	36851	36877	36896	36912
8	32255	32106	32136	32243	32254	32280	32299	32315
9	28663	28664	28583	28675	28681	28662	28681	28697
10	25485	25580	25462	25734	25740	25766	25785	25394
11	23109	23227	23379	23320	23323	23349	23364	23384
12	21387	21317	21211	21392	21368	21359	21380	21400
13	19693	19661	19599	19709	19539	19530	19548	19567
14	18233	18284	18276	18293	18185	18176	18194	18213
15	17045	17003	17022	16998	17015	17038	17006	17025
16	15622	15916	15892	15906	15923	15945	15807	15825
17	14747	14866	14902	14762	14779	14801	14817	14835
18	13781	14037	14029	14121	14058	14080	14096	14114
19	13019	13295	13350	13210	13227	13249	13265	13283
20	12504	12660	12604	12533	12543	12565	12581	12599
21	12053	11723	12010	11999	12009	12031	12047	11937
22	11157	11238	11448	11462	11467	11481	11457	11474
23	10957	10939	10931	10940	10912	10936	10948	10967
24	10129	10477	10411	10482	10420	10469	10481	10357
25	9932	10007	9882	9904	9923	9972	9984	10002
26	9889	9793	9882	9904	9846	9895	9907	9831
27	9416	9382	9504	9526	9458	9499	9511	9529
28	8960	9175	9133	9168	9187	9146	9158	9176
29	8645	8816	8863	8755	8774	8800	8820	8838
30	8391	8562	8474	8548	8567	8564	8567	8427
31	8280	8221	8267	8204	8223	8249	8267	8285
32	7813	7991	7955	8007	7844	7870	7888	7906
33	7652	7735	7726	7771	7758	7641	7659	7677
34	7303	7514	7516	7522	7530	7482	7500	7518
35	7303	7312	7203	7266	7285	7311	7136	7154
36	6939	7073	7050	7073	7092	7093	6983	7001
37	6870	6857	6858	6895	6907	6853	6871	6889
38	6659	6653	6708	6666	6685	6711	6694	6712
39	6329	6263	6452	6514	6533	6491	6509	6527
40	6329	6263	6239	6190	6209	6235	6253	6271
41	6142	6204	6147	6190	6209	6022	6040	6058
42	5988	5642	6063	5977	5996	6022	6040	6058
43	5814	5642	5906	5885	5904	5873	5891	5909
44	5651	5642	5784	5786	5777	5752	5768	5786
45	5651	5642	5636	5635	5617	5643	5631	5649
46	5513	5483	5464	5507	5494	5522	5489	5507
47	5380	5148	4988	5403	5390	5374	5390	5350
48	5259	5148	4988	5274	5288	5289	5210	5228
49	5065	5148	4988	5062	5174	5169	5121	5139
50	5065	4756	4988	5062	4992	4993	5018	5036

File size (Bytes)	DL							
CR	1	2	3	4	5	6	7	8
51	4942	4756	4925	4896	4914	4915	4940	4957
52	4818	4756	4822	4837	4851	4852	4846	4863
53	4713	4756	4763	4768	4751	4752	4687	4704
54	4629	4668	4604	4609	4661	4662	4587	4604
55	4512	4595	4504	4509	4561	4562	4587	4464
56	4446	4466	4504	4509	4423	4424	4447	4464
57	4402	4276	4366	4371	4423	4424	4384	4401
58	4301	4276	4318	4308	4329	4339	4337	4332
59	4254	4231	4207	4261	4260	4203	4224	4243
60	4027	4175	4194	4150	4171	4173	4194	4198
61	4027	4091	4112	4120	4102	4114	4126	4126
62	4027	4056	4045	4042	4046	4056	4025	4042
63	3987	3571	3957	3958	3979	3977	3937	3954
64	4027	4175	4168	4150	4171	4173	4181	4172
65	4027	4091	4112	4113	4102	4114	4110	4109
66	4027	4056	4045	4042	4046	4056	4025	4042
67	3987	3571	3957	3958	3979	3977	3997	3954
68	3732	3571	3811	3930	3891	3917	3937	3795
69	3732	3571	3811	3870	3878	3758	3778	3795
70	3732	3571	3811	3724	3732	3758	3778	3795
71	3732	3571	3764	3724	3732	3758	3731	3748
72	3544	3571	3574	3724	3685	3711	3541	3558
73	3544	3571	3574	3487	3495	3521	3541	3558
74	3544	3571	3574	3487	3495	3521	3541	3558
75	3544	3571	3574	3487	3495	3521	3541	3558
76	3310	3533	3449	3487	3495	3521	3416	3433
77	3310	3410	3449	3487	3370	3396	3416	3433
78	3310	3410	3404	3362	3370	3396	3416	3433
79	3310	3280	3391	3362	3370	3396	3371	3388
80	3310	3280	3348	3317	3325	3351	3358	3313
81	3310	3280	3315	3317	3312	3276	3296	3313
82	3209	3280	3264	3242	3247	3276	3280	3279
83	3209	2947	3059	3242	3234	3242	3232	3230
84	3185	2947	3059	3199	3186	3193	3180	3197
85	3043	2947	3059	3166	3167	3160	3114	3131
86	3043	2947	3059	3101	3133	3094	3114	3131
87	3043	2947	3059	3050	3068	3094	3063	3080
88	3043	2947	3059	3050	3017	3043	3063	3011
89	3013	2947	3024	2981	3017	2974	2994	3011
90	2917	2947	2979	2981	2948	2974	2994	2806
91	2917	2947	2905	2776	2948	2769	2789	2806
92	2917	2899	2905	2776	2743	2769	2789	2806
93	2566	2899	2899	2776	2743	2769	2789	2806
94	2566	2850	2867	2776	2743	2769	2789	2806
95	2566	2813	2735	2776	2743	2769	2789	2806
96	2566	2813	2735	2776	2743	2769	2789	2806
97	2566	2435	2735	2776	2743	2769	2754	2771
98	2566	2435	2735	2741	2743	2734	2754	2722
99	2566	2435	2648	2696	2708	2731	2706	2722
100	2566	2435	2648	2696	2663	2686	2706	2704

A

Table A.8: The PSNR of JPEG2000 for various size of CR and DL with tile size of 512 * 512

PSNR(dB)	DL							
CR	1	2	3	4	5	6	7	8
1	55.65	55.25	55.13	55.05	54.94	54.86	54.78	54.70
2	53.72	54.62	54.81	54.80	54.69	54.59	54.54	54.54
3	47.37	48.11	48.28	48.31	48.29	48.28	48.27	48.27
4	44.02	44.57	44.76	44.75	44.74	44.73	44.73	44.71
5	42.13	42.97	43.12	43.15	43.15	43.14	43.14	43.13
6	40.95	41.67	41.86	41.86	41.86	41.85	41.85	41.85
7	39.87	40.82	40.98	40.98	40.98	40.97	40.98	40.97
8	38.93	40.03	40.23	40.27	40.27	40.27	40.27	40.27
9	38.10	39.44	39.62	39.66	39.65	39.64	39.64	39.64
10	37.47	38.93	39.11	39.17	39.17	39.17	39.17	39.10
11	36.98	38.42	38.77	38.79	38.78	38.78	38.78	38.78
12	36.50	38.02	38.31	38.39	38.38	38.37	38.37	38.37
13	35.84	37.66	37.94	38.00	37.96	37.95	37.95	37.95
14	35.16	37.36	37.65	37.70	37.66	37.66	37.66	37.65
15	34.64	37.00	37.38	37.41	37.41	37.41	37.40	37.39
16	34.07	36.61	37.06	37.12	37.12	37.12	37.06	37.06
17	33.72	36.25	36.70	36.71	36.71	36.71	36.71	36.71
18	33.36	35.98	36.40	36.49	36.46	36.46	36.46	36.46
19	33.06	35.70	36.18	36.18	36.18	36.18	36.18	36.18
20	32.88	35.39	35.94	35.96	35.96	35.96	35.96	35.96
21	32.70	34.99	35.75	35.79	35.79	35.79	35.79	35.74
22	32.14	34.80	35.54	35.62	35.61	35.61	35.59	35.59
23	31.99	34.66	35.29	35.39	35.37	35.37	35.37	35.37
24	31.45	34.46	35.08	35.18	35.14	35.16	35.15	35.09
25	31.32	34.27	34.85	34.93	34.93	34.94	34.94	34.93
26	31.29	34.18	34.85	34.93	34.90	34.90	34.90	34.85
27	30.98	34.00	34.67	34.75	34.71	34.71	34.71	34.71
28	30.66	33.89	34.51	34.58	34.59	34.56	34.56	34.56
29	30.43	33.65	34.38	34.40	34.40	34.40	34.40	34.40
30	30.24	33.47	34.20	34.30	34.30	34.28	34.28	34.21
31	30.16	33.23	34.10	34.14	34.14	34.14	34.14	34.14
32	29.80	33.07	33.88	34.01	33.89	33.88	33.89	33.88
33	29.66	32.89	33.72	33.85	33.82	33.72	33.72	33.72
34	29.39	32.72	33.57	33.67	33.66	33.61	33.61	33.61
35	29.39	32.55	33.36	33.49	33.49	33.49	33.36	33.36
36	29.09	32.33	33.25	33.36	33.36	33.34	33.25	33.25
37	29.03	32.13	33.12	33.23	33.23	33.17	33.17	33.17
38	28.86	31.94	33.01	33.07	33.08	33.08	33.05	33.05
39	28.58	31.61	32.84	32.97	32.97	32.93	32.93	32.93
40	28.58	31.61	32.70	32.76	32.76	32.76	32.76	32.76
41	28.34	31.56	32.64	32.76	32.76	32.63	32.63	32.63
42	28.12	31.11	32.57	32.62	32.63	32.63	32.63	32.63
43	27.88	31.11	32.44	32.56	32.57	32.54	32.54	32.54
44	27.63	31.11	32.34	32.50	32.49	32.44	32.45	32.45
45	27.63	31.11	32.19	32.37	32.35	32.35	32.33	32.33
46	27.40	30.96	32.04	32.27	32.25	32.25	32.21	32.21
47	27.18	30.71	31.68	32.17	32.16	32.11	32.11	32.07
48	26.97	30.71	31.68	32.06	32.07	32.04	31.96	31.96
49	26.62	30.71	31.68	31.88	31.96	31.93	31.87	31.87
50	26.62	30.42	31.68	31.88	31.78	31.76	31.77	31.77

PSNR(dB)	DL							
CR	1	2	3	4	5	6	7	8
51	26.41	30.42	31.62	31.72	31.70	31.68	31.69	31.69
52	26.10	30.42	31.52	31.66	31.65	31.63	31.61	31.61
53	25.80	30.42	31.46	31.60	31.57	31.55	31.47	31.47
54	25.57	30.33	31.32	31.46	31.48	31.46	31.38	31.38
55	25.25	30.25	31.24	31.37	31.39	31.37	31.38	31.26
56	25.08	30.09	31.24	31.37	31.27	31.25	31.26	31.26
57	24.96	29.86	31.12	31.25	31.27	31.25	31.19	31.19
58	24.71	29.86	31.08	31.18	31.18	31.17	31.15	31.13
59	24.59	29.80	30.98	31.14	31.12	31.04	31.05	31.05
60	24.00	29.72	30.97	31.04	31.04	31.02	31.02	31.01
61	24.00	29.62	30.87	31.02	30.98	30.97	30.96	30.93
62	24.00	29.56	30.79	30.93	30.91	30.89	30.83	30.82
63	23.90	28.88	30.66	30.82	30.82	30.78	30.70	30.70
64	24.00	29.72	30.94	31.04	31.04	31.02	31.01	30.99
65	24.00	29.62	30.87	31.01	30.98	30.97	30.94	30.91
66	24.00	29.56	30.79	30.93	30.91	30.89	30.83	30.82
67	23.90	28.88	30.66	30.82	30.82	30.78	30.79	30.70
68	23.16	28.88	30.48	30.78	30.70	30.70	30.70	30.50
69	23.16	28.88	30.48	30.70	30.68	30.50	30.50	30.50
70	23.16	28.88	30.48	30.51	30.50	30.50	30.50	30.50
71	23.16	28.88	30.41	30.51	30.50	30.50	30.43	30.43
72	22.62	28.88	30.17	30.51	30.43	30.43	30.19	30.19
73	22.62	28.88	30.17	30.20	30.19	30.19	30.19	30.19
74	22.62	28.88	30.17	30.20	30.19	30.19	30.19	30.19
75	22.62	28.88	30.17	30.20	30.19	30.19	30.19	30.19
76	21.93	28.81	30.02	30.20	30.19	30.19	30.04	30.04
77	21.93	28.62	30.02	30.20	30.04	30.04	30.04	30.04
78	21.93	28.62	29.96	30.05	30.04	30.04	30.04	30.04
79	21.93	28.38	29.94	30.05	30.04	30.04	29.97	29.97
80	21.93	28.38	29.88	29.99	29.97	29.97	29.95	29.87
81	21.93	28.38	29.83	29.99	29.95	29.87	29.88	29.87
82	21.66	28.38	29.77	29.89	29.87	29.87	29.85	29.83
83	21.66	27.88	29.50	29.89	29.85	29.83	29.79	29.76
84	21.58	27.88	29.50	29.83	29.79	29.76	29.72	29.72
85	21.20	27.88	29.50	29.79	29.76	29.72	29.64	29.64
86	21.20	27.88	29.50	29.71	29.72	29.64	29.64	29.64
87	21.20	27.88	29.50	29.64	29.64	29.64	29.58	29.58
88	21.20	27.88	29.50	29.64	29.58	29.58	29.58	29.50
89	21.12	27.88	29.45	29.56	29.58	29.50	29.50	29.50
90	20.85	27.88	29.37	29.56	29.50	29.50	29.50	29.25
91	20.85	27.88	29.26	29.31	29.50	29.25	29.25	29.25
92	20.85	27.80	29.26	29.31	29.25	29.25	29.25	29.25
93	19.97	27.80	29.25	29.31	29.25	29.25	29.25	29.25
94	19.97	27.71	29.18	29.31	29.25	29.25	29.25	29.25
95	19.97	27.65	28.91	29.31	29.25	29.25	29.25	29.25
96	19.97	27.65	28.91	29.31	29.25	29.25	29.25	29.25
97	19.97	27.12	28.91	29.31	29.25	29.25	29.20	29.20
98	19.97	27.12	28.91	29.26	29.25	29.20	29.20	29.12
99	19.97	27.12	28.73	29.18	29.20	29.19	29.12	29.12
100	19.97	27.12	28.73	29.18	29.13	29.12	29.12	29.09

A

Table A.9: The SSIM of JPEG2000 for various size of CR and DL with tile size of 512 * 512

SSIM	DL							
CR	1	2	3	4	5	6	7	8
1	0.9985	0.9983	0.9983	0.9983	0.9983	0.9983	0.9982	0.9982
2	0.9978	0.9981	0.9982	0.9982	0.9982	0.9982	0.9982	0.9982
3	0.9905	0.9920	0.9923	0.9924	0.9924	0.9924	0.9924	0.9924
4	0.9783	0.9805	0.9813	0.9813	0.9813	0.9813	0.9813	0.9813
5	0.9682	0.9729	0.9738	0.9740	0.9740	0.9740	0.9740	0.9740
6	0.9585	0.9639	0.9652	0.9653	0.9653	0.9652	0.9653	0.9653
7	0.9515	0.9561	0.9578	0.9578	0.9578	0.9578	0.9578	0.9578
8	0.9434	0.9505	0.9521	0.9523	0.9523	0.9523	0.9523	0.9523
9	0.9352	0.9456	0.9469	0.9475	0.9475	0.9474	0.9474	0.9474
10	0.9267	0.9413	0.9427	0.9432	0.9432	0.9432	0.9432	0.9427
11	0.9200	0.9364	0.9394	0.9396	0.9395	0.9395	0.9396	0.9395
12	0.9166	0.9327	0.9355	0.9357	0.9357	0.9357	0.9357	0.9357
13	0.9108	0.9287	0.9319	0.9325	0.9321	0.9321	0.9321	0.9321
14	0.9025	0.9255	0.9287	0.9291	0.9288	0.9288	0.9288	0.9288
15	0.8956	0.9224	0.9259	0.9260	0.9261	0.9261	0.9261	0.9261
16	0.8842	0.9197	0.9230	0.9237	0.9238	0.9238	0.9231	0.9231
17	0.8774	0.9156	0.9195	0.9197	0.9197	0.9197	0.9197	0.9197
18	0.8716	0.9118	0.9171	0.9182	0.9181	0.9181	0.9181	0.9181
19	0.8647	0.9088	0.9145	0.9145	0.9146	0.9146	0.9146	0.9146
20	0.8607	0.9053	0.9107	0.9109	0.9109	0.9109	0.9110	0.9109
21	0.8567	0.8980	0.9081	0.9086	0.9086	0.9086	0.9086	0.9081
22	0.8510	0.8957	0.9059	0.9062	0.9062	0.9062	0.9060	0.9060
23	0.8498	0.8934	0.9029	0.9045	0.9045	0.9044	0.9045	0.9045
24	0.8427	0.8907	0.9013	0.9015	0.9014	0.9014	0.9014	0.9006
25	0.8398	0.8879	0.8981	0.8989	0.8989	0.8989	0.8989	0.8989
26	0.8395	0.8868	0.8981	0.8989	0.8979	0.8979	0.8980	0.8972
27	0.8350	0.8841	0.8952	0.8960	0.8958	0.8958	0.8957	0.8957
28	0.8294	0.8828	0.8930	0.8946	0.8947	0.8938	0.8938	0.8938
29	0.8229	0.8804	0.8918	0.8922	0.8922	0.8923	0.8923	0.8923
30	0.8189	0.8783	0.8890	0.8915	0.8915	0.8915	0.8915	0.8895
31	0.8167	0.8758	0.8878	0.8885	0.8885	0.8885	0.8885	0.8885
32	0.8097	0.8740	0.8844	0.8876	0.8848	0.8848	0.8848	0.8848
33	0.8056	0.8708	0.8825	0.8842	0.8838	0.8829	0.8829	0.8829
34	0.8000	0.8683	0.8807	0.8820	0.8820	0.8813	0.8814	0.8813
35	0.8000	0.8664	0.8769	0.8798	0.8798	0.8798	0.8772	0.8772
36	0.7986	0.8630	0.8759	0.8772	0.8773	0.8771	0.8763	0.8763
37	0.7971	0.8604	0.8750	0.8762	0.8762	0.8758	0.8758	0.8758
38	0.7932	0.8572	0.8736	0.8748	0.8748	0.8748	0.8746	0.8745
39	0.7888	0.8491	0.8704	0.8732	0.8733	0.8727	0.8727	0.8727
40	0.7888	0.8491	0.8667	0.8695	0.8695	0.8695	0.8695	0.8695
41	0.7841	0.8479	0.8651	0.8695	0.8695	0.8659	0.8658	0.8658
42	0.7815	0.8396	0.8645	0.8658	0.8659	0.8659	0.8658	0.8658
43	0.7794	0.8396	0.8634	0.8642	0.8643	0.8638	0.8637	0.8637
44	0.7747	0.8396	0.8625	0.8633	0.8633	0.8629	0.8629	0.8630
45	0.7747	0.8396	0.8604	0.8623	0.8622	0.8622	0.8620	0.8620
46	0.7683	0.8362	0.8582	0.8614	0.8613	0.8613	0.8609	0.8609
47	0.7642	0.8297	0.8533	0.8597	0.8596	0.8592	0.8593	0.8583
48	0.7655	0.8297	0.8533	0.8579	0.8582	0.8578	0.8571	0.8571
49	0.7561	0.8297	0.8533	0.8568	0.8571	0.8570	0.8553	0.8553
50	0.7561	0.8211	0.8533	0.8568	0.8541	0.8541	0.8541	0.8541

SSIM	DL							
CR	1	2	3	4	5	6	7	8
51	0.7526	0.8211	0.8516	0.8540	0.8528	0.8528	0.8528	0.8527
52	0.7460	0.8211	0.8503	0.8527	0.8520	0.8519	0.8516	0.8516
53	0.7475	0.8211	0.8490	0.8516	0.8509	0.8508	0.8497	0.8497
54	0.7478	0.8202	0.8468	0.8497	0.8498	0.8497	0.8477	0.8477
55	0.7433	0.8190	0.8448	0.8476	0.8477	0.8477	0.8477	0.8448
56	0.7427	0.8162	0.8448	0.8476	0.8449	0.8448	0.8448	0.8448
57	0.7420	0.8112	0.8421	0.8447	0.8449	0.8448	0.8434	0.8434
58	0.7434	0.8112	0.8410	0.8433	0.8435	0.8433	0.8430	0.8430
59	0.7413	0.8104	0.8385	0.8430	0.8430	0.8406	0.8407	0.8406
60	0.7435	0.8090	0.8384	0.8405	0.8406	0.8403	0.8402	0.8401
61	0.7435	0.8067	0.8376	0.8401	0.8400	0.8400	0.8399	0.8396
62	0.7435	0.8058	0.8364	0.8395	0.8393	0.8389	0.8381	0.8381
63	0.7419	0.7933	0.8348	0.8380	0.8381	0.8375	0.8365	0.8365
64	0.7435	0.8090	0.8383	0.8405	0.8406	0.8403	0.8401	0.8400
65	0.7435	0.8067	0.8376	0.8401	0.8400	0.8400	0.8396	0.8393
66	0.7435	0.8058	0.8364	0.8395	0.8393	0.8389	0.8381	0.8381
67	0.7419	0.7933	0.8348	0.8380	0.8381	0.8375	0.8375	0.8365
68	0.7392	0.7933	0.8297	0.8375	0.8365	0.8365	0.8365	0.8313
69	0.7392	0.7933	0.8297	0.8364	0.8365	0.8312	0.8312	0.8313
70	0.7392	0.7933	0.8297	0.8313	0.8314	0.8312	0.8312	0.8313
71	0.7392	0.7933	0.8283	0.8313	0.8314	0.8312	0.8299	0.8299
72	0.7268	0.7933	0.8233	0.8313	0.8300	0.8299	0.8248	0.8249
73	0.7268	0.7933	0.8233	0.8249	0.8250	0.8248	0.8248	0.8249
74	0.7268	0.7933	0.8233	0.8249	0.8250	0.8248	0.8248	0.8249
75	0.7268	0.7933	0.8233	0.8249	0.8250	0.8248	0.8248	0.8249
76	0.7237	0.7923	0.8199	0.8249	0.8250	0.8248	0.8215	0.8215
77	0.7237	0.7882	0.8199	0.8249	0.8216	0.8214	0.8215	0.8215
78	0.7237	0.7882	0.8191	0.8216	0.8216	0.8214	0.8215	0.8215
79	0.7237	0.7824	0.8186	0.8216	0.8216	0.8214	0.8207	0.8207
80	0.7237	0.7824	0.8179	0.8208	0.8208	0.8207	0.8202	0.8181
81	0.7237	0.7824	0.8172	0.8208	0.8203	0.8180	0.8181	0.8181
82	0.7180	0.7824	0.8154	0.8182	0.8182	0.8180	0.8179	0.8176
83	0.7180	0.7706	0.8064	0.8182	0.8180	0.8177	0.8171	0.8169
84	0.7156	0.7706	0.8064	0.8174	0.8171	0.8169	0.8163	0.8162
85	0.7031	0.7706	0.8064	0.8167	0.8170	0.8162	0.8147	0.8147
86	0.7031	0.7706	0.8064	0.8152	0.8163	0.8147	0.8147	0.8147
87	0.7031	0.7706	0.8064	0.8135	0.8147	0.8147	0.8130	0.8130
88	0.7031	0.7706	0.8064	0.8135	0.8130	0.8130	0.8130	0.8110
89	0.6988	0.7706	0.8055	0.8115	0.8130	0.8109	0.8110	0.8110
90	0.6925	0.7706	0.8042	0.8115	0.8110	0.8109	0.8110	0.8019
91	0.6925	0.7706	0.8023	0.8024	0.8110	0.8018	0.8019	0.8019
92	0.6925	0.7684	0.8023	0.8024	0.8019	0.8018	0.8019	0.8019
93	0.6759	0.7684	0.8021	0.8024	0.8019	0.8018	0.8019	0.8019
94	0.6759	0.7663	0.8013	0.8024	0.8019	0.8018	0.8019	0.8019
95	0.6759	0.7646	0.7951	0.8024	0.8019	0.8018	0.8019	0.8019
96	0.6759	0.7646	0.7951	0.8024	0.8019	0.8018	0.8019	0.8019
97	0.6759	0.7532	0.7951	0.8024	0.8019	0.8018	0.8010	0.8010
98	0.6759	0.7532	0.7951	0.8015	0.8019	0.8010	0.8010	0.7997
99	0.6759	0.7532	0.7909	0.8002	0.8010	0.8010	0.7997	0.7997
100	0.6759	0.7532	0.7909	0.8002	0.7997	0.7997	0.7997	0.7995

A

B

The Matlab code for 1D DWT

.

```
1
2  function z = DWTwhole(x)
3  %The DWT process which take the whole image at once
4  % Cohen-Daubechies-Feauveau 9/7 lifting parameters
5  alpha = -1.586134342;
6  beta = -0.052980118;
7  gamma = 0.882911075;
8  delta = 0.443506852;
9  K1 = 1.230174104914001;
10 K2 = 1/1.230174104914001;
11 % K1 = 1/1.230174104914001;
12 % K2 = 1.230174104914001/2;
13
14 % Obtain the size of input data //default 8*8*4096
15 [R,C] = size(x); %obtain the length and width
16 y = x; % a copy of input data
17 y1 = zeros(R,C); %intermediate matrix
18 z = zeros(R,C); %create the output matrix
19
20
21 % Lifting scheme DWT
22
23     % horizontal DWT
24     % Lifting Scheme step 1 (even predict odd)
25     % right boundary handling
26     y1(:, C) = y(:, C) + 2 * alpha * y(:, C-1);
```

```
    for i = 2:2:C-1
        y1(:, i) = y(:, i) + alpha * (y(:, i-1) + y(:, i+1));
    end
    % Lifting Scheme step 2 (odd update even)
    for i = 3:2:C-1
        y1(:, i) = y(:, i) + beta * (y1(:, i-1) + y1(:, i+1));
    end
    % left boundary handling
    y1(:, 1) = y(:, 1) + 2 * beta * y1(:, 2);

    % Lifting Scheme step 3 (even predict odd)
    % right boundary handling
    y1(:, C) = y1(:, C) + 2 * gamma * y1(:, C-1);
    for i = 2:2:C-1
        y1(:, i) = y1(:, i) + gamma * (y1(:, i-1) + y1(:, i+1));
    end
    % Lifting Scheme step 4 (odd update even)
    for i = 3:2:C-1
        y1(:, i) = y1(:, i) + delta * (y1(:, i-1) + y1(:, i+1));
    end
    % left boundary handling
    y1(:, 1) = y1(:, 1) + 2 * delta * y1(:, 2);

    % Lifting Scheme step 5 (scaling)
    y1(:, 1:2:C) = y1(:, 1:2:C) * K2;
    y1(:, 2:2:C) = y1(:, 2:2:C) * K1;

% deinterleaving
if mod (C,2)==1
    for i = 1:(C-1)/2
        y(:, i) = y1(:, 2*i);
    end
    for i = 1:(C-1)/2 +1
        y(:, (C-1)/2 + i) = y1(:, 2*(i-1)+1);
    end
else
    for i = 1:C/2
        y(:, i) = y1(:,2*(i-1)+1 );
        y(:, C/2 + i) = y1(:,2*i );
    end
end
y1(:)=0;
```

```
69
70
71
72      % verticalDWT
73      % Lifting Scheme step 1 (even predict odd)
74      % bottom boundary handling
75      y1(R, :) = y(R, :) + 2 * alpha * y(R-1, :);
76      for i = 2:2:R-1
77          y1(i, :) = y(i, :) + alpha * (y(i-1, :) + y(i+1, :));
78      end
79      % Lifting Scheme step 2 (odd update even)
80      for i = 3:2:R-1
81          y1(i, :) = y(i, :) + beta * (y1(i-1, :) + y1(i+1, :));
82      end
83      % top boundary handling
84      y1(1, :) = y(1, :) + 2 * beta * y1(2, :);
85
86
87      % Lifting Scheme step 3 (even predict odd)
88      % bottom boundary handling
89      y1(R, :) = y1(R, :) + 2 * gamma * y1(R-1, :);
90      for i = 2:2:R-1
91          y1(i, :) = y1(i, :) + gamma * (y1(i-1, :) + y1(i+1, :));
92      end
93      % Lifting Scheme step 4 (odd update even)
94      for i = 3:2:R-1
95          y1(i, :) = y1(i, :) + delta * (y1(i-1, :) + y1(i+1, :));
96      end
97      % top boundary handling
98      y1(1, :) = y1(1, :) + 2 * delta * y1(2, :);
99
100     % Lifting Scheme step 5 (scaling)
101     y1(1:2:R, :) = y1(1:2:R, :) * K2;
102     y1(2:2:R, :) = y1(2:2:R, :) * K1;
103
104
105     y(:) =0;
106
107 % deinterleaveing
108 if mod (R,2)==1
109     for i = 1:(R-1)/2
110         y( i ,:) = y1( 2*i ,:);
```

```
    end
    for i = 1:(R-1)/2 +1
        y((R-1)/2 + i,:) = y1(2*(i-1)+1,:);
    end
else
    for i = 1:R/2
        y( i  ,:) = y1( 2*(i-1)+1  ,:);
        y( R/2 + i  ,:) = y1( 2*i  ,:);
    end
end



z = y;

end
```

B

C

Partial code for verification extracted from OPENJPEG

.

```
/* <summary>                                */
/* Forward 5-3 wavelet transform in 2-D. */
/* </summary>                               */
static INLINE OPJ_BOOL opj_dwt_encode_procedure(opj_thread_pool_t* tp,
        opj_tcd_tilecomp_t * tilec,
        opj_encode_and_deinterleave_v_fnptr_type p_encode_and_deinterleave_v,
        opj_encode_and_deinterleave_h_one_row_fnptr_type
        p_encode_and_deinterleave_h_one_row)
{
    OPJ_INT32 i;
    OPJ_INT32 *bj = 00;
    OPJ_UINT32 w;
    OPJ_INT32 l;

    OPJ_SIZE_T l_data_size;

    opj_tcd_resolution_t * l_cur_res = 0;
    opj_tcd_resolution_t * l_last_res = 0;
    const int num_threads = opj_thread_pool_get_thread_count(tp);
    OPJ_INT32 * OPJ_RESTRICT tiledp = tilec->data;

    w = (OPJ_UINT32)(tilec->x1 - tilec->x0);
```

```
23      l = (OPJ_INT32)tilec->numresolutions - 1;
24
25      l_cur_res = tilec->resolutions + l;
26      l_last_res = l_cur_res - 1;
27
28      l_data_size = opj_dwt_max_resolution(tilec->resolutions, tilec->numresolutions);
29      /* overflow check */
30      if (l_data_size > (SIZE_MAX / (NB_ELTS_V8 * sizeof(OPJ_INT32)))) {
31          /* FIXME event manager error callback */
32          return OPJ_FALSE;
33      }
34      l_data_size *= NB_ELTS_V8 * sizeof(OPJ_INT32);
35      bj = (OPJ_INT32*)opj_aligned_32_malloc(l_data_size);
36      /* l_data_size is equal to 0 when numresolutions == 1 but bj is not used */
37      /* in that case, so do not error out */
38      if (l_data_size != 0 && ! bj) {
39          return OPJ_FALSE;
40      }
41      i = l;
42
43
44
45
46
47      while (i--) {
48          OPJ_UINT32 j;
49          OPJ_UINT32 rw;          /* width of the resolution level computed   */
50          OPJ_UINT32 rh;          /* height of the resolution level computed  */
51          OPJ_UINT32
52          rw1;      /* width of the resolution level once lower than computed one
                ↪                                          */
53          OPJ_UINT32
54          rh1;      /* height of the resolution level once lower than computed one
                ↪                                         */
55          OPJ_INT32 cas_col;  /* 0 = non inversion on horizontal filtering 1 = inversion
                ↪ between low-pass and high-pass filtering */
56          OPJ_INT32 cas_row;  /* 0 = non inversion on vertical filtering 1 = inversion
                ↪ between low-pass and high-pass filtering   */
57          OPJ_INT32 dn, sn;
58
59          rw  = (OPJ_UINT32)(l_cur_res->x1 - l_cur_res->x0);
60          rh  = (OPJ_UINT32)(l_cur_res->y1 - l_cur_res->y0);
```

```
61          rw1 = (OPJ_UINT32)(l_last_res->x1 - l_last_res->x0);
62          rh1 = (OPJ_UINT32)(l_last_res->y1 - l_last_res->y0);
63
64          cas_row = l_cur_res->x0 & 1;
65          cas_col = l_cur_res->y0 & 1;
66
67          sn = (OPJ_INT32)rh1;
68          dn = (OPJ_INT32)(rh - rh1);
69
70
71
72          OPJ_FLOAT32* float_ptr = (OPJ_FLOAT32*)(tiledp);
73          //Print out the data before the DWT
74          FILE* fp;
75          fp = fopen("D:/Imperial_study/spring_term/Individual_Project_jpeg2000/OPENjpeg/
                ↪ data_Original.csv", "w");
76
77
78
79          /* Perform vertical pass */
80          if (num_threads <= 1 || rw < 2 * NB_ELTS_V8) {
81              for (j = 0; j + NB_ELTS_V8 - 1 < rw; j += NB_ELTS_V8) {
82                  p_encode_and_deinterleave_v(tiledp + j,
83                                              bj,
84                                              rh,
85                                              cas_col == 0,
86                                              w,
87                                              NB_ELTS_V8);
88              }
89              if (j < rw) {
90                  p_encode_and_deinterleave_v(tiledp + j,
91                                              bj,
92                                              rh,
93                                              cas_col == 0,
94                                              w,
95                                              rw - j);
96              }
97          } else {
98              OPJ_UINT32 num_jobs = (OPJ_UINT32)num_threads;
99              OPJ_UINT32 step_j;
100
101             if (rw < num_jobs) {
```

```
102                 num_jobs = rw;
103             }
104             step_j = ((rw / num_jobs) / NB_ELTS_V8) * NB_ELTS_V8;
105
106             for (j = 0; j < num_jobs; j++) {
107                 opj_dwt_encode_v_job_t* job;
108
109                 job = (opj_dwt_encode_v_job_t*) opj_malloc(sizeof(opj_dwt_encode_v_job_t
                        ↪ ));
110                 if (!job) {
111                     opj_thread_pool_wait_completion(tp, 0);
112                     opj_aligned_free(bj);
113                     return OPJ_FALSE;
114                 }
115                 job->v.mem = (OPJ_INT32*)opj_aligned_32_malloc(l_data_size);
116                 if (!job->v.mem) {
117                     opj_thread_pool_wait_completion(tp, 0);
118                     opj_free(job);
119                     opj_aligned_free(bj);
120                     return OPJ_FALSE;
121                 }
122                 job->v.dn = dn;
123                 job->v.sn = sn;
124                 job->v.cas = cas_col;
125                 job->rh = rh;
126                 job->w = w;
127                 job->tiledp = tiledp;
128                 job->min_j = j * step_j;
129                 job->max_j = (j + 1 == num_jobs) ? rw : (j + 1) * step_j;
130                 job->p_encode_and_deinterleave_v = p_encode_and_deinterleave_v;
131                 opj_thread_pool_submit_job(tp, opj_dwt_encode_v_func, job);
132             }
133             opj_thread_pool_wait_completion(tp, 0);
134         }
135
136         sn = (OPJ_INT32)rw1;
137         dn = (OPJ_INT32)(rw - rw1);
138
139
140
141         /* Perform horizontal pass */
142         if (num_threads <= 1 || rh <= 1) {
```

```
143             for (j = 0; j < rh; j++) {
144                 OPJ_INT32* OPJ_RESTRICT aj = tiledp + j * w;
145                 (*p_encode_and_deinterleave_h_one_row)(aj, bj, rw,
146                                                        cas_row == 0 ? OPJ_TRUE :
                                                            ↪ OPJ_FALSE);
147             }
148         }  else {
149             OPJ_UINT32 num_jobs = (OPJ_UINT32)num_threads;
150             OPJ_UINT32 step_j;
151
152             if (rh < num_jobs) {
153                 num_jobs = rh;
154             }
155             step_j = (rh / num_jobs);
156
157             for (j = 0; j < num_jobs; j++) {
158                 opj_dwt_encode_h_job_t* job;
159
160                 job = (opj_dwt_encode_h_job_t*) opj_malloc(sizeof(opj_dwt_encode_h_job_t
                        ↪ ));
161                 if (!job) {
162                     opj_thread_pool_wait_completion(tp, 0);
163                     opj_aligned_free(bj);
164                     return OPJ_FALSE;
165                 }
166                 job->h.mem = (OPJ_INT32*)opj_aligned_32_malloc(l_data_size);
167                 if (!job->h.mem) {
168                     opj_thread_pool_wait_completion(tp, 0);
169                     opj_free(job);
170                     opj_aligned_free(bj);
171                     return OPJ_FALSE;
172                 }
173                 job->h.dn = dn;
174                 job->h.sn = sn;
175                 job->h.cas = cas_row;
176                 job->rw = rw;
177                 job->w = w;
178                 job->tiledp = tiledp;
179                 job->min_j = j * step_j;
180                 job->max_j = (j + 1U) * step_j; /* this can overflow */
181                 if (j == (num_jobs - 1U)) {  /* this will take care of the overflow */
182                     job->max_j = rh;
```

```
                }
                job->p_function = p_encode_and_deinterleave_h_one_row;
                opj_thread_pool_submit_job(tp, opj_dwt_encode_h_func, job);
            }
            opj_thread_pool_wait_completion(tp, 0);
        }

        l_cur_res = l_last_res;

        --l_last_res;


        for (OPJ_UINT32 i = 0; i < 512; i++) {
            for (OPJ_UINT32 j = 0; j < 512; j++) {
                fprintf(fp, "%.7f, ", float_ptr[i * 512 + j]);  // print each pixels
            }
            fprintf(fp, "\n");  // end and switch line
        }
        //close the file
        fclose(fp);
    }


    opj_aligned_free(bj);
    return OPJ_TRUE;
}
```

D

Verilog code for tiling module

.

```
module Tiling_tb;

logic clk;
logic nrst;
logic [31:0] din;
logic din_valid;
logic [31:0] dout;
logic dout_valid;

Tiling inst1(.*);

// clock and reset
initial begin
    clk = '0;
    forever #5ns clk = ~clk;
end

initial begin
    nrst = '0;
    #20ns
    @(negedge clk)
    nrst <= '1;
```

```
23  end
24
25  //input and valid
26  logic [7:0] din_f2f;
27  logic din_f2f_valid;
28  always_ff @ (posedge clk, negedge nrst)
29      if(~nrst) begin
30          din_f2f_valid <= 0;
31          din_f2f    <= 8'd5;
32      end
33      else begin
34          din_f2f_valid <= 1;
35          if(din_f2f_valid)
36              din_f2f <= din_f2f + 1'd1;
37      end
38
39  //fix to float transition
40  Fix2float f2f_inst (
41    .aclk(clk),                                    // input wire aclk
42    .aresetn(nrst),                               // input wire aresetn
43    .s_axis_a_tvalid(din_f2f_valid),              // input wire s_axis_a_tvalid
44    .s_axis_a_tdata(din_f2f),                 // input wire [7 : 0] s_axis_a_tdata
45    .m_axis_result_tvalid(din_valid),  // output wire m_axis_result_tvalid
46    .m_axis_result_tdata(din)    // output wire [31 : 0] m_axis_result_tdata
47  );
48
49  endmodule
```

E

Verilog code for 1D DWT

The 1D DWT verilog code

```
module DWT_1D(// do the whole DWT calculation
    input clk,
    input nrst,
    input [31:0] din,
    input din_valid,
    output [31:0] dout,
    output dout_valid
);
//intermediate wire
wire [31:0] dout_int;
wire dout_int_valid;
DWT_1D_1 dwt_inst1(
    .clk(clk),
    .nrst(nrst),
    .din(din),
    .din_valid(din_valid),
    .dout(dout_int),
    .dout_valid(dout_int_valid)
);
DWT_1D_2 dwt_inst2(
    .clk(clk),
    .nrst(nrst),
```

```
23      .din(dout_int),
24      .din_valid(dout_int_valid),
25      .dout(dout),
26      .dout_valid(dout_valid)
27  );
28  endmodule
29
30  module DWT_1D_1(//do the prediction and update once with parameter alpha and
        beta
31      input clk,
32      input nrst,
33      input [31:0] din,
34      input din_valid,
35      output wire [31:0] dout,
36      output wire dout_valid
37      );
38  //shift register counter
39  reg [8:0] din_cnt;
40  always @(posedge clk or negedge nrst) begin
41      if (~nrst)
42          din_cnt <= 9'd511;
43      else if (din_valid)
44          din_cnt <= din_cnt +1;
45  end
46  //shift register
47  wire [31:0] dout_odd_0;
48  wire [31:0] dout_odd_1;
49  wire [31:0] dout_odd_2;
50  wire odd_valid;
51  wire [31:0] dout_even;
52  wire even_valid;
53  Shift_register SR_inst (
54      .clk(clk),
55      .nrst(nrst),
56      .din(din),
57      .din_valid(din_valid),
58      .dout_odd_0(dout_odd_0),
```

```
    .dout_odd_1(dout_odd_1),
    .dout_odd_2(dout_odd_2),
    .odd_valid(odd_valid),
    .dout_even(dout_even),
    .even_valid(even_valid)
);
//buffer even output
reg [31:0] dout_even_delay [12:0];
reg [12:0] dout_even_valid_delay;
integer k;
always @(posedge clk or negedge nrst) begin
    if (~nrst) begin
        for (k = 0; k <13;k=k+1) begin
            dout_even_delay[k] <= 0;
            dout_even_valid_delay <= 0;
        end
    end
    else begin
        dout_even_delay[0] <= dout_even;
        dout_even_valid_delay[0] <= even_valid;
        for (k = 1; k <13;k=k+1) begin
            dout_even_delay[k] <= dout_even_delay[k-1];
            dout_even_valid_delay[k] <= dout_even_valid_delay[k-1];
        end
    end
end
//ALU_odd control
wire [31:0] dout_odd_new;
wire dout_odd_new_valid;
ALU_odd_1 alu_inst1(
    .clk(clk),
    .nrst(nrst),
    .din_0((din_cnt == 9'd0)? dout_odd_2 :dout_odd_0),
    .din_1(dout_odd_1),
    .din_2(dout_odd_2),
    .din_valid(odd_valid),
```

```
95      .dout_alu(dout_odd_new),
96      .dout_alu_valid(dout_odd_new_valid)
97  );
98  //ALU_even control
99  wire [31:0] dout_even_new;
100 wire dout_even_new_valid;
101 ALU_even_1 alu_inst2(
102     .clk(clk),
103     .nrst(nrst),
104     .din_0((odd_cnt == 1) ? dout_odd_new: delay0),
105     .din_1(dout_even_delay[12]),
106     .din_2(dout_odd_new),
107     .din_valid(dout_even_valid_delay[12] & dout_odd_new_valid),
108     .dout_alu(dout_even_new),
109     .dout_alu_valid(dout_even_new_valid)
110 );
111 //odd and even counter
112 reg [8:0] odd_cnt;
113 reg [8:0] even_cnt;
114 always @(posedge clk or negedge nrst) begin
115     if (~nrst) begin
116         odd_cnt <=1;
117         even_cnt <=0;
118     end
119     else begin
120         odd_cnt <= dout_odd_new_valid ? odd_cnt +2:odd_cnt;
121         even_cnt <= dout_even_new_valid ? even_cnt + 2: even_cnt;
122     end
123 end
124 //odd output shifted by 2 to feed even alu
125 reg [31:0] delay0;
126 always @(posedge clk or negedge nrst) begin
127     if (~nrst)
128         delay0 <= 0;
129     else if(dout_odd_new_valid)
130         delay0 <= dout_odd_new;
131 end
```

```
132 //otuput control
133 //odd output shifted by 13 to form output sequence
134 reg [31:0] delay_output [12:0];
135 reg [12:0] delay_output_valid;
136 integer i;
137 integer j;
138 always @(posedge clk or negedge nrst) begin
139     if (~nrst) begin
140         for (i = 0; i<13;i= i+1) begin
141             delay_output[i] <=0;
142             delay_output_valid[i] <=0;
143         end
144     end
145     else begin
146         delay_output[0] <= dout_odd_new;
147         delay_output_valid[0] <= dout_odd_new_valid;
148         for (j = 12; j>0 ; j=j-1) begin
149             delay_output[j] <= delay_output[j-1];
150             delay_output_valid[j] <= delay_output_valid[j-1];
151         end
152     end
153 end
154 assign dout = (dout_cnt[0] == 0)? dout_even_new: delay_output[12];
155 //dout_valid
156 assign dout_valid = dout_even_new_valid | delay_output_valid[12];
157 //output counter
158 reg [8:0] dout_cnt;
159 always @(posedge clk or negedge nrst) begin
160     if(~nrst)
161         dout_cnt <= 0;
162     else
163         dout_cnt <= dout_valid ? dout_cnt +1:dout_cnt;
164 end
165 endmodule
166
167 module DWT_1D_2(//do prediction and update second time with parameter gama
        delta and do the scale with para k.
```

```
168     input clk,
169     input nrst,
170     input [31:0] din,
171     input din_valid,
172     output wire [31:0] dout,
173     output wire dout_valid
174     );
175 //shift register counter
176 reg [8:0] din_cnt;
177 always @(posedge clk or negedge nrst) begin
178     if (~nrst)
179         din_cnt <= 9'd511;
180     else if (din_valid)
181         din_cnt <= din_cnt +1;
182 end
183 //shift register
184 wire [31:0] dout_odd_0;
185 wire [31:0] dout_odd_1;
186 wire [31:0] dout_odd_2;
187 wire odd_valid;
188 wire [31:0] dout_even;
189 wire even_valid;
190 Shift_register SR_inst (
191     .clk(clk),
192     .nrst(nrst),
193     .din(din),
194     .din_valid(din_valid),
195     .dout_odd_0(dout_odd_0),
196     .dout_odd_1(dout_odd_1),
197     .dout_odd_2(dout_odd_2),
198     .odd_valid(odd_valid),
199     .dout_even(dout_even),
200     .even_valid(even_valid)
201 );
202 //buffer even output
203 reg [31:0] dout_even_delay [12:0];
204 reg [12:0] dout_even_valid_delay;
```

```
integer k;
always @(posedge clk or negedge nrst) begin
    if (~nrst) begin
        for (k = 0; k <13;k=k+1)
            dout_even_delay[k] <= 0;
        dout_even_valid_delay <= 0;
    end
    else begin
        dout_even_delay[0] <= dout_even;
        dout_even_valid_delay[0] <= even_valid;
        for (k = 1; k <13;k=k+1) begin
            dout_even_delay[k] <= dout_even_delay[k-1];
            dout_even_valid_delay[k] <= dout_even_valid_delay[k-1];
        end
    end
end
//ALU_odd control
wire [31:0] dout_odd_new;
wire dout_odd_new_valid;
ALU_odd_2 alu_inst1(
    .clk(clk),
    .nrst(nrst),
    .din_0((din_cnt == 9'd0)? dout_odd_2 :dout_odd_0),
    .din_1(dout_odd_1),
    .din_2(dout_odd_2),
    .din_valid(odd_valid),
    .dout_alu(dout_odd_new),
    .dout_alu_valid(dout_odd_new_valid)
);
//ALU_even control
wire [31:0] dout_even_new;
wire dout_even_new_valid;
ALU_even_2 alu_inst2(
    .clk(clk),
    .nrst(nrst),
    .din_0((odd_cnt == 1) ? dout_odd_new: delay0),
    .din_1(dout_even_delay[12]),
```

```
242     .din_2(dout_odd_new),
243     .din_valid(dout_even_valid_delay[12] & dout_odd_new_valid),
244     .dout_alu(dout_even_new),
245     .dout_alu_valid(dout_even_new_valid)
246 );
247 //odd and even counter
248 reg [8:0] odd_cnt;
249 reg [8:0] even_cnt;
250 always @(posedge clk or negedge nrst) begin
251     if (~nrst) begin
252         odd_cnt <=1;
253         even_cnt <=0;
254     end
255     else begin
256         odd_cnt <= dout_odd_new_valid ? odd_cnt +2:odd_cnt;
257         even_cnt <= dout_even_new_valid ? even_cnt + 2: even_cnt;
258     end
259 end
260 //odd output shifted by 2 to feed even alu
261 reg [31:0] delay0;
262 always @(posedge clk or negedge nrst) begin
263     if (~nrst)
264         delay0 <= 0;
265     else if(dout_odd_new_valid)
266         delay0 <= dout_odd_new;
267 end
268 //otuput control
269 //odd output shifted by 13 to form output sequence
270 reg [31:0] delay_output [12:0];
271 reg [12:0] delay_output_valid;
272 integer i;
273 integer j;
274 always @(posedge clk or negedge nrst) begin
275     if (~nrst) begin
276         for (i = 0; i<13;i= i+1) begin
277             delay_output[i] <=0;
278             delay_output_valid[i] <=0;
```

```
279         end
280     end
281     else begin
282         delay_output[0] <= dout_odd_new;
283         delay_output_valid[0] <= dout_odd_new_valid;
284         for (j = 12; j>0 ; j=j-1) begin
285             delay_output[j] <= delay_output[j-1];
286             delay_output_valid[j] <= delay_output_valid[j-1];
287         end
288     end
289 end
290 
291 assign dout = (dout_cnt[0] == 0)? dout_even_last: dout_odd_last;
292 //dout_valid
293 assign dout_valid = dout_even_last_valid | dout_odd_last_valid;
294 //output counter
295 reg [8:0] dout_cnt;
296 always @(posedge clk or negedge nrst) begin
297     if(~nrst)
298         dout_cnt <= 0;
299     else
300         dout_cnt <= dout_valid ? dout_cnt +1:dout_cnt;
301 end
302 //multiplier to scale the output data with parameter k and 1/k and the output
        counter
303 wire [31:0] dout_odd_last;
304 wire dout_odd_last_valid;
305 wire [31:0] dout_even_last;
306 wire dout_even_last_valid;
307 floating_multiply multiply_scale_even (
308   .aclk(clk),                                    // input wire aclk
309   .aresetn(nrst),                               // input wire aresetn
310   .s_axis_a_tvalid(dout_even_new_valid),             // input wire
          s_axis_a_tvalid
311   .s_axis_a_tdata(dout_even_new),                // input wire [31 : 0]
          s_axis_a_tdata
```

```
312   .s_axis_b_tvalid(dout_even_new_valid),            // input wire
          s_axis_b_tvalid
313   .s_axis_b_tdata(32'b00111111010100000001100111000011),              // input
          wire [31 : 0] s_axis_b_tdata 1/k
314   .m_axis_result_tvalid(dout_even_last_valid),  // output wire
          m_axis_result_tvalid
315   .m_axis_result_tdata(dout_even_last)    // output wire [31 : 0]
          m_axis_result_tdata
316 );
317 floating_multiply multiply_scale_odd (
318   .aclk(clk),                                    // input wire aclk
319   .aresetn(nrst),                               // input wire aresetn
320   .s_axis_a_tvalid(delay_output_valid[12]),            // input wire
          s_axis_a_tvalid
321   .s_axis_a_tdata(delay_output[12]),              // input wire [31 : 0]
          s_axis_a_tdata
322   .s_axis_b_tvalid(delay_output_valid[12]),            // input wire
          s_axis_b_tvalid
323   .s_axis_b_tdata(32'b00111111100111010111011001011000),              // input
          wire [31 : 0] s_axis_b_tdata k
324   .m_axis_result_tvalid(dout_odd_last_valid),  // output wire
          m_axis_result_tvalid
325   .m_axis_result_tdata(dout_odd_last)    // output wire [31 : 0]
          m_axis_result_tdata
326 );
327 endmodule
328 //The shift register verilog code:
329 module Shift_register #(parameter depth = 6, buffer_size = 3)(
330     input                   clk,
331     input                   nrst,
332     input [31:0]            din,
333     input                   din_valid,
334     output wire [31:0]      dout_odd_0,
335     output wire [31:0]      dout_odd_1,
336     output wire [31:0]      dout_odd_2,
337     output wire              odd_valid,
338     output wire [31:0]      dout_even,
```

```
339     output wire                 even_valid
340     );
341 //shift register define and input/output counter
342 reg [31:0] reg_int[buffer_size-1:0];
343 reg [8:0] din_cnt;
344 reg [8:0] dout_cnt;
345 reg dout_ready;
346 
347 reg dout_cnt_ready;
348 always @(posedge clk or negedge nrst) begin
349     if (~nrst)
350         din_cnt <=0;
351     else if (din_valid)
352         din_cnt <= din_cnt+1'd1;
353 end
354 always @(posedge clk or negedge nrst) begin
355     if (~nrst) begin
356         dout_cnt <=0;
357         dout_ready<=0;
358         dout_cnt_ready<=0;
359     end else if (din_valid) begin
360         if ((din_cnt == 9'd2) & !dout_cnt_ready) begin
361             dout_cnt <= dout_cnt + 1'd1;
362             dout_cnt_ready <=1;
363         end
364 
365         if (dout_cnt_ready)
366             dout_cnt <=dout_cnt+1'd1;
367 
368         if (din_cnt == 9'd1)
369             dout_ready <=1;
370     end
371 end
372 //reg dout_ready_new;
373 //always @(posedge clk or negedge nrst) begin
374 //    if (~nrst)
375 //        dout_ready_new <=0;
```

```
376 //    else begin
377 //        dout_ready_new <= dout_ready & din_valid;
378 //    end
379 //end
380
381 wire dout_ready_new;
382 assign dout_ready_new = din_valid & dout_ready;
383
384 //shift register control
385 integer i;
386 integer j;
387 integer k;
388 integer l;
389 always @(posedge clk or negedge nrst) begin
390     if (~nrst) begin
391         for (i = 0; i < buffer_size; i = i + 1)
392             reg_int[i] <= 0;
393     end
394     else if(din_valid) begin
395         reg_int[0] <= din;
396         for (k = buffer_size-1; k > 0; k = k - 1)
397             reg_int[k] <= reg_int[k-1];
398     end
399 end
400 //output control
401 assign      dout_odd_0 = reg_int[0];
402 assign      dout_odd_1 = reg_int[1];
403 assign      dout_odd_2 = reg_int[2];
404 assign      dout_even  = reg_int[1];
405 //output valid control
406 assign even_valid = dout_ready_new & ~dout_cnt[0];
407 assign odd_valid = dout_ready_new & dout_cnt[0];
408 endmodule
```

F

Verilog code for 1D DWT


```
//The odd ALU code:

module ALU_odd_1(
    input           clk,
    input           nrst,
    input [31:0]    din_0,
    input [31:0]    din_1,
    input [31:0]    din_2,
    input           din_valid,
    output [31:0]   dout_alu,
    output          dout_alu_valid
    );
//adder one control
wire dout_adder1_valid;
wire [31:0] dout_adder1;
floating_add add_inst1 (
  .aclk(clk),                                  // input wire aclk
  .aresetn(nrst),                            // input wire aresetn
  .s_axis_a_tvalid(din_valid),            // input wire s_axis_a_tvalid
  .s_axis_a_tdata(din_0),              // input wire [31 : 0] s_axis_a_tdata
  .s_axis_b_tvalid(din_valid),            // input wire s_axis_b_tvalid
  .s_axis_b_tdata(din_2),              // input wire [31 : 0] s_axis_b_tdata
  .m_axis_result_tvalid(dout_adder1_valid),  // output wire
```

```
      m_axis_result_tvalid
  .m_axis_result_tdata(dout_adder1)    // output wire [31 : 0]
      m_axis_result_tdata
);
//multiplier one control
reg [31:0] alpha = 32'b10111111110010110000011001110011;//alpha parameter
wire [31:0] din_alpha;
assign din_alpha = alpha;
wire dout_mul_valid;
wire [31:0] dout_mul;
floating_multiply multiply_inst1 (
  .aclk(clk),                                  // input wire aclk
  .aresetn(nrst),                             // input wire aresetn
  .s_axis_a_tvalid(dout_adder1_valid),            // input wire s_axis_a_tvalid
  .s_axis_a_tdata(dout_adder1),               // input wire [31 : 0]
      s_axis_a_tdata
  .s_axis_b_tvalid(dout_adder1_valid),            // input wire s_axis_b_tvalid
  .s_axis_b_tdata(din_alpha),              // input wire [31 : 0]
      s_axis_b_tdata
  .m_axis_result_tvalid(dout_mul_valid),  // output wire m_axis_result_tvalid
  .m_axis_result_tdata(dout_mul)    // output wire [31 : 0] m_axis_result_tdata
);
//delay middle data module control
wire dout_NB8_valid;
wire [31:0] dout_NB8;
 nonblocking8 NB_inst8(
  .clk(clk),
  .nrst(nrst),
  .din(din_1),
  .din_valid(din_valid),
  .dout(dout_NB8),
  .dout_valid(dout_NB8_valid)
);
//second adder control
wire dout_adder2_valid;
wire [31:0] dout_adder2;
floating_add add_inst2 (
```

```
57    .aclk(clk),                                    // input wire aclk
58    .aresetn(nrst),                               // input wire aresetn
59    .s_axis_a_tvalid(dout_mul_valid),              // input wire s_axis_a_tvalid
60    .s_axis_a_tdata(dout_mul),                // input wire [31 : 0] s_axis_a_tdata
61    .s_axis_b_tvalid(dout_NB8_valid),              // input wire s_axis_b_tvalid
62    .s_axis_b_tdata(dout_NB8),                // input wire [31 : 0] s_axis_b_tdata
63    .m_axis_result_tvalid(dout_adder2_valid),  // output wire
          m_axis_result_tvalid
64    .m_axis_result_tdata(dout_adder2)    // output wire [31 : 0]
          m_axis_result_tdata
65  );
66  assign dout_alu = dout_adder2;
67  assign dout_alu_valid = dout_adder2_valid;
68
69  endmodule
70
71
72
73  module ALU_odd_2(
74      input            clk,
75      input            nrst,
76      input [31:0]     din_0,
77      input [31:0]     din_1,
78      input [31:0]     din_2,
79      input            din_valid,
80      output [31:0]    dout_alu,
81      output           dout_alu_valid
82      );
83  //adder one control
84  wire dout_adder1_valid;
85  wire [31:0] dout_adder1;
86  floating_add add_inst1 (
87    .aclk(clk),                                    // input wire aclk
88    .aresetn(nrst),                               // input wire aresetn
89    .s_axis_a_tvalid(din_valid),              // input wire s_axis_a_tvalid
90    .s_axis_a_tdata(din_0),                // input wire [31 : 0] s_axis_a_tdata
```

F

```
91    .s_axis_b_tvalid(din_valid),            // input wire s_axis_b_tvalid
92    .s_axis_b_tdata(din_2),               // input wire [31 : 0] s_axis_b_tdata
93    .m_axis_result_tvalid(dout_adder1_valid),  // output wire
          m_axis_result_tvalid
94    .m_axis_result_tdata(dout_adder1)    // output wire [31 : 0]
          m_axis_result_tdata
95  );
96  //multiplier one control
97  reg [31:0] gama = 32'b00111111011000100000011001110110;//gama parameter
98  wire [31:0] din_gama;
99  assign din_gama = gama;
100 wire dout_mul_valid;
101 wire [31:0] dout_mul;
102 floating_multiply multiply_inst1 (
103   .aclk(clk),                                  // input wire aclk
104   .aresetn(nrst),                             // input wire aresetn
105   .s_axis_a_tvalid(dout_adder1_valid),            // input wire s_axis_a_tvalid
106   .s_axis_a_tdata(dout_adder1),               // input wire [31 : 0]
          s_axis_a_tdata
107   .s_axis_b_tvalid(dout_adder1_valid),            // input wire s_axis_b_tvalid
108   .s_axis_b_tdata(din_gama),              // input wire [31 : 0] s_axis_b_tdata
109   .m_axis_result_tvalid(dout_mul_valid),  // output wire m_axis_result_tvalid
110   .m_axis_result_tdata(dout_mul)    // output wire [31 : 0] m_axis_result_tdata
111 );
112 //delay middle data module control
113 wire dout_NB8_valid;
114 wire [31:0] dout_NB8;
115  nonblocking8 NB_inst8(
116   .clk(clk),
117   .nrst(nrst),
118   .din(din_1),
119   .din_valid(din_valid),
120   .dout(dout_NB8),
121   .dout_valid(dout_NB8_valid)
122 );
123 //second adder control
124 wire dout_adder2_valid;
```

```
125 wire [31:0] dout_adder2;
126 floating_add add_inst2 (
127   .aclk(clk),                                    // input wire aclk
128   .aresetn(nrst),                               // input wire aresetn
129   .s_axis_a_tvalid(dout_mul_valid),              // input wire s_axis_a_tvalid
130   .s_axis_a_tdata(dout_mul),                // input wire [31 : 0] s_axis_a_tdata
131   .s_axis_b_tvalid(dout_NB8_valid),              // input wire s_axis_b_tvalid
132   .s_axis_b_tdata(dout_NB8),                // input wire [31 : 0] s_axis_b_tdata
133   .m_axis_result_tvalid(dout_adder2_valid),  // output wire
          m_axis_result_tvalid
134   .m_axis_result_tdata(dout_adder2)    // output wire [31 : 0]
          m_axis_result_tdata
135 );
136 assign dout_alu = dout_adder2;
137 assign dout_alu_valid = dout_adder2_valid;
138
139 endmodule
140
141
142 //The even ALU code:
143
144
145 module ALU_even_1(
146     input            clk,
147     input            nrst,
148     input [31:0]     din_0,
149     input [31:0]     din_1,
150     input [31:0]     din_2,
151     input            din_valid,
152     output [31:0]    dout_alu,
153     output           dout_alu_valid
154     );
155
156 //adder one control
157 wire dout_adder1_valid;
158 wire [31:0] dout_adder1;
159 floating_add add_inst1 (
```

F

```
160   .aclk(clk),                                    // input wire aclk
161   .aresetn(nrst),                               // input wire aresetn
162   .s_axis_a_tvalid(din_valid),                  // input wire s_axis_a_tvalid
163   .s_axis_a_tdata(din_0),                     // input wire [31 : 0] s_axis_a_tdata
164   .s_axis_b_tvalid(din_valid),                  // input wire s_axis_b_tvalid
165   .s_axis_b_tdata(din_2),                     // input wire [31 : 0] s_axis_b_tdata
166   .m_axis_result_tvalid(dout_adder1_valid),  // output wire
          m_axis_result_tvalid
167   .m_axis_result_tdata(dout_adder1)    // output wire [31 : 0]
          m_axis_result_tdata
168 );
169
170
171 //multiplier one control
172 reg [31:0] beta = 32'b10111101010110010000000110101110;//beta parameter
173 wire [31:0] din_beta;
174 assign din_beta = beta;
175 wire dout_mul_valid;
176 wire [31:0] dout_mul;
177 floating_multiply multiply_inst1 (
178   .aclk(clk),                                      // input wire aclk
179   .aresetn(nrst),                                 // input wire aresetn
180   .s_axis_a_tvalid(dout_adder1_valid),              // input wire s_axis_a_tvalid
181   .s_axis_a_tdata(dout_adder1),                 // input wire [31 : 0]
          s_axis_a_tdata
182   .s_axis_b_tvalid(dout_adder1_valid),              // input wire s_axis_b_tvalid
183   .s_axis_b_tdata(din_beta),                  // input wire [31 : 0] s_axis_b_tdata
184   .m_axis_result_tvalid(dout_mul_valid),  // output wire m_axis_result_tvalid
185   .m_axis_result_tdata(dout_mul)    // output wire [31 : 0] m_axis_result_tdata
186 );
187
188 //delay middle data module control
189 wire dout_NB8_valid;
190 wire [31:0] dout_NB8;
191  nonblocking8 NB_inst8(
192   .clk(clk),
193   .nrst(nrst),
```

```
194   .din(din_1),
195   .din_valid(din_valid),
196   .dout(dout_NB8),
197   .dout_valid(dout_NB8_valid)
198 );
199
200 //second adder control
201 wire dout_adder2_valid;
202 wire [31:0] dout_adder2;
203 floating_add add_inst2 (
204   .aclk(clk),                                    // input wire aclk
205   .aresetn(nrst),                               // input wire aresetn
206   .s_axis_a_tvalid(dout_mul_valid),             // input wire s_axis_a_tvalid
207   .s_axis_a_tdata(dout_mul),               // input wire [31 : 0] s_axis_a_tdata
208   .s_axis_b_tvalid(dout_NB8_valid),             // input wire s_axis_b_tvalid
209   .s_axis_b_tdata(dout_NB8),               // input wire [31 : 0] s_axis_b_tdata
210   .m_axis_result_tvalid(dout_adder2_valid),  // output wire
          m_axis_result_tvalid
211   .m_axis_result_tdata(dout_adder2)    // output wire [31 : 0]
          m_axis_result_tdata
212 );
213
214 assign dout_alu = dout_adder2;
215 assign dout_alu_valid = dout_adder2_valid;
216
217 endmodule
218
219
220
221
222
223
224 module ALU_even_2(
225     input            clk,
226     input            nrst,
227     input [31:0]     din_0,
228     input [31:0]     din_1,
```

```
229     input [31:0]    din_2,
230     input           din_valid,
231     output [31:0]   dout_alu,
232     output          dout_alu_valid
233     );
234
235 //adder one control
236 wire dout_adder1_valid;
237 wire [31:0] dout_adder1;
238 floating_add add_inst1 (
239   .aclk(clk),                                  // input wire aclk
240   .aresetn(nrst),                              // input wire aresetn
241   .s_axis_a_tvalid(din_valid),            // input wire s_axis_a_tvalid
242   .s_axis_a_tdata(din_0),               // input wire [31 : 0] s_axis_a_tdata
243   .s_axis_b_tvalid(din_valid),            // input wire s_axis_b_tvalid
244   .s_axis_b_tdata(din_2),               // input wire [31 : 0] s_axis_b_tdata
245   .m_axis_result_tvalid(dout_adder1_valid),  // output wire
          m_axis_result_tvalid
246   .m_axis_result_tdata(dout_adder1)    // output wire [31 : 0]
          m_axis_result_tdata
247 );
248
249
250 //multiplier one control
251 reg [31:0] delta = 32'b00111110111000110001001101010101;//delta parameter
252 wire [31:0] din_delta;
253 assign din_delta = delta;
254 wire dout_mul_valid;
255 wire [31:0] dout_mul;
256 floating_multiply multiply_inst1 (
257   .aclk(clk),                                  // input wire aclk
258   .aresetn(nrst),                              // input wire aresetn
259   .s_axis_a_tvalid(dout_adder1_valid),            // input wire s_axis_a_tvalid
260   .s_axis_a_tdata(dout_adder1),               // input wire [31 : 0]
          s_axis_a_tdata
261   .s_axis_b_tvalid(dout_adder1_valid),            // input wire s_axis_b_tvalid
262   .s_axis_b_tdata(din_delta),              // input wire [31 : 0]
```

```
        s_axis_b_tdata
263   .m_axis_result_tvalid(dout_mul_valid),  // output wire m_axis_result_tvalid
264   .m_axis_result_tdata(dout_mul)    // output wire [31 : 0] m_axis_result_tdata
265 );
266
267 //delay middle data module control
268 wire dout_NB8_valid;
269 wire [31:0] dout_NB8;
270  nonblocking8 NB_inst8(
271   .clk(clk),
272   .nrst(nrst),
273   .din(din_1),
274   .din_valid(din_valid),
275   .dout(dout_NB8),
276   .dout_valid(dout_NB8_valid)
277 );
278
279 //second adder control
280 wire dout_adder2_valid;
281 wire [31:0] dout_adder2;
282 floating_add add_inst2 (
283   .aclk(clk),                                  // input wire aclk
284   .aresetn(nrst),                             // input wire aresetn
285   .s_axis_a_tvalid(dout_mul_valid),            // input wire s_axis_a_tvalid
286   .s_axis_a_tdata(dout_mul),              // input wire [31 : 0] s_axis_a_tdata
287   .s_axis_b_tvalid(dout_NB8_valid),            // input wire s_axis_b_tvalid
288   .s_axis_b_tdata(dout_NB8),              // input wire [31 : 0] s_axis_b_tdata
289   .m_axis_result_tvalid(dout_adder2_valid),  // output wire
        m_axis_result_tvalid
290   .m_axis_result_tdata(dout_adder2)    // output wire [31 : 0]
        m_axis_result_tdata
291 );
292
293 assign dout_alu = dout_adder2;
294 assign dout_alu_valid = dout_adder2_valid;
295
296 endmodule
```

```
297
298
299 //The delay module verilog code:
300
301
302
303 module nonblocking8
304 (
305   input wire       clk    ,
306   input wire       nrst   ,
307   input wire  [31:0]  din      ,
308   input   wire            din_valid,
309   output  reg   [31:0]  dout     ,
310   output  reg             dout_valid
311 );
312
313 //Din_counter
314 reg [2:0] din_cnt;
315 reg dout_ready;
316 //delay register
317 reg [31:0] delay [6:0];
318
319 //din counter
320 always @(posedge clk, negedge nrst) begin
321     if (~nrst)
322         din_cnt <= 0;
323     else if (din_valid && !dout_ready) begin
324         din_cnt <= din_cnt + 1'd1;
325         dout_ready <= 1;
326     end else if (dout_ready)
327         din_cnt <= din_cnt +1;
328 end
329
330 //dout_valid control
331 reg [6:0] din_valid_shift;
332 always @(posedge clk, negedge nrst) begin
333     if(~nrst) begin
```

```
334         din_valid_shift <= 8'd0;
335         dout_valid <= 0;
336     end else begin
337         din_valid_shift <= {din_valid_shift[5:0],din_valid};
338         dout_valid <= din_valid_shift[6];
339     end
340 end
341 
342 //out control
343 integer i;
344 always@(posedge clk or negedge nrst) begin
345   if(!nrst) begin
346      for (i = 0; i < 7; i = i + 1) begin
347          delay[i] <= 32'd0;
348      end
349      dout <= 32'd0;
350     end else begin
351      delay[0] <= din; //
352        for (i = 0; i < 7; i = i + 1) begin
353            delay[i+1] <= delay[i]; //
354        end
355        dout <= delay[6]; //
356     end
357 end
358 endmodule
359 
360 
361 
362 module nonblocking12
363 (
364   input wire       clk    ,
365   input wire       nrst   ,
366   input wire  [31:0]  din      ,
367   input   wire            din_valid,
368   output  reg   [31:0]  dout     ,
369   output  reg             dout_valid
370 );
```

```
371
372 //Din_counter
373 reg [3:0] din_cnt;
374
375 //delay register
376 reg [31:0] delay0;
377 reg [31:0] delay1;
378 reg [31:0] delay2;
379 reg [31:0] delay3;
380 reg [31:0] delay4;
381 reg [31:0] delay5;
382 reg [31:0] delay6;
383 reg [31:0] delay7;
384 reg [31:0] delay8;
385 reg [31:0] delay9;
386 reg [31:0] delay10;
387
388
389 //din counter
390 always @(posedge clk, negedge nrst) begin
391     if (~nrst)
392         din_cnt <= 0;
393     else
394         if (din_valid)
395             din_cnt <= din_cnt + 1'd1;
396 end
397
398 //dout_valid control
399 always @(posedge clk, negedge nrst) begin
400     if(~nrst)
401         dout_valid <= 0;
402     else if(din_valid && din_cnt == 9'd11)
403         dout_valid <= 1;
404 end
405
406 //out control
407 always@(posedge clk or negedge nrst) begin
```

```
408   if(!nrst)
409     begin
410             delay0 <= 0;
411             delay1 <= 0;
412             delay2 <= 0;
413             delay3 <= 0;
414             delay4 <= 0;
415             delay5 <= 0;
416             delay6 <= 0;
417             delay7 <= 0;
418             delay8 <= 0;
419             delay9 <= 0;
420             delay10 <= 0;
421             dout <= 0;
422     end
423   else if (din_valid)begin
424             delay0 <= din;
425             delay1 <= delay0;
426             delay2 <= delay1;
427             delay3 <= delay2;
428             delay4 <= delay3;
429             delay5 <= delay4;
430             delay6 <= delay5;
431             delay7 <= delay6;
432             delay8 <= delay7;
433             delay9 <= delay8;
434             delay10 <= delay9;
435             dout <= delay10;
436     end
437 end
438 endmodule
```

F

G

Verilog code for Ping Pang Buffer

```
1
2  module pp_buf
3  (
4      input  logic        clk,
5      input  logic        nrst,
6      input  logic [31:0] din,
7      input  logic        din_valid,
8      output logic [31:0] dout,
9      output logic        dout_valid
10 );
11
12 // Din Counter
13 logic [17:0] din_cnt;//ipnut counter
14 logic [17:0] dout_cnt_control;//output control counter
15 logic [17:0] dout_cnt;// output counter
16 logic [8:0] din_col;// counter for control and debug
17 logic [8:0] din_row;
18 assign dout_cnt = {din_cnt[8:0],din_cnt[17:9]};
19
20 always_ff @ (posedge clk, negedge nrst)
21     if(~nrst) begin
22         din_col <= '0;
23         din_row <= '0;
```

```
24          din_cnt <= '0;
25          dout_cnt_control <= '0;
26      end
27      else if(din_valid) begin
28              din_col <= din_col + 1'd1;
29              din_row <= (din_col == 9'd511) ? din_row + 1'd1 : din_row;
30              din_cnt <= din_cnt + 1;
31              dout_cnt_control <= din_cnt;
32      end
33
34  // RAM
35  logic wea;//wirte enable a
36  logic web;// write enable b
37  logic [31:0] douta;//output from ram0
38  logic [31:0] doutb;//output from ram1
39  logic rsta_busy0;
40  logic rsta_busy1;
41
42  blk_mem_gen_0 BM_0 (
43    .clka(clk),             // input wire clka
44    .rsta(~nrst),             // input wire rsta
45    .wea(wea),               // input wire [0 : 0] wea
46    .addra((wea)?din_cnt:dout_cnt),          // input wire [17 : 0] addra
47    .dina(din),             // input wire [31 : 0] dina
48    .douta(douta),           // output wire [31 : 0] douta
49    .rsta_busy(rsta_busy0)  // output wire rsta_busy
50  );
51
52  blk_mem_gen_0 BM_1 (
53    .clka(clk),             // input wire clka
54    .rsta(~nrst),             // input wire rsta
55    .wea(web),               // input wire [0 : 0] wea
56    .addra((web)?din_cnt:dout_cnt),          // input wire [17 : 0] addra
57    .dina(din),             // input wire [31 : 0] dina
58    .douta(doutb),           // output wire [31 : 0] douta
59    .rsta_busy(rsta_busy1)  // output wire rsta_busy
60  );
```

```
61
62  //write enable Control.  write ram0 read ram1 first, then change
63
64  always_ff @ (posedge clk, negedge nrst)
65      if(~nrst) begin
66          wea <=1;
67          web <=0;
68      end else begin
69          if(din_col==9'd511 & din_row == 9'd511 & din_valid) begin
70              wea <=~wea;
71              web <=~web;
72          end
73      end
74
75  //buffer raedy signal
76  logic ready;
77  always_ff @ (posedge clk, negedge nrst)
78      if(~nrst)
79          ready <= '0;
80      else if(din_valid & dout_cnt_control=='1)
81              ready <= ~ready;
82
83  // dout valid control
84  always_ff @ (posedge clk, negedge nrst)
85      if(~nrst)
86          dout_valid <= '0;
87      else
88          if(din_valid & ready==1)
89              dout_valid <= 1'd1;
90
91  //dout control
92
93  assign dout = (ready)?douta:doutb;
94
95  endmodule
```

G

Bibliography

[1] J. Zeng, L. Kuang, M. Cacho-Soblechero, and P. Georgiou, "An ultra-high frame rate ion imaging platform using isfet arrays with real-time compression," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 15, no. 4, pp. 831–832, 2021. DOI: 10.1109/TBCAS.2021.3105328.

[2] V. Elamaran and A. Praveen, "Comparison of dct and wavelets in image coding," in *2012 International Conference on Computer Communication and Informatics*, 2012, pp. 1–4. DOI: 10.1109/ICCCI.2012.6158923.

[3] S. Naveen Kumar, M. V. Vamshi Bharadwaj, and S. Subbarayappa, "Performance comparison of jpeg, jpeg xt, jpeg ls, jpeg 2000, jpeg xr, hevc, evc and vvc for images," in *2021 6th International Conference for Convergence in Technology (I2CT)*, 2021, pp. 1–8. DOI: 10.1109/I2CT51068.2021.9418160.

[4] K. Arora and M. Shukla, "A comprehensive review of image compression techniques," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 2, pp. 1169–1172, 2014.

[5] R. Maini and S. Mehra, "A review on jpeg 2000 image compression," *International Journal of Computer Applications*, vol. 11, no. 9, pp. 43–47, 2010.

[6] C. A. Párraga, T. Troscianko, and D. Tolhurst, "Spatiochromatic properties of natural images and human vision," *Current biology*, vol. 12, no. 6, pp. 483–487, 2002.

[7] J. Yu, "Advantages of uniform scalar dead-zone quantization in image coding system," in *2004 International Conference on Communications, Circuits and Systems (IEEE Cat. No.04EX914)*, vol. 2, 2004, 805–808 Vol.2. DOI: 10.1109/ICCCAS.2004.1346303.

[8] C. Christopoulos, A. Skodras, and T. Ebrahimi, "The jpeg2000 still image coding system: An overview," *IEEE Transactions on Consumer Electronics*, vol. 46, no. 4, pp. 1103–1127, 2000. DOI: 10.1109/30.920468.

[9] R. Singh and V. K. Srivastava, "Jpeg2000: A review and its performance comparison with jpeg," in *2012 2nd International Conference on Power, Control and Embedded Systems*, 2012, pp. 1–7. DOI: 10.1109/ICPCES.2012.6508098.

[10] M. Rabbani, "JPEG2000: Image Compression Fundamentals, Standards and Practice," *Journal of Electronic Imaging*, vol. 11, no. 2, p. 286, 2002. DOI: 10.1117/1.1469618. [Online]. Available: https://doi.org/10.1117/1.1469618.

[11] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The jpeg 2000 still image compression standard," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 36–58, 2001. DOI: 10.1109/79.952804.

[12] G. Hudson, A. Léger, B. Niss, and I. Sebestyén, "Jpeg at 25: Still going strong," *IEEE MultiMedia*, vol. 24, no. 2, pp. 96–103, 2017. DOI: 10.1109/MMUL.2017.38.

[13] G. Wallace, "The jpeg still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992. DOI: 10.1109/30.125072.

[14] D. Granrath, "The role of human visual models in image processing," *Proceedings of the IEEE*, vol. 69, no. 5, pp. 552–561, 1981. DOI: 10.1109/PROC.1981.12024.

[15] S. A. Khayam, "The discrete cosine transform (dct): Theory and application," *Michigan State University*, vol. 114, no. 1, p. 31, 2003.

[16] M. Marcellin, M. Gormish, A. Bilgin, and M. Boliek, "An overview of jpeg-2000," in *Proceedings DCC 2000. Data Compression Conference*, 2000, pp. 523–541. DOI: 10.1109/DCC.2000.838192.

[17] M. Unser and T. Blu, "Mathematical properties of the jpeg2000 wavelet filters," *IEEE Transactions on Image Processing*, vol. 12, no. 9, pp. 1080–1090, 2003. DOI: 10.1109/TIP.2003.812329.

[18] D. Taubman, "High performance scalable image compression with ebcot," *IEEE Transactions on Image Processing*, vol. 9, no. 7, pp. 1158–1170, 2000. DOI: 10.1109/83.847830.

[19] D. Taubman and M. Marcellin, "Jpeg2000: Standard for interactive imaging," *Proceedings of the IEEE*, vol. 90, no. 8, pp. 1336–1357, 2002. DOI: 10.1109/JPROC.2002.800725.

[20] K. Andra, C. Chakrabarti, and T. Acharya, "A high-performance jpeg2000 architecture," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 3, pp. 209–218, 2003. DOI: 10.1109/TCSVT.2003.809834.

[21] K. Yoon, S. Bae, S. Choi, and J. Choi, "The lifting-based dwt filter hardware design for jpeg2000," in *2008 International SoC Design Conference*, vol. 01, 2008, pp. I-160-I–163. DOI: 10.1109/SOCDC.2008.4815597.

[22] A. Katharotiya, S. Patel, and M. Goyani, "Comparative analysis between dct & dwt techniques of image compression," *Journal of information engineering and applications*, vol. 1, no. 2, pp. 9–17, 2011.

[23] A. Horé and D. Ziou, "Image quality metrics: Psnr vs. ssim," in *2010 20th International Conference on Pattern Recognition*, 2010, pp. 2366–2369. DOI: 10.1109/ICPR.2010.579.

[24] K. Slavakis and I. Yamada, "Biorthogonal bases of compactly supported matrix valued wavelets," in *ISSPA '99. Proceedings of the Fifth International Symposium on Signal Processing and its Applications (IEEE Cat. No.99EX359)*, vol. 2, 1999, 981–984 vol.2. DOI: 10.1109/ISSPA.1999.815836.

[25] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *Journal of Fourier Analysis and Applications*, vol. 4, pp. 247–269, 1998. DOI: https://doi.org/10.1007/BF02476026. [Online]. Available: https://link.springer.com/article/10.1007/BF02476026.

[26] M. W. Marcellin, M. A. Lepley, A. Bilgin, T. J. Flohr, T. T. Chinen, and J. H. Kasner, "An overview of quantization in jpeg 2000," *Signal Processing: Image Communication*, vol. 17, no. 1, pp. 73–84, 2002, JPEG 2000, ISSN: 0923-5965. DOI: https://doi.org/10.1016/S0923-5965(01)00027-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0923596501000273.

[27] C.-J. Lian, K.-F. Chen, H.-H. Chen, and L.-G. Chen, "Analysis and architecture design of block-coding engine for ebcot in jpeg 2000," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 3, pp. 219–230, 2003. DOI: 10.1109/TCSVT.2003.809833.