

2007

The study and HDL implementation of the JPEG 2000 MQ coder

David B. Mundy
University of Dayton

Follow this and additional works at: https://ecommons.udayton.edu/graduate_theses

Recommended Citation

Mundy, David B., "The study and HDL implementation of the JPEG 2000 MQ coder" (2007). *Graduate Theses and Dissertations*. 4576.
https://ecommons.udayton.edu/graduate_theses/4576

This Thesis is brought to you for free and open access by the Theses and Dissertations at eCommons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of eCommons. For more information, please contact mschlangen1@udayton.edu, ecommons@udayton.edu.

THE STUDY AND HDL IMPLEMENTATION OF THE
JPEG 2000 MQ CODER

A Thesis

Submitted to
the Engineering School of the
UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for
The Degree
Master of Science in Electrical Engineering

by

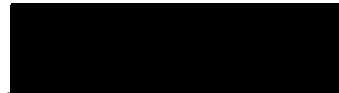
David B. Mundy, B.E.E
UNIVERSITY OF DAYTON

Dayton, Ohio

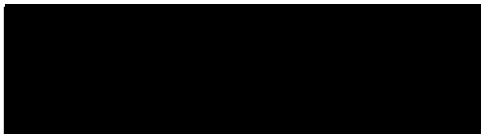
May 2007

THE STUDY AND HDL IMPLEMENTATION OF THE JPEG 2000 MQ CODER

APPROVED BY:



Frank A. Scarpino, Ph.D.
Adviser Committee Chairman
Professor, Electrical & Computer Engineering



John Weber, Ph.D.
Committee Member
Professor, Electrical & Computer
Engineering



Eric Balster, Ph.D.
Committee Member
Research Engineer.
Information Directorate
Air Force Research Laboratory



Donald L. Moon, Ph.D.
Associate Dean, School of Engineering



Joseph E. Saliba, Ph.D., P.E.
Dean, School of Engineering

ABSTRACT

THE STUDY AND HDL IMPLEMENTATION OF THE JPEG 2000 MQ CODER

Name: David B. Mundy
University of Dayton, 2007

Advisor: Frank A. Scarpino

The JPEG 2000 compression system provides higher compression rates with comparable image quality when compared against compression standards such as JPEG. JPEG 2000 also offers features not available in some current standards such as resolution scalability, SNR scalability, and region of interest coding. These features and compression results however do come with tradeoffs. These tradeoffs are mainly complexity and processing requirements. The JPEG 2000 is well known to be much more complex and have much more intensive processing requirements than its predecessor JPEG. These are major hurdles for JPEG 2000 implementors; especially in environments which require real-time compression capability. These requirements are further exacerbated in environments requiring low power consumption which often can not utilize powerful general purpose processors. This thesis discusses the MQ-coder; the arithmetic encoder specified by the JPEG 2000 standard and offers a hardware implementation in an effort to supplement some of the software processing requirements, as well as reducing the overall compression times.

ACKNOWLEDGMENTS

I would like to thank...

- **Frank Scarpino, PhD:** For all of your guidance, advice, and mentoring throughout my education both as an undergraduate and a graduate student. Without your persistent dedication to facilitating opportunities for graduate students, myself, and many others would not be where we are today.
- **John Weber, PhD:** For your participation in my graduate thesis committee in addition to your efforts in facilitating the graduate research programs at The University of Dayton and Wright Patterson Air Force Base.
- **Eric Balster, PhD:** For your assistance in learning many important aspects of image compression technology, as well as lending a hand in deciphering the JPEG 2000 standard. In addition I appreciate your participation in my graduate thesis committee.
- **Kerry Hill, Al Scarpelli, and Rob Ewing:** I would like to thank you all, as well as the rest of the researchers and managers at Wright-Patterson Air Force Base, whose financial support and use of their resources made this thesis and my continuing education possible.
- **My parents, Bill and Barb Mundy:** For all your love and encouragement through my life. Without your support, none of this would be possible.

TABLE OF CONTENTS

	Page
Approval	ii
Abstract	iii
Acknowledgments	iv
List of Tables	viii
List of Figures	ix
 Chapters:	
1. Introduction	1
1.1 The need for Compression	1
1.1.1 JPEG2000 Image Compression Standard and its Features .	3
1.1.2 The Cost of Compression Efficiency	5
1.1.3 The Opportunities for Parallelism	6
1.2 Thesis Organization	7
2. Information Theory	8
2.1 A Brief Introduction to Information Theory	8
2.2 Application of Entropy	11
3. The JPEG2000 Project Overview	13
3.1 JPEG 2000 Image Compression Methodology	13
3.1.1 JPEG 2000 Top Level Overview for Lossy Image Compression	14

4.	The MQ Coder	32
4.1	Introduction	32
4.2	Arithmetic Coding	32
4.2.1	Arithmetic Coding Example	33
4.3	Binary Arithmetic Encoding	35
4.4	The MQ Coder	36
4.4.1	Data Representation	38
4.4.2	Probability Estimation	38
4.4.3	Context State	40
4.5	MQ-Algorithm	40
4.6	Bit Stuffing	42
4.6.1	Flushing	43
5.	Hardware Implementation	44
5.0.2	VHDL	44
5.1	Approach	44
5.1.1	JPEG 2000 Software Development	45
5.1.2	VHDL Test Benching	45
5.1.3	Hardware Testing	46
5.2	HDL Design	48
5.2.1	Probability State Transition Table	48
5.2.2	Context State Memory	49
5.2.3	The State Machines	51
5.2.4	Results	58
5.2.5	Performance	60
5.2.6	Design Tradeoffs	62
5.2.7	Speedup	63
6.	Conclusions & Future Work	65
6.1	Conclusion	65
6.2	Future Work	65
Appendices:		
A.	C Language MQ-Coder Implementation	67
B.	VHDL Language MQ-Coder Implementation	74

C. Reference	90
C.1 JPEG 2000 Reference	91
C.2 MQ Coder Reference	92
Bibliography	96

LIST OF TABLES

Table	Page
C.1 Context Values for κ^{sig} Used as Context or to Form Context for All Passes	91
C.2 Context Values for κ^{mag} Used in Magnitude Refinement Pass	91
C.3 Context Values for κ^{sign} and χ^{flip} Used in Sign Coding	92
C.4 MQ Context State Initialization	92
C.5 MQ-Coder Probability State Transition Table	93

LIST OF FIGURES

Figure	Page
1.1 Example of JPEG2000 Image Encoded with Resolution Scalability . .	4
1.2 Example of JPEG2000 Image Encoded with SNR Scalability	5
1.3 Example of JPEG2000 Image Encoded with Region of Interest Coding	5
2.1 Binary Entropy Plot	11
3.1 Overview of JPEG 2000 Encoding System	14
3.2 Graphical Representation of the JPEG 2000 Level Offset Procedure .	15
3.3 Discrete Wavelet Transform with Sub-band Annotations	18
3.4 One Level and Two Level Discrete Wavelet Transform Decomposition	19
3.5 Graphical Representation of Deadzone Scalar Quantization	21
3.6 Graphical Representation of Image Bit-planes	23
3.7 Encoding Scan Pattern	25
3.8 Rate-Distortion Information to Quality Layers	30
3.9 JPEG 2000 Packet Construction	31
4.1 Arithmetic Encoding Example (Encoding Symbol String B, A, A, B, C)	34
4.2 Representation of Coding Interval for Binary Arithmetic Coder	36
4.3 MQ Coder Coding Process Example	37
4.4 MQ-coder Coding Interval Representation.	38
4.5 MQ Coder Interval Normalized to 16-bits	39
4.6 MQ Coder Data Representation/Registers	39
5.1 VHDL Test Bench Verification Approach	46
5.2 Hardware Verification	47
5.3 JPEG 2000 Hardware MQ-Coder Test Scheme	48
5.4 Overview of Hardware Architecture	49
5.5 Probability State Transition Table Module Interface	49
5.6 Context State Memory Module Interface	50
5.7 Encode State Machine	54
5.8 Encoding State Machine State Transition Table	55
5.9 Transfer Byte State Machine	56
5.10 Transfer Byte State Transition Table.	58
5.11 Encoder Flush State Machine	59
5.12 Encoder Flush State Transition Table	60

5.13	Software Encoding Times Per Symbol Encoded	62
5.14	Hardware Clocks Per Symbol Encoded	63
5.15	Software Hardware Comparison	64
C.1	MQ Encoder Main Routine (MQ-Encode(x, κ))[15, p.646]	94
C.2	MQ Encoder Initialization Routine[15, p.478]	94
C.3	MQ Encoder Transfer Byte Routine (Transfer-Byte \bar{T}, C, L, \bar{t})[15, p.479]	95
C.4	MQ Encoder Put Byte Routine (Put-Byte(\bar{T}, L))[15, p.479]	95
C.5	MQ Encoder Flush Routine[15, p.496]	95

CHAPTER 1

Introduction

This thesis topic is inspired by the new image compression standard called JPEG 2000. This standard was created by the Joint Pictures Expert Group as a successor to the JPEG standard. In addition to JPEG 2000, which compresses still images, there is an extension which allows for motion. This is called Motion JPEG 2000. After producing a software implementation of JPEG 2000 for still images, it became evident that a software only implementation of either JPEG 2000 or Motion JPEG 2000 that could operate in near real-time would be difficult if not impossible with the currently available general purpose processing resources. Due to JPEG 2000's computational complexity, and bit-level manipulations, a hardware implementation of selected portions of the standard became more appealing. One of the main sources of latency in the JPEG 2000 standard is the arithmetic encoder[14], known as the MQ-Coder. Therefore, the MQ-Coder is a logical place to start when adding hardware acceleration to the JPEG 2000 compression system.

1.1 The need for Compression

Typically (color) images have 3 color planes where each is represented by an 8-bit value from 0 through 255. Thus, each pixel requires 24-bits of data to convey the color information. When this is multiplied by the number of pixels in an image of

reasonable size, the number of bits required to convey the image is nontrivial. For example, a standard VGA image is 640 pixels wide and 480 pixels high. Thus, the total number of bits required to store this information is

$$\begin{aligned}\text{bits} &= 640 \cdot 480 \cdot 24 = 7,372,800\text{bits} \\ &= 921,600\text{Bytes}\end{aligned}\tag{1.1}$$

Although by today's standards, nine hundred kilobytes is not a substantial amount of data. However, if one extrapolates this image into either a series of images or a video, the storage space, and possible transmission bandwidth requirements add up quickly. For example, if a video, which typically runs at thirty frames per second plays for only one minute, would require 1,658,880,000 Bytes of storage. This is over 1 gigabyte, of data for only a one minute video. In addition, the resolution of the previous example is relatively small in light of today's standards and for applications such as high-definition television, and digital cameras, which commonly have resolutions in the 4 to 6 megapixel range.

There are many different techniques and standards for compressing both still images and video, and often these two domains often share some of the same techniques. In video compression such as MPEG, three different types of frames exist; inner frames, predictive frames, and back predictive frames. Periodically, an inner frame, which consists of an entire image, is placed into the compressed code-stream. The predictive and back predictive frames then key off of the information in the inner frames and only store the differences from the inner frame. When an inner frame is needed, the image is compressed in a similar manner to JPEG. Sometimes the compression of video is much simpler. In the case of motion JPEG 2000, the motion

is produced by a simple series of still images compressed with JPEG 2000. While the still image JPEG 2000 is complex, the compressed motion code-stream relatively simple. Motion JPEG 2000 under most circumstances does not provide as much compression as MPEG, but offers other advantages such as random access to individual frames without the need to reconstruct it from predictive frames.

1.1.1 JPEG2000 Image Compression Standard and its Features

The JPEG 2000 compression standard became an international standard in December 2000[8]. Although up to now it has not been widely adopted for general purpose applications, it is beginning to show up in more specialized applications. For example, the medical community has begun to use the lossless mode (often referred to as "reversible") of JPEG 2000 to store data from medical imaging equipment such as MRI's and x-rays. The reversible mode of the standard is of particular interest to the medical community because the image is not degraded from its original form, and the only distortion comes from the equipments native spatial resolution. Even though no relevant data is "thrown away" in the reversible mode, it still offers a compression ratio of around 2.5:1[6].

The JPEG 2000 standard in lossy mode (irreversible mode) typically compresses images with %20 greater efficiency over its predecessor; JPEG[15]. In addition, the new standard adds capabilities which are very attractive. Some of these new capabilities include resolution scalability, SNR scalability, and region of interest coding.

Resolution scalability allows the image resolution to increase as more of the code-stream is decoded. Resolution scalability also adds the ability to transmit a subset of the full code-stream to a remote location depending on the resolution desired at that

location. For example, in a military field application, reconnaissance imaging may be stored on a central server. A base station which has large, high resolution screens and access to large amounts of bandwidth to transport the code-stream may want the entire high resolution image. In contrast, there may field teams that only have the capability to view the same reconnaissance images on a small PDA. Because of the PDA's small screen and limited resolution, it would be a waste in both bandwidth and memory to transport the entire high resolution image.



Figure 1.1: Example of JPEG2000 Image Encoded with Resolution Scalability

SNR scalability improves the image *quality* as more of the code-stream is decoded. In other words, as more of the code-stream is decoded, the distortion originally caused by the compression is decreased, and thus the SNR increases.

The region of interest (ROI) capability allows for various spatial regions of the image to be coded with less or no compression, thus reducing the distortion, yielding a more clear image.



Figure 1.2: Example of JPEG2000 Image Encoded with SNR Scalability



Figure 1.3: Example of JPEG2000 Image Encoded with Region of Interest Coding

1.1.2 The Cost of Compression Efficiency

As discussed previously, the JPEG 2000 standard compresses images with a greater coding efficiency than its predecessor JPEG [15]. This increase in coding efficiency has the effect of JPEG 2000 requiring on average 5.3 times longer to compress an image than JPEG[6]. This result leads to the conclusion that JPEG 2000 is significantly more complex than JPEG[6]. Thus, to compress an image with JPEG 2000 in the same amount of time requires significantly more processing resources than it would

with JPEG. JPEG 2000 and JPEG share some of the same processing steps, but often in JPEG 2000, these steps examine the image samples with finer granularity. For example, the encoding process in JPEG 2000 is termed Embedded Block Coder with Optimal Truncationb (EBCOT). This encoder takes blocks of the original image (which is similar to JPEG) however, it then encodes each bit-plane¹ one at a time with a combination of three coding passes. Therefore, every bit in the picture is visited by the bit-plane encoder three times. Although each bit is visited three times, only one of these three will result in the arithmetic encoder coding any information. In addition to having a complex encoding scheme which requires bit-level access; after each of the previously mentioned coding passes is complete, the encoder must make a distortion estimation to determine how much the distortion is decreased with the additional information encoded by the previous pass. These distortion estimates, along with the length of each code stream produced by the encoder are saved and then used to truncate the resulting code stream in an optimal location. Because of the additional complexity, the compression of a JPEG 2000 image takes considerable resources. This problem is exacerbated with a low power requirement. Therefore, for embedded implementations such as military applications, a hardware accelerated implementation is a very attractive option.

1.1.3 The Opportunities for Parallelism

The JPEG 2000 committee understood the complexity and the computational requirements of its future standard. Thus, they built into the standard provisions to

¹A bit plane is a 2 dimensional array of the bits that exist within the same significance level of the pixels of an image. This is described in more detail in Section 3.1.1.

allow for an image elements to be compressed in parallel. Every processing environment has limited resources, however JPEG 2000 supports a virtually unlimited image size. The maximum image size is $2^{32} \times 2^{32}$ pixels, or 4,294,967,296 x 4,294,967,296 pixels, or 18,446,744,073,709,551,616 pixels. Thus, this nearly unimaginable image size can be broken into pieces called *tiles*. These tiles are then processed completely independently tile by tile, thus can be processed in parallel. Once, the tiles are independently processed, they are combined into the JPEG 2000 bit-stream. Because each tile is processed independently, some gain in compression efficiency can be achieved. However, small tiles at low bit-rates, can lead to blocking artifacts[16]. Therefore, the tile sizes must be chosen wisely.

1.2 Thesis Organization

The following chapter begins with a brief introduction to the ideas and concepts behind information theory. This chapter is followed by an introduction to the JPEG 2000 compression system. Chapter 3 is intended to give the reader an overall idea of where the MQ-coder fits into the JPEG 2000 compression scheme. The thesis then moves on in Chapter 4 which discusses the MQ-coder algorithm and arithmetic coding in general. Also introduced is the precise algorithm implemented in hardware. Chapter 5 discusses the hardware implementation, along with the design and hardware verification process. This chapter also provides performance results. The final chapter provides some conclusions based upon the work in this thesis and provides ideas for possible future work.

CHAPTER 2

Information Theory

2.1 A Brief Introduction to Information Theory

Essentially, information theory provides methods to quantify the amount of information a particular object contains. In this line of thinking, information is anything that *reduces uncertainty*[13], which is often measured in terms of degrees of freedom. For example, a binary digit has two possible symbols; either 1 or 0, which would have an uncertainty of 2 symbols. To further quantify this example using logarithms (using base 2 because our units are *bits*), the uncertainty for two equally likely symbols is [13].

$$\log_2(2) = 1\text{bits} \quad (2.1)$$

Suppose symbols are not equally likely. In this case, the uncertainty may be less than the 1. For example, a computer file may contain mostly zeros with sparsely distributed ones throughout the file. Thus, the uncertainty is somewhere between $\log_2(1)$ and $\log_2(2)$ [13]. The final uncertainty is determined by the probability that a symbol will occur. If the file contains 100 total bits, with 95 zeros, and 5 are ones, then,

$$\begin{aligned} P_0 &= \frac{95}{100} = 0.95 \\ P_1 &= \frac{5}{100} = 0.05 \end{aligned} \quad (2.2)$$

By rearranging Equation 2.1 and substituting M as the number of possible symbols, the uncertainty U of a symbol can be determined based upon its probability of occurring[13].

$$\begin{aligned}
 \log_2(2) &= \log_2(M) \text{ where } M = 2 \\
 &= -\log_2(M^{-1}) \\
 &= -\log_2\left(\frac{1}{M}\right) \\
 U &= -\log_2(P)
 \end{aligned} \tag{2.3}$$

Using Equation 2.3 and the probabilities from Equation 2.2, It is easy to see that when the probability of a symbol is high, the uncertainty is low, as shown in Equations 2.4 and 2.5.

$$U_0 = -\log\left(\frac{95}{100}\right) \approx 0.0223 \text{bits} \tag{2.4}$$

$$U_1 = -\log\left(\frac{5}{100}\right) \approx 1.3010 \text{bits} \tag{2.5}$$

The sum of the probabilities of the possible symbols must equal 1, thus,

$$\sum_{i=1}^M P_i = 1 \tag{2.6}$$

The uncertainty of any symbol relates to the probability in the following manner

$$U_{P \rightarrow 1} = \lim_{P \rightarrow 1} (-\log_2(P)) = 0 \tag{2.7}$$

$$U_{P \rightarrow 0} = \lim_{P \rightarrow 0} (-\log_2(P)) = \infty \tag{2.8}$$

Now, as the probability of any symbol approaches zero, it becomes less and less likely that the symbol will appear. Thus, if the symbol appears, it will be very *surprising*. This leads to the term *surprisal* as coined by Tribus[13]. By analogy, if the probability

of a symbol appearing is zero, then it will be infinitely surprising if it does appear. The surprisal for the i^{th} symbol can be found as

$$u_i = -\log_2(P_i) \quad (2.9)$$

If we now say that uncertainty is the average surprisal[13] for a string of length N with M unique symbols and that the i^{th} symbol shows up N_i times, then the total number of symbols in the string can be found as

$$N = \sum_{i=1}^M N_i. \quad (2.10)$$

For a given string of symbols, there is M surprisals u_i and for each of the M unique symbols, a total of N_i symbols of type M_i . Thus, the total average surprisal for the string is found as

$$\frac{\sum_{i=1}^M N_i u_i}{\sum_{i=1}^M N_i} = \sum_{i=1}^M \frac{N_i}{N} u_i. \quad (2.11)$$

Now, substituting $\frac{N_i}{N}$ for the probability of each symbol P_i the average surprisal is

$$H = \sum_{i=1}^M P_i u_i. \quad (2.12)$$

Finally, Shannon's general equation for uncertainty or *entropy* is found by substituting in u_i from Equation 2.9[13].

$$H = - \sum_{i=1}^M P_i \log_2(P_i) \quad (2.13)$$

The units for entropy H is bit in this case because the units of the base two logarithm is bits per symbol. The graph in Figure 2.1 shows the H function for two symbols. This represents the uncertainty of the second symbol occurring given the probability of the first symbol. Notice that the uncertainty is a maximum when both symbols are equally likely (i.e. $P = 0.5$).

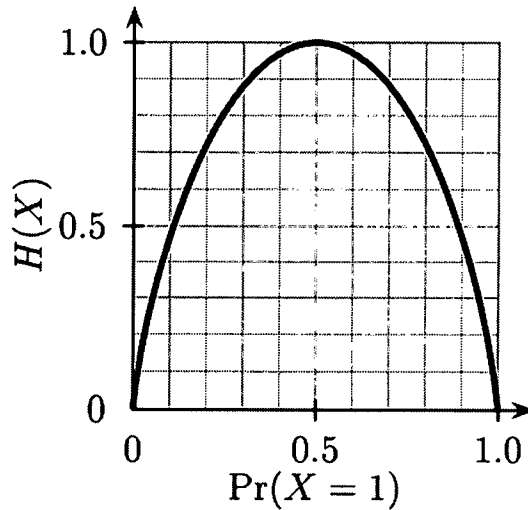


Figure 2.1: Binary Entropy Plot

2.2 Application of Entropy

What exactly does it mean to say an image requires for example, 1.5 bits per pixel? This simply means that on average, the information contained within each pixel requires 1.5 bits to represent in binary form. To expand this example, suppose that each pixel of an image is represented by three different components (red, blue, green), each requiring eight bits. This would be a total of $B = (3)(8) = 24$ bits per pixel. 24 bits per pixel, however, may not be the minimum number of bits required to represent the information. This is what Shannon's general equation for entropy tells us; the average lower bound for which a given set of information can be represented[15]. Therefore, the upper bound is B bits per pixel with a lower bound of H bits per pixel.

$$H \leq B \tag{2.14}$$

The upper bound of B bits per pixel is only reached if the data contained is *entirely random* [15]. Coding schemes allow for the lower bound H to be approached *arbitrarily closely* if the complexity of the coding scheme is unbounded[15]. Strings of symbols that are not entirely random contain redundancies that the an encoding scheme can exploit.

CHAPTER 3

The JPEG2000 Project Overview

The work encompassed by this thesis started in a reconfigurable computing and embedded systems group. There was a desire by the US Air Force to develop a JPEG 2000 encoder for military use in an embedded environment. This desire led to the research and initial development of a software JPEG 2000 encoding system. The software development was in part to learn how JPEG 2000 operates, as well as to develop building blocks for transitioning pieces of the encoding scheme into reconfigurable computing resources such as FPGA's. During the development, it became apparent that for low-power environments, custom dedicated hardware such as an FPGA would be highly advantageous. This realization led to the first such hardware transition; the MQ-Coder. This chapter contains a brief overview of the JPEG 2000 compression scheme. This chapter is not meant to be a definitive guide. Its purpose however, is to give the reader an overview of the JPEG 2000 compression system and how the MQ-coder fits into the overall scheme. For a detailed description of the JPEG 2000 compression scheme, please see [15] or [4].

3.1 JPEG 2000 Image Compression Methodology

The JPEG 2000 codec follows processing steps found in many of the methods often found in image compression community. This section offers more specifics on

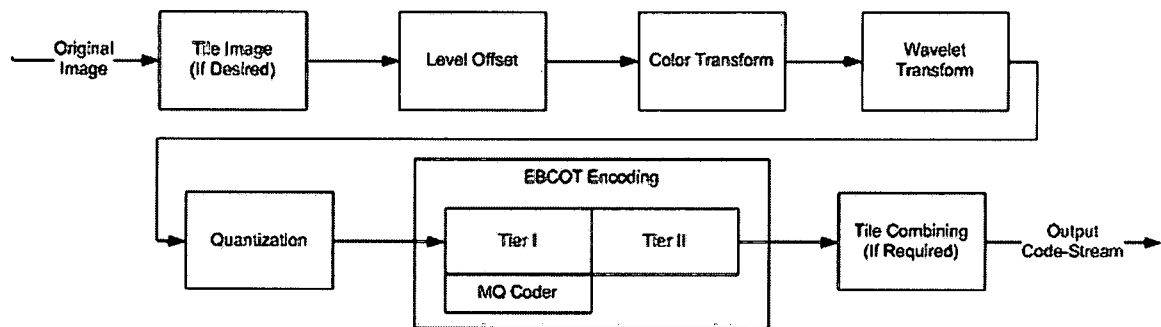


Figure 3.1: Overview of JPEG 2000 Encoding System

how JPEG 2000 works internally as well as how it natively offers the capability for concurrent processing of different data elements. For reference, Figure 3.1 shows a high level pictorial outline of the JPEG 2000 compression system.

3.1.1.1 JPEG 2000 Top Level Overview for Lossy Image Compression

Image Tiling

Before an image is compressed, it may be desirable to split the image into pieces to compress separately. This process is typically useful when either compressing a very large image, or if the compression system has limited resources such as memory. The original image is split into *tiles* prior to being compressed. The compression system then treats each tile as a separate image. Once the compression system has formed the individual code-streams for each tile, all of the separate pieces are combined into the final code-stream. Tiling the image does not come without cost, such as "JPEG like" blocking artifacts at the tile boundaries at low bit-rates[16].

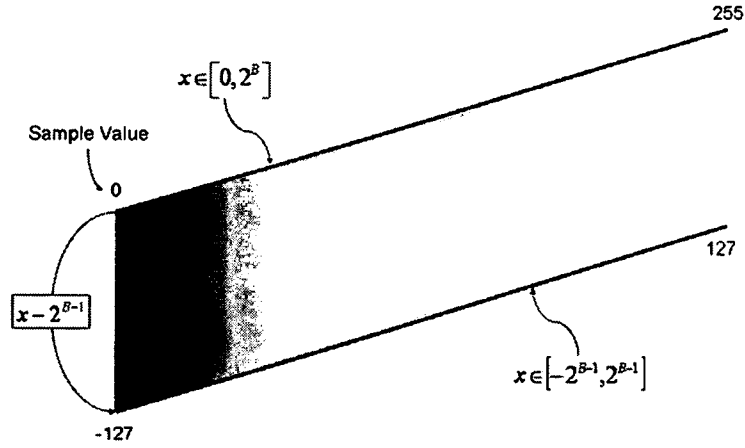


Figure 3.2: Graphical Representation of the JPEG 2000 Level Offset Procedure

Level Offset

After tiling, the level offset is performed. This operation is conditional based on whether the original image is signed or unsigned. If the data is unsigned, hence has no negative values, the data is shifted by the factor of -2^{B-1} where B is the bit-depth of the original image. This will ensure that the values are bounded by

$$-2^{B-1} \leq x < 2^{B-1} \quad (3.1)$$

The level offset is desirable because it centers the data about zero. After the discrete wavelet transform is performed, the high pass content will have a distribution that is centered about zero, making the resulting data easier to compress[15]. See Figure 3.2 for a graphical representation of the level offset.

Color Transform

The purpose of the transform is to exploit the spectral redundancies between the three color planes[8]. Compression systems often use a color transform to leverage the images spectral redundancy to decrease the final required bit-rate of the image. One popular color transform known as either the RGB to YUV color transform, or RGB to Y,Cb,Cr color transform. Color images are composed of three color component planes; red, green, and blue. Generally, the color information for an image is distributed evenly across color planes. Thus, each color plane is equally *hard* to compress. The color transform redistributes the information such that the distribution is no longer even. After transformation, the majority of the image's information is contained in the Y color plane; also known as the as the *luminance* color plane. This component when view without the other two color planes is essentially a grey scale image of the original color image. The remaining color information is distributed across the two remaining color planes and are often referred to as the *chrominance components* [15]. The redistribution of information increases the entropy in the Y color plane, however, the other two color planes' entropy is decreased allowing for a net higher rate of compression. The color transform also allows for the two chrominance color planes to be downsampled or *decimated* with very little distortion detectable by the human eye. Downsampling often occurs by a factor of 2 in both X and Y directions effectively shrinking the chrominance components to one quarter of their original resolution. Downsampling is desirable because "the human visual system is known to be substantially less sensitive to distortion in the chrominance components than the luminance component"[15]. An example of the effects of downsampling pre and post color transform is shown in [7]. The JPEG 2000 standard specifies a different

color transform depending on if the lossless, or lossy mode is used. In the JPEG 2000 vernacular, the color transform is known as the irreversible color transform, or the ICT [15]. The color transform is optional, but provides substantial improvement in compression efficiency and is therefore recommended for most image types[12]. Below, the color transform is shown as a relationship of weighted sums and differences.

$$\alpha_R \triangleq 0.299, \alpha_G \triangleq 0.587, \alpha_B \triangleq 0.114$$

$$x_Y \triangleq \alpha_R x_R + \alpha_G x_G + \alpha_B x_B \quad (3.2)$$

$$x_{Cb} \triangleq \frac{0.5}{1 - \alpha_R} (x_B - x_Y) \quad (3.3)$$

$$x_{Cr} \triangleq \frac{0.5}{1 - \alpha_R} (x_R - x_Y) \quad (3.4)$$

Where x_R , x_G , and x_B are pixels in the same spatial location of red, green and blue corresponds respectively to x_Y , x_{Cb} , and x_{Cr} which represent the luminance and two chrominance components. The above equations may also be expressed in a matrix multiplication equivalent

$$\begin{pmatrix} x_Y \\ x_{Cb} \\ x_{Cr} \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{pmatrix} \begin{pmatrix} x_R \\ x_G \\ x_B \end{pmatrix}$$

The color transform is unity gain, thus does not effect the bit-depth of the original image [15]. The unity gain property is important when expressing to the decoder how many bits are required to decode the image.

Discrete Wavelet Transform

The JPEG 2000 compression standard utilizes the discrete wavelet transform (DWT). The DWT is similar to the color transform in that it exploits redundancies in the image. The wavelet transform however, exploits the *spatial* redundancy

of the image yielding a reduction in the image's spatial entropy[8]. This reduction in entropy allows for the image to be compressed to a lower bit-rate[15]. There are an infinite number of wavelet transforms[8], however there are only two utilized in the baseline JPEG 2000 standard. The lossy mode of JPEG 2000 uses the CDF 9/7 filter kernel, and the lossless mode uses the 5/3 filter kernel[15]. The wavelet transform is in essence a set of specific high pass and low pass filters which each output a separate sub-signal. The output of the low pass filter is the *trend* of the signal over time, and the output of the high pass filter is the difference signal or *fluctuation* about the trend signal[17]. The image is filtered in both the vertical and horizontal directions then the output for these filters are then decimated by 2 and then stored in a specific location within the output image. The output of the wavelet transform filters is stored into 4 quadrants. The upper left quadrant is the output of the low pass filter in both the vertical and horizontal directions where the lower right corner is the output of the high pass filters in both directions. The two remaining quadrants are a combination of both high and low pass filters. Figure 3.3 depicts the output of the wavelet transform with the quadrants labeled. These quadrants are known as sub-bands.

The discrete wavelet transform can be applied to an image more than once; in an effort to further reduce the spatial entropy of the image. This process, known as multi-resolution analysis, is performed by applying the same set of filters to the LL (upper left) sub-band. This re-analysis of the LL sub-band is depicted in Figure 3.4, which shows the 1 level and 2 level analysis of the same image.

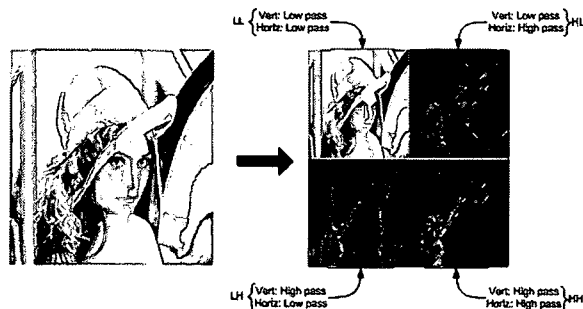


Figure 3.3: Discrete Wavelet Transform with Sub-band Annotations

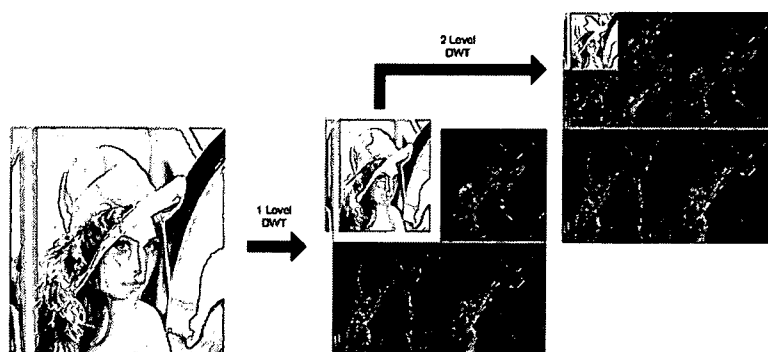


Figure 3.4: One Level and Two Level Discrete Wavelet Transform Decomposition

The wavelet transform has the benefit of not producing blocking artifacts at low bit-rates, which is something that plagues JPEG. Because the wavelet transform encompasses the entire image (or tile) it produces more of a blurring, or smoothing effect; which the human eye is more capable of resolving compared to blocking artifacts[3].

A convenient and straight forward method of implementing the discrete wavelet transform is using the filter bank approach which is a set of FIR filters [8]. The filter bank approach does not take into account the down-sampling that occurs after the image data is passed through the filters. Therefore, extra, redundant processing

is done to the data; half of which is thrown away [8]. To eliminate this redundant processing, the concept of *lifting* was introduced in the 1990's[8]. Lifting is a process which allows for downsampling prior to data filtration without loss of information. In addition, lifting allows the data to be processed in parallel, which is very attractive for hardware implementations of the discrete wavelet transform[8].

Quantization

Quantization is a mapping of numbers from more specific to a less specific values. Quantization is used in JPEG 2000 to reduce complexity of the coefficients created by the wavelet transform. This operation is another transformation that lowers the entropy of the image. However, due to its very nature, quantization is a lossy process. In its simplest form, quantization is applied by simply dividing the pixel value by a constant factor, the resulting value is then rounded down to the nearest integer value. The effect of quantization allows the image to be compressed more efficiently in two different ways. One, the division lowers the value of all of the pixel and thus lowering the number of bits that must be compressed. Two, due to the rounding process, the probability that neighboring pixels are the same value is higher, thus, lowering the frequency content of the image which results in a more compressible image. Specifically, JPEG 2000 utilizes deadzone scalar quantization, defined in Equation 3.5 [15]. In addition, a graphical representation of deadzone scalar quantization is shown in figure 3.5.

$$q_b = \text{sign}(y_b) \left\lfloor \frac{|y_b|}{\Delta_b} \right\rfloor \quad (3.5)$$

Deadzone scalar quantization has the benefit that the zone about zero is twice as wide as the other values. This is advantageous especially in JPEG 2000. Recall in

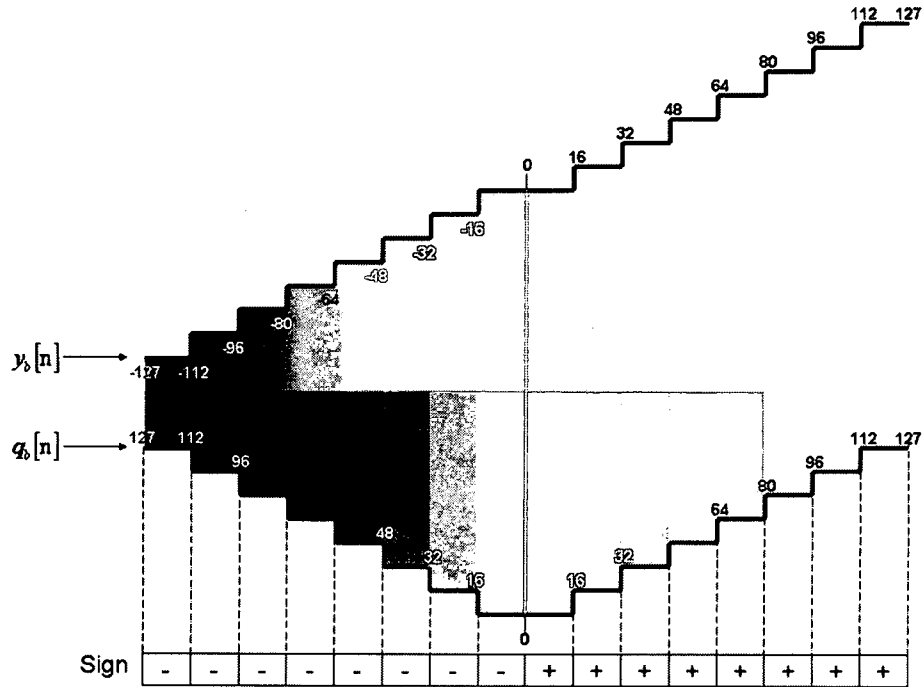


Figure 3.5: Graphical Representation of Deadzone Scalar Quantization

Section 3.1.1 on page 15 the discussion of the level offset procedure. Due to the centering of the information about zero by both the level offset and the wavelet transform, the quantization forcing more values to zero allows for more efficient compression. It should be noted that JPEG 2000 uses a sign and magnitude representation of the image data. Examining equation 3.5 it becomes apparent that it contains two separable components. With χ representing the sign and v representing the magnitude of the sample values, the sample values are now represented as two discrete terms [15].

$$\chi = \text{sign}(y) \quad (3.6)$$

$$q_b = \left\lfloor \frac{|y_b|}{\Delta_b} \right\rfloor \quad (3.7)$$

The purpose of these discrete terms will become more apparent in Section 3.1.1.

Image Partitions

After tiling, JPEG 2000 further partitions the image. All of the partitions are restricted such that any one partition may only lie within one sub-band of the wavelet transform. In other words, a partition may not cross sub-band boundaries. Within each sub-band, the image samples are broken into partitions called code-blocks. These code-blocks are the smallest partition that the image is divided into. Codeblocks are input into the Tier I encoder (Tier I is discussed in Section 3.1.1) and are processed independently from one another. The next largest partition is known as a precinct. Again, the precinct does not cross sub-band boundaries and can contain multiple code blocks. Although, a precinct can not cross sub-band boundaries, a precinct can contain information from multiple sub-bands within the same resolution level of the wavelet transform. For example, precincts that lie in the LL sub-band of the wavelet transform only encompasses data within the LL sub-band. This is because the information contained within the sub-band contains all of the spatial information necessary to reconstruct that resolution level. however, for higher resolution levels, a precinct will encompass the HL, LH, and HH sub-bands. This is because all three of these sub-bands are required to spatially describe the image in that resolution level. The precincts purpose is to divide the image information into *packets*. The precinct packet is organized by Tier II of the JPEG 2000 encoding process (Tier II is described in Section 3.1.1) on page 28.

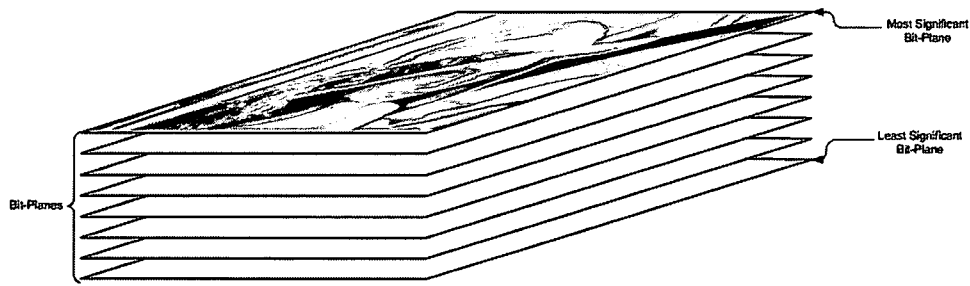


Figure 3.6: Graphical Representation of Image Bit-planes

Tier I Encoding - Entropy Coding

Tier I encoding is the most computationally intensive portion of JPEG 2000. Each code block (as discussed in Section 3.1.1) is processed independently. The Tier I encodes each code block with one of three different coding passes. Based on the image content, *significance* information is derived and is used to determine what coding pass is used to encode the data. Each of the coding passes then derives *context* information from the image content and then passes the context and data value the context adaptive MQ arithmetic encoder.

It should be noted that the JPEG 2000 encodes one *bit-plane* at a time; this is known as *bit-plane encoding*. To understand JPEG 2000, the concept of a bit-plane must be understood. This is a paradigm shift over the traditional thinking of encoding a coefficient's value. In bit-plane coding, the most significant bits are encoded across the entire code block first followed by the next most significant bits in the code block etc. The set of bits within the same binary significance level form a bit-plane. Figure 3.6 shows a graphical representation of bit-planes. There are three possible coding passes, which occur in the following order: 1) Significance Propagation, 2) Magnitude Refinement, and 3) Cleanup. In addition there is a sign encoding procedure which is

only performed once per quantized wavelet coefficient. Tier I coding starts with the most significant bit-plane. The only coding pass that takes place on this bit-plane is the cleanup pass. After the most significant bit-plane encoding is complete, the next most significant bit-plane is encoded using the previously defined coding pass order. Before discussing the purpose of each of the encoding passes, a few terms must be defined.

Each coefficient within a code block has a 3 different "state variables"² who control which coding pass is used on the current bit each coefficient. The value of the state is either a 1 or 0. The first state variable is called the significance σ . Before encoding begins, the significance of each coefficient is initialized to 0. Once the first non-zero magnitude bit of a coefficient is found, the significance transitions to 1[15]. The next state variable is called the delayed significance. The delayed significance $\vec{\sigma}$ is a copy of the significance, however it delayed from the significance value by one bit-plane[15]. The third and final state variable is called coding pass membership π . This value signals to the magnitude refinement pass whether or not the bit was already processed by the significance propagation pass[15].

The scan pattern for JPEG 2000 encoding is not a traditional raster scan pattern, rather, it consists of horizontal stripes which are 4 coefficients high and as wide as the code-block. The scan starts in the upper left corner and proceeds down to the next coefficient. Once the fourth coefficient in the column is reached, the pattern returns to the top of the *stripe row*. Once the entire stripe row is processed, the next stripe row is encoded, this process is represented graphically in Figure 3.7[15].

²The term state variable does not refer to the mathematical modeling of a system, rather is a variable that represents information required for the encoder [15]

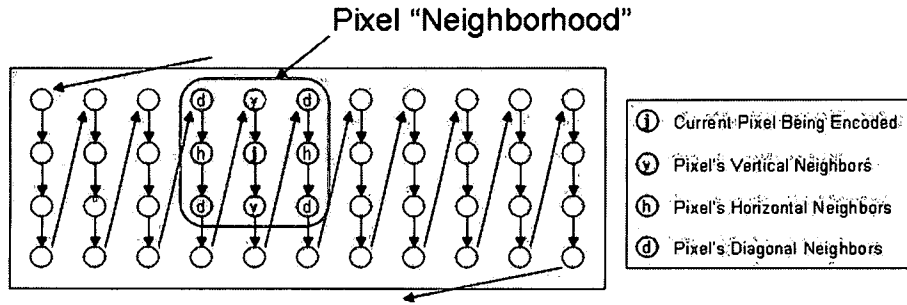


Figure 3.7: Encoding Scan Pattern

1) Significance Propagation

The significance propagation pass only operates on a coefficient's bits that are not already significant. Operating on a bit at a time, this pass determines if any of the coefficient's neighbors³ are significant. If any of the neighboring coefficients are significant, the current coefficient's bit is encoded with the MQ-coder and is made significant by setting $\sigma \leftarrow 1$. Thus, significance propagates outward from coefficients that are already significant. In addition, the coding pass membership state variable $\vec{\sigma}$ is set to 1 indicating that the coefficient's bit has already been encoded by the significance propagation pass and should not be encoded by the magnitude refinement pass.

One element of the JPEG 2000 encoding scheme not discussed yet is the formation of context data which is passed to the MQ-Coder. The Context information is formed differently for each type of coding. The significance propagation pass⁴ uses contexts 0 through 8. These are formed using the values

³A coefficient's neighbors include coefficients that lie next a coefficient in the vertical, horizontal, and diagonal directions.

⁴The type encoding used in the significance propagation pass is known as zero-coding[1].

$$\begin{aligned}
\kappa^h &\triangleq \sigma[j_1, j_2 - 1] + \sigma[j_1, j_2 + 1] \\
\kappa^v &\triangleq \sigma[j_1 - 1, j_2] + \sigma[j_1 + 1, j_2] \\
\kappa^d &\triangleq \sum_{k_1, k_2 \in \{-1, 1\}} \sigma[j_1 + k_1, j_2 + k_2]
\end{aligned} \tag{3.8}$$

Using the values derived in Equation 3.8 in conjunction with the lookup Table C.1 on page 91 the context κ^{sig} is formed[15].

2) Magnitude Refinement

Coefficients that are already significant and not encoded by the significance propagation pass are encoded in the magnitude refinement pass. The magnitude refinement pass on a bit-plane occurs after the significance propagation pass, therefore, it is possible that a coefficient is significant but its bit in the current bit-plane should not be encoded by the magnitude refinement pass. This is the purpose of the coding pass membership state variable $\overrightarrow{\sigma}$. This state variable keeps track of when a bit is encoded by the significance propagation pass. As the name of this pass states, the coefficient's encoded magnitude value is refined. Tier 1 encodes a coefficient's bits from the most significant bit-plane to the least significant bit-plane, thus as bits from progressively less-significant bit-planes are added to the encoded coefficient value, the magnitude is refined to a greater extent for every bit-plane. The magnitude refinement pass also forms context information. This context value is based on 2 different values, 1) the context value κ^{sig} which is formed by the significance propagation pass and 2) the delayed significance. These two values in conjunction with a lookup table form the context value known as κ^{mag} . See Appendix C for the κ^{mag} lookup table[15].

3) Cleanup Pass

The cleanup pass encompasses 2 different coding paths. The first path is similar to the significance propagation pass and uses zero-coding. The second path performs a run-length coding scheme when $\kappa^{\text{sig}} = 0$ for all 4 coefficients within a the column of a stripe row. The extent to which the run length coding can be utilized is of course based upon the content of the image. When run length coding occurs, the length of the run is measured while the bit-value is equal to zero. The run length in conjunction with two different context values are passed to the MQ coder. In addition, any bit not encoded in either the significance propagation or the magnitude refinement pass is encoded in the cleanup pass[15].

Sign Encoding

After quantization, the images coefficients are represented as sign and magnitude components. Due to this, the sign must be encoded as some point. Sign encoding occurs as soon as a coefficient becomes significant. Thus, a coefficient can have its sign encoded in either the significance propagation pass or the cleanup pass.

The sign encoding's context formation is a based on the significance and sign value of the neighboring vertical an horizontal pixels. An intermediate value known as the horizontal an vertical sign bias functions are computed as

$$\begin{aligned}\chi^h &\triangleq \chi[j_1, j_2 - 1]\sigma[j_1, j_2 - 1] + \chi[j_1, j_2 + 1]\sigma[j_1, j_2 + 1] \\ \chi^v &\triangleq \chi[j_1 - 1, j_2]\sigma[j_1 - 1, j_2] + \chi[j_1 + 1, j_2]\sigma[j_1 + 1, j_2]\end{aligned}\tag{3.9}$$

where χ is the pixel sign value $\chi \in -1, 1$ and σ is the significance of the pixel $\sigma \in 0, 1$. These intermediate values are then truncated to the range $[-1, 1]$ with the functions

$$\begin{aligned}\bar{\chi}^h &\triangleq \text{sign}(\chi^h) \min\{1, |\chi^h|\} \\ \bar{\chi}^v &\triangleq \text{sign}(\chi^v) \min\{1, |\chi^v|\}\end{aligned}\tag{3.10}$$

The context κ^{sign} is then determined by lookup table using $\bar{\chi}^h$ and $\bar{\chi}^v$ before the encoding takes place. See Appendix C for the κ^{mag} lookup table as well as a pseudo code representation of the sign coding algorithm.

Rate-Distortion Information

In addition to coding the data, the JPEG 2000 encoder makes distortion estimates after each coding pass. This information is then used for rate control and also determines how the image data is distributed between the quality layers.

Tier II Encoding - Code Stream Formation

Once the individual code blocks are encoded using Tier I, Tier II takes the individual pieces and "stitches" them together in the JPEG 2000 image format. This includes adding header information for the entire image, header information for the tile or tiles, and adding "packet headers" to the individual code blocks and placing them in the code stream in an organized fashion. The complete definition of the Tier II coding structure is beyond the scope of this thesis, however, the main points will be covered. For more detail than is provided here, please refer to [15] or [4].

Packets, Progressions and Precincts

JPEG 2000 provides five different progressions or "scalabilities" in which an image can be encoded. Each progression has different attributes which may be desirable

across a range of applications. Each of these progressions produce "quality layers" which contain a portion of the image's overall data. When all of the quality layers are used to reconstruct an image, the resulting image is of the highest quality available with respect to how the image was encoded[15]. The different progressions are referred to as, resolution scalable, signal to noise ratio scalable, spatially scalable, and component scalable. Each scalability uses a different technique to determine what information defines a quality layer. For example, signal to noise ratio uses the rate-distortion information discussed in section 3.1.1 to determine the portions of a code-block's code stream which go into each quality layer. Figure 3.8 shows the process of defining the quality layers from the rate-distortion information graphically. In contrast, resolution scalability uses the resolution level boundaries defined by the wavelet transform (as discussed in 3.1.1) to determine the quality layers. In a resolution scalable progression, each successive resolution level of the wavelet transform contains twice the resolution of the previous.

Recall from the discussion in Section 3.1.1 that precincts contain at least one code-block from the same spatial area of each sub-band in a wavelet resolution level. Each of these precincts are used to form one packet per quality layer. Therefore, for each precinct, there exists one packet per quality layer. Each sub-band precinct contains its own header information within the packet header, which appears before any of the encoded image information (provided by the MQ coder) appears in the code stream. The order in which the sub-band precincts appear in the code stream is HL, LH, HH [15]. The packet structure is shown graphically in Figure 3.9.

The information included within each precinct packet header is 1) inclusion information, 2) the number of zero bit-planes, 3) the number of coding passes, and 4) The

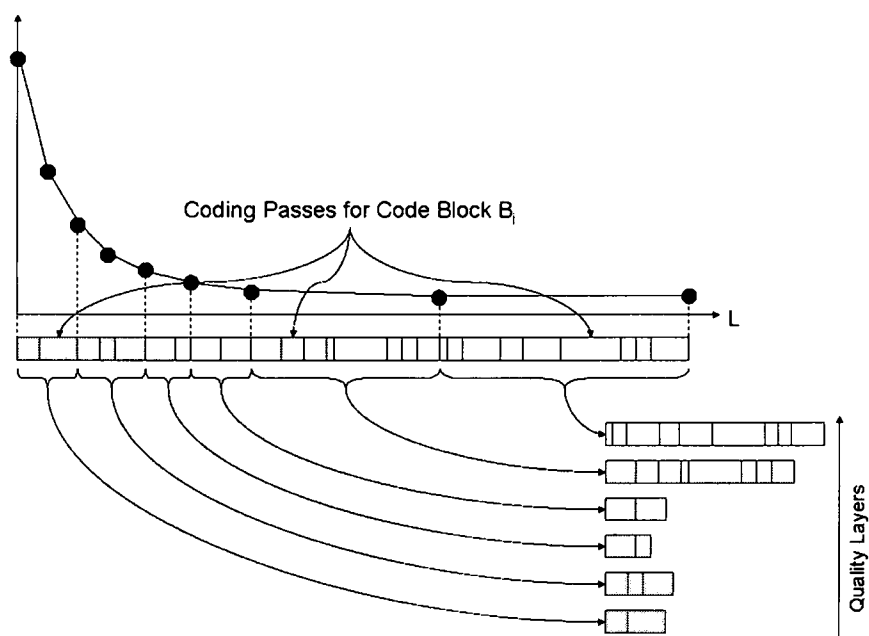


Figure 3.8: Rate-Distortion Information to Quality Layers

length of the code stream for the precinct packet [15]. This information is included for each sub-band within the precinct. For more detailed information on packet header, see [15] or [1].

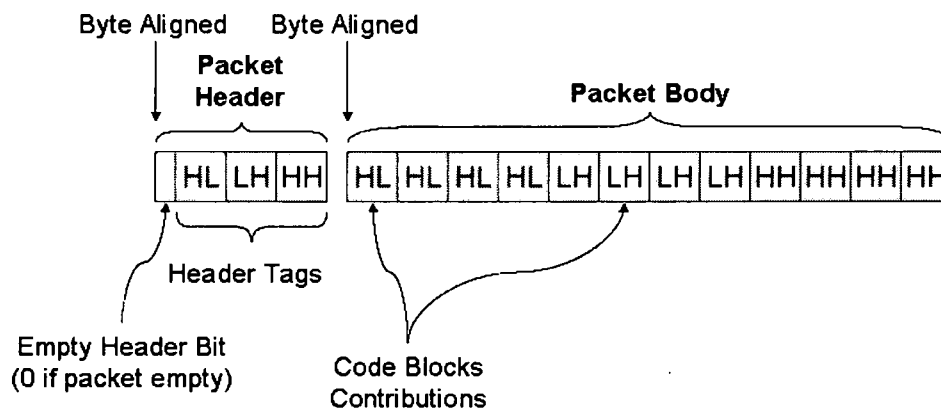


Figure 3.9: JPEG 2000 Packet Construction

CHAPTER 4

The MQ Coder

4.1 Introduction

Arithmetic coding was first developed by Elias, Rissanen, and Pasco[10]. The first incarnations of arithmetic encoders were impractical because they relied on ever increasing mathematical precision. The technique which arithmetic coding uses is similar to a guessing game where a probabilistic model of the information being encoded *guesses* what the next symbol will be. The encoder then encodes if the guess was correct or not. This section will begin with a discussion of arithmetic coding followed by the JPEG 2000 specific incarnation, the MQ-coder.

4.2 Arithmetic Coding

Arithmetic coding uses a probabilistic model to predict the *most probable symbol*. This model uses the symbol with the largest *interval* to determine the most probable symbol. Periodically, the intervals of each symbol are updated given the previously encoded symbols. Before any symbols are encoded, the total span of the model is from 0 to 1. As each symbol is encoded, the total span is reduced to the sub-interval of the previously encoded symbol. Because the overall interval is decreased, to accurately represent the new intervals, the precision of the intervals increases. Thus,

theoretically, this type of arithmetic encoding requires infinite precision arithmetic [10][1].

4.2.1 Arithmetic Coding Example

This example encodes the a source which produces three possible symbols, A, B, and C. Some information about the source is known before encoding begins. This example recalculates the probabilities after each symbol is known but the initial probabilities are set as $P_A = 0.4$, $P_B = 0.42$, and $P_C = 0.18$. Please reference this example's accompanying graphical representation in Figure 4.1. This example encodes the string of symbols, B, A, A, B, C into a binary output string. Initially, the total interval spans $[0, 1.0)$, with the sub-interval of each symbol equaling the probability of each symbol. The first symbol 'B' is encoded. 'B' currently has the largest sub-interval and is therefore the most probable symbol. The current interval for 'B' spans the output interval of both 0 and 1, therefore, there is no code output. Before the next symbol is coded, span is changed to $[0.18, 0.6)$. The probabilities of each symbol is also recalculated, which increases the probability of 'B' and decreases the probabilities of both 'A' and 'C'. The probabilities of each symbol are adjusted based on a pre-determined probabilistic model. Note that the precision of the sub-intervals also increase. The next symbol 'A' is observed. In this case, the symbol is not the most probable symbol. However, it is the second most probable symbol. In this case, the interval is smaller which leads to a greater probability that an entire output interval will be spanned. This is exactly what occurs; the span of '10' is covered and is therefore output. It is instructive to look at what occur if the 2 alternative symbols are encoded. If the second symbol is 'B', no span is entirely

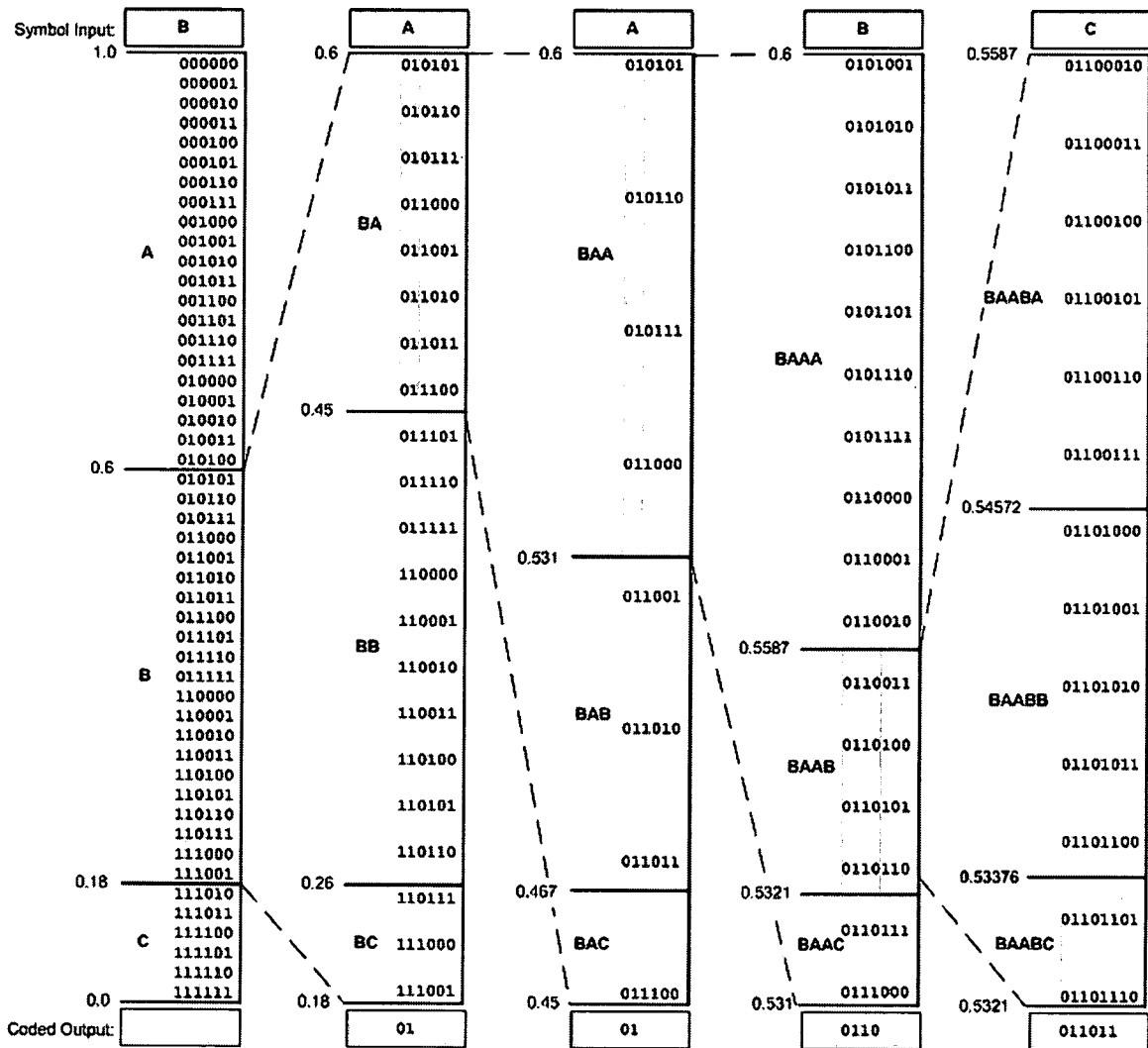


Figure 4.1: Arithmetic Encoding Example (Encoding Symbol String B, A, A, B, C)

covered and therefore nothing is output. In the case of 'C', the interval of '11' is entirely covered and therefore 2 bits are output. As the encoder continues to the third symbol, the probabilities are once again adjusted. The symbol which is 'A' is now also most probable symbol. In this case no span is entirely covered and therefore the encoder does not output any bits. The fourth symbol is 'B' who's span is reduced considerably. This symbol's span adds an additional 2 symbols which are '10'. Now,

the coded output is '0110'. The final symbol input is C. This symbol has always had the lowest probability, and thus the smallest span. In this case, 2 more bits are added to the coded output. The final coded output is 6 bits as the binary string '011011'. The former example attempts to illustrate several things. First, it is fairly easy to see how the selected probability for each symbol affects the coded output. In some cases, small changes in probability would have produced either more or less output bits. Second, the probabilities for each symbol is listed as coding progresses. As each symbol is coded, the need for more and more precise representation of the probability is required. The required precision for probability representation will soon grow to a point that no practical implementation can be realized. Therefore, a method to insure the probabilities require finite precision arithmetic is needed.

4.3 Binary Arithmetic Encoding

Binary arithmetic encoding as its name eludes only codes two different symbols, a '1' or a '0'. Similar to arithmetic encoding, in that each symbol has a corresponding probability and span, and is referred to as either the *most probable symbol* (MPS) or the *least probable symbol* (LPS). The MPS is defined as the subinterval $[A - Q, A)$ where Q is the current estimated probability, and the LPS is defined as the remainder $[0, A - Q)$. During each iteration of the encoder, the span A is updated using Equation 4.1. If the symbol being encoded is an MPS, then C (which represents the encoded information) is calculated Equation 4.2. Figure 4.2 represents the coding interval graphically [9].

$$A = A - AQ \quad (4.1)$$

$$C = C - AQ \quad (4.2)$$

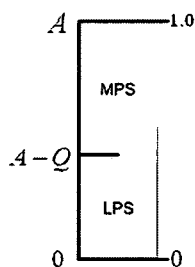


Figure 4.2: Representation of Coding Interval for Binary Arithmetic Coder

4.4 The MQ Coder

The MQ encoder is a context adaptive binary arithmetic encoder. It is the sibling of the QM-coder which was developed for the JBIG (Joint Bi-level Image Processing Group)[1]. The MQ-coder has several features that make it attractive as a processing efficient, and coding efficient encoding scheme. The MQ-coder due to its *renormalization* procedure allows it to represent probabilities with finite precision [15]. In addition, approximations of the probability spans allow it to be multiplication free.

Building on equations 4.1 and 4.2 from the previous section, it is observed that these equations require multiplication. This is because A can deviate significantly from 1. The MQ-Coder uses an approximation to eliminate this multiplication. To accomplish this, the bounds on A change from $[0, 1)$ to $[0.75, 1.5)$, allowing A to remain approximately 1. With this approximation, equations 4.1 and 4.2 become [9]

$$C = C + Q \quad (4.3)$$

$$A = A - Q \quad (4.4)$$

Due to approximating $A \approx 1$, an artifact arises allowing the MPS subinterval to become less than the LPS subinterval. When this occurs, the MPS and LPS intervals

are exchanged in a process known as *conditional exchange*. This can occur when A is small (i.e. close to 0.5) and Q is close to 0.5[9]. The conditional exchange essentially causes the MPS and LPS to be exchanged (i.e. $MPS \leftrightarrow LPS$)

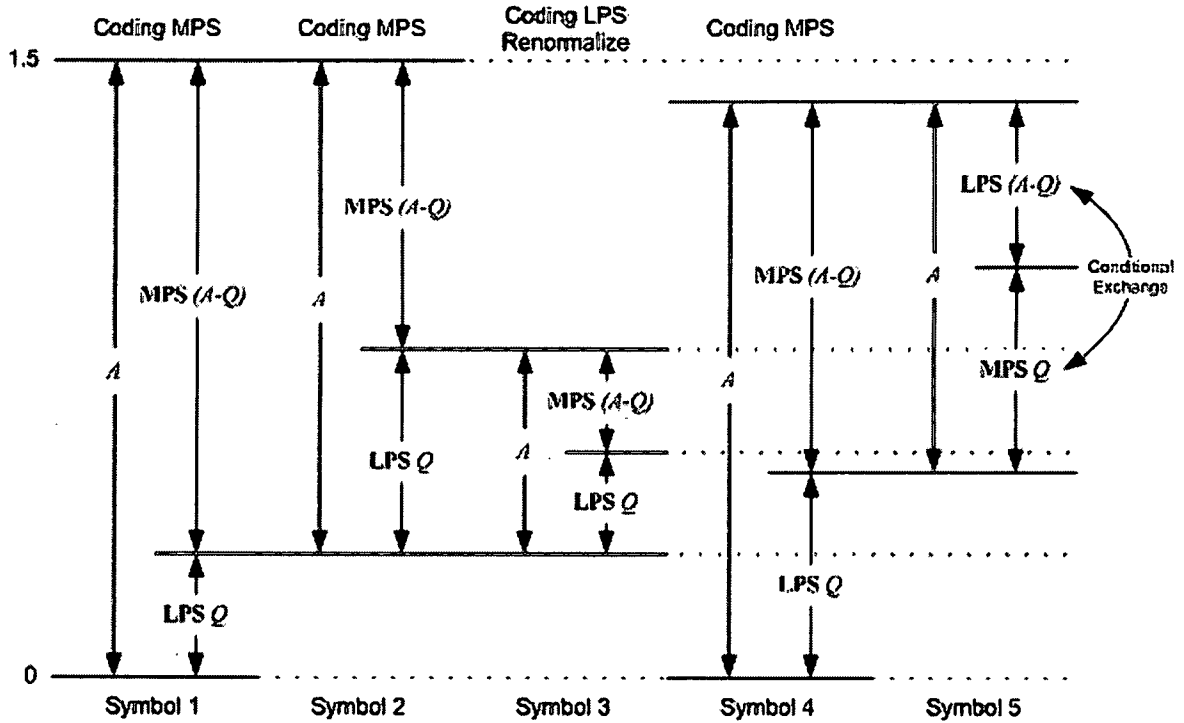


Figure 4.3: MQ Coder Coding Process Example

The upper range ,1.5, of A is guaranteed because the only operation ever performed on A is subtraction. The lower bound is guaranteed due to an operation known as *renormalization*. Renormalization is a simple process of consecutive left shift operations until A is greater than 0.75. This process only occurs when $A < 0.75$

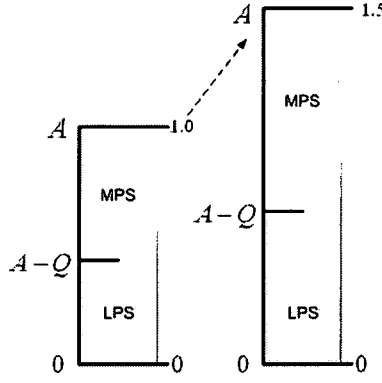


Figure 4.4: MQ-coder Coding Interval Representation.

4.4.1 Data Representation

The MQ-coder uses a 16-bit representation for A . To build upon the discussion in the previous section, A has the range of $[0.75, 1.5)$. Normalizing this to a 16-bit representation yields a maximum value of $2^{16} - 1 = 65,535$. The lower bound is then found as $2^{16} \frac{0.75}{1.5} = 32,768$. This is a convenient lower bound because $32,768 = 2^{15}$. Therefore, our bounds when normalized to the 16-bit representation are $[2^{15}, 2^{16} - 1) = [32,768, 65,535)$. C uses a 28-bit representation to store the actively encoded information along with the renormalized information. Figure 4.6 represents both A and C . This figure defines some special regions of the C register. These regions are discussed in more detail in Section 4.5.

4.4.2 Probability Estimation

The MQ-Coder's probability estimation is driven by a probability transition table (see Table C.5 in Appendix C) which adjusts the coding intervals each time an LPS is encoded, or when renormalization occurs[1]. Each transition in the table is

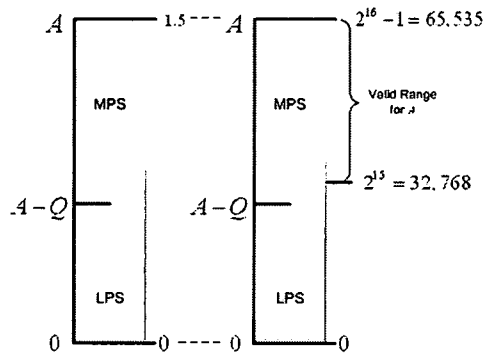


Figure 4.5: MQ Coder Interval Normalized to 16-bits

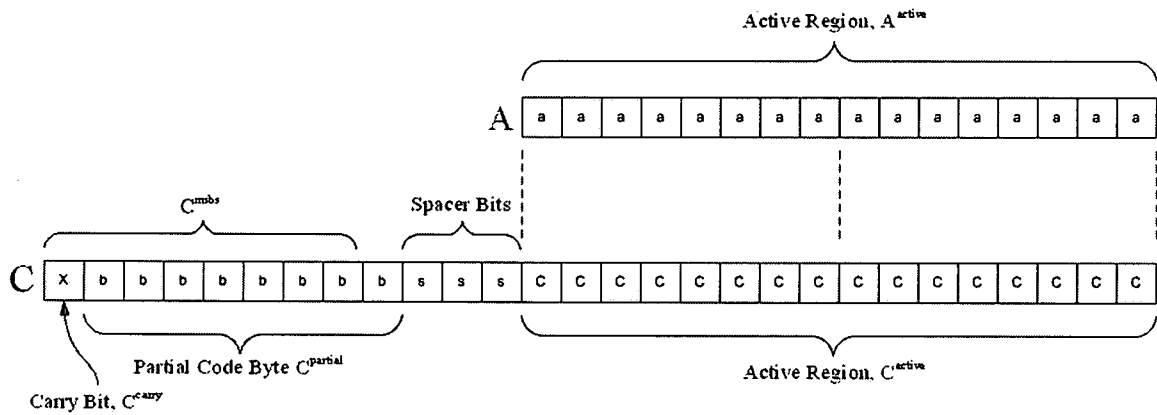


Figure 4.6: MQ Coder Data Representation/Registers

well defined and is based on whether an MPS or LPS is being encoded at the time the transition occurs. The probability estimate Q is a 16-bit representation of the current estimated MPS probability. The derivation of the probability state transition table is beyond the scope of this thesis. For more information, reference [15] or [11].

4.4.3 Context State

Recall from Section 3.1.1 that prior to encoding a bit, *context* information is calculated based on the surrounding pixels. This context information which lies between 0 and 18 is fed into the MQ coder. The MQ-coder tracks the probabilities for each of the context states *independently*[15]. Tracking the probability of each context independently, improves the coding efficiency because each of the different contexts will have a different estimated probability. Before encoding begins, each of the contexts are initialized to a specific value. These values are shown in Table C.4 on page 92. The majority of the contexts are initialized to $\Sigma = 0$. Using Table C.5 to look up the probability corresponding to $\Sigma = 0$, it is found that it equals $0x5601^5 = 22,017$. If this number is normalized back to the range of $[0, 1.5]$ it is found that the probability of an MPS starts out as

$$\frac{1.5}{2^{16} - 1} 22,017 = 0.504. \quad (4.5)$$

Therefore, the probability estimate the MQ-coder initially uses for the majority of contexts are initialized such that the probability of an MPS and LPS are approximately equal. Also in Table C.4 is s_κ which stores the current value (either 1 or 0) of the MPS. The MPS for each context is initialized to zero. This value is changed when a conditional exchange occurs[15].

4.5 MQ-Algorithm

This section discusses the general operation of the MQ-coder algorithm. Appendix C.2 contains a full pseudo code representation of the MQ coder beginning on page

⁵The 0x prefix in front of numbers represents the number is in base 16 (hexadecimal).

94. The information presented here is intended to be an aid in understanding this pseudo code. This section will reference specific variable names as they presented in the pseudo code and the MQ-coder registers in Figure 4.6. The MQ-coder algorithm presented here is a byte oriented (i.e. The output occurs one byte at a time). This method stores up one byte in the upper 8-bits of the C register (not including C^{carry}) and then transfers it to an external storage location. These upper 8-bits are represented in Figure 4.6 as either C^{msbs} or C^{partial} depending on if bit stuffing occurs (discussed in the following section). The MQ-coder must first be initialized so that the registers as well as the probabilities for each context state are initialized to the appropriate values. The registers are initialized to the values as shown in Figure C.2. Notice that $A \leftarrow 0x8000 = 2^{15}$, Therefore A will require renormalization when the first symbol is encoded. The 19 different context have their probability state Σ and MPS symbols initialized to the values shown in Table C.4.

Once initialization is complete, context values (κ) and symbols (x) are fed into the encoder. Based upon the current context's probability $Q(\Sigma_\kappa)$ the new span of A is calculated. Next, using the context's current MPS symbol s_κ , it is determined if the current symbol being encoded is an MPS or an LPS. If an MPS is encoded, the C register is recalculated a $C = C + Q(\Sigma_\kappa)$. Unless renormalization is required (i.e. $A < 2^{15}$) the value for A does not change and the, the probability Q for the current context κ . If an LPS is encoded, C is only modified by the mandatory renormalization procedure, unless a conditional exchange occurs (i.e. $A < Q(\Sigma_\kappa)$). The encoder's Put-Byte(\bar{T}, L) routine will output bytes from the \bar{T} buffer creating the code stream. The \bar{T} buffer is very important, as it allows the encoder to examine, and modify if necessary, the previously encoded byte, to determine if bit stuffing is required. A byte

is output from the C register to the \bar{T} buffer once the variable $\bar{t} = 0$. This variable is decremented by one each time a renormalization shift occurs. It is then reset to either 7 if bit stuffing occurs, or 8 if it does not.

4.6 Bit Stuffing

Bit stuffing performs two different functions. The first function allows for a less complicated and more efficient MQ-coder implementation. As a result of bit stuffing, the MQ-coder guarantees that no two consecutive output bytes are greater 0xFF8F [15]. JPEG 2000 leverages this gap produced by bit stuffing by specifying explicit delimiting markers that lie between 0xFF90 and 0xFFFF. This guarantees that the JPEG 2000 delimiting markers can not be confused with image data [15], [4]. Bit stuffing allows for a simpler implementation because it stops any carry bits which are produced during the arithmetic coding process from propagating any further than the previous output byte. Otherwise, the carry bits could propagate to and beyond the first byte output by the encoder, possibly requiring the modification of all previously output bytes. Bit stuffing occurs if the previous byte output by the encoder is 0xFF. When the previous byte is 0xFF, the encoder is setup such that only 7-bits are transferred to the output register before the byte is output. If the previous byte is not 0xFF, before the current byte is output, any carry located in C^{carry} (see Figure 4.6) is added to the previous output byte. After the propagation of any carry bit is complete, it must once again check to make sure bit stuffing is not required. This operation may be more clear by examining the pseudo code of this process in Figure C.3. The MQ-decoder will know that bit stuffing occurred when a 0xFF arises in the incoming encoded bit stream and handle it appropriately.

The spacer bits in the C register (see Figure 4.6) plays an important role in bounding the byte following a 0xFF in the output code-stream[15]. These 3 spacer bits allow for an upper non-inclusively bound of the output after an 0xFF occurs to $2^7 - 2^{7-S} = 2^7 - 2^{7-3} = 0x90$. While this is not obvious, examination of the pseudo-code in Figure C.3 helps clarify this process. In addition this bound plays into JPEG 2000's delimiting markers by guaranteeing the MQ-coder never outputs a two byte sequence greater than 0xFF8F.

4.6.1 Flushing

Once all of the desired bits are encoded, the MQ-coder must be "flushed." This is because there are bits in the C register (see Figure 4.6) which still contains information. Without flushing, this information would never be transferred into the encoded bit-stream. By the end of a code block, the MQ-coder will have adapted and therefore output bytes fairly infrequently, and therefore the C register may contain considerable information. The flushing routine in general examines \bar{t} to determine how many bits have been added to the C register since the last byte transfer. It then shifts the bits in the C register until they can be transfered to the code stream. After all bytes are moved out of the C register, a final byte transfer is completed to append the value in the \bar{T} buffer to the code-stream. As bytes are transferred out of the C register, bit stuffing protocol must be observed[15].

CHAPTER 5

Hardware Implementation

5.0.2 VHDL

VHDL (Very high speed integrated circuit **H**ardware **D**esign **L**anguage.) Was originally developed as a method for describing digital circuits and making timing analysis. This language was later adopted for a similar purpose of describing either the structure or functionality of a digital circuit and then "synthesizing" the description which can then be used in conjunction with an FPGA (Field Programmable Gate Array) or other similar programmable logic device to make a functioning digital circuit[5]. The syntax of VHDL is similar to that of the software programming language ADA [2]. Thus, VHDL is type safe and requires explicit conversions between differing data types.

Another popular hardware design language called Verilog. Its syntax is more like the C software language. Both languages accomplish the same goal, but for this project VHDL was chosen.

5.1 Approach

The following section discusses the approach used to develop the MQ Coder in hardware. It begins by discussing the advantages of prototyping in software for algorithm verification. This section continues by discussing the methods for verifying the

HDL design, and finally verification in hardware. Once the testing and verification framework are established, the actual hardware design is discussed in detail. Finally, the performance results are presented with supporting data.

5.1.1 JPEG 2000 Software Development

Inherently, software is easier to design, implement, test, and debug than hardware. Thus, it is advantageous, especially with a complex algorithm to first write a software implementation to help understand the algorithm, as well as solve any problems that may arise. Often issues will present themselves when prototyping in software that are overlooked during design or examination of an algorithm. Once aware of the problem, it is almost always easier to find and fix the problem in a software environment over a hardware environment. This is the approach taken when during the MQ coder hardware design.

5.1.2 VHDL Test Benching

A VHDL test bench is used extensively when initially designing and debugging the hardware. The test bench is used to verify that the output of the hardware version of the MQ coder matched the software version. The test bench allows for convenient automation of the testing process for hardware.

The VHDL test bench is an additional piece of VHDL code that allows the user to instantiate the hardware module that is being tested. It then allows for programmatic modification of the module's input ports and programmatic examination of the module's output ports. Another feature of the test bench is the ability to read and write text files, Which is convenient because it allows easy comparison between the hardware and software versions.

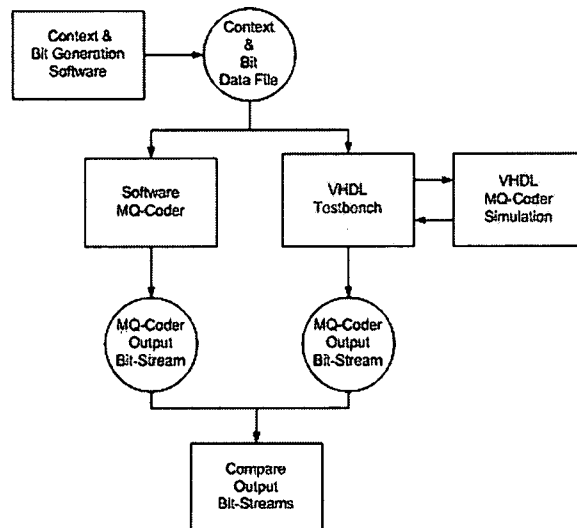


Figure 5.1: VHDL Test Bench Verification Approach

5.1.3 Hardware Testing

Once the HDL code is tested for functionality using the test bench, the HDL must be tested in hardware for it to be of any use. Therefore, a test setup is devised using a simple 8-bit USB 1.1 interface. In addition, a simple set of operations are devised, which allow the interface to initialize the encoder, transmit the context and bit data from the computer to the FPGA for processing and then request the retrieval of the encoded bytes from the hardware MQ-coder. First to verify the operation of the hardware MQ coder, a process comparable to the test bench is used. Using a random set of context and bit values, the output of the hardware MQ coder is compared to the output of the software version of the encoder. Again this process is automated allowing for quick verification using around one million context and bit value pairs. Once the hardware is verified using random data, the original software only JPEG 2000 encoder is modified allowing for the insertion of the hardware MQ coder into the

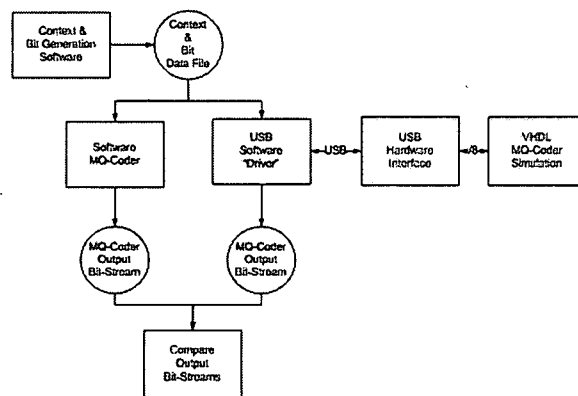


Figure 5.2: Hardware Verification

process. This method shows that the MQ coder correctly encoding the data provided by Tier I

Testing with Real Image Data

After verification with random data, the final test is to see if the hardware MQ-coder can encode a real image. This is accomplished using JPEG 2000 encoding software previously written and inserting the hardware MQ coder in place of the software MQ coder. The hardware MQ coder produces the same image compression results as the software MQ coder provides. Due to the USB 1.1 interface between the hardware and software, the image takes considerably longer to encode than the software only version. This however does not indicate that the hardware version is slower, only that a data bottleneck exists between the software and hardware. The actual performance of the hardware encoder verses the software is discussed in Section 5.2.7.

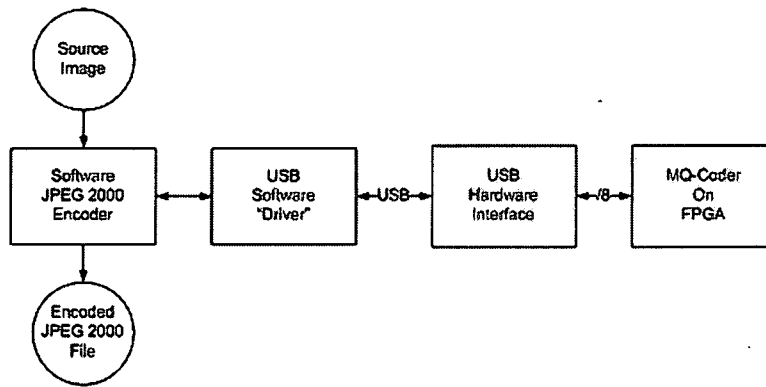


Figure 5.3: JPEG 2000 Hardware MQ-Coder Test Scheme

5.2 HDL Design

The MQ coder in this design uses 3 different modules, which correlate strongly with the C code prototype. The input and output of the MQ coder occurs in the top level module which contains interfacing logic as well as the state machines which drive the encoding process. The context state memory module stores the current probability state, and handles the probability state transition. The final module is the probability state transition lookup table, which is a simple but fairly large lookup table of probabilities and state transitions. These modules will be discussed in detail in the following three sections from the bottom up. The complete VHDL source code for the MQ-Coder is supplied in Appendix B.

5.2.1 Probability State Transition Table

The probability state transition table is based off of Table C.5. It is completely combinational logic, and therefore has no clock, and a simple interface as shown in Figure 5.5. The operation is simple and behaves as any lookup table. The current

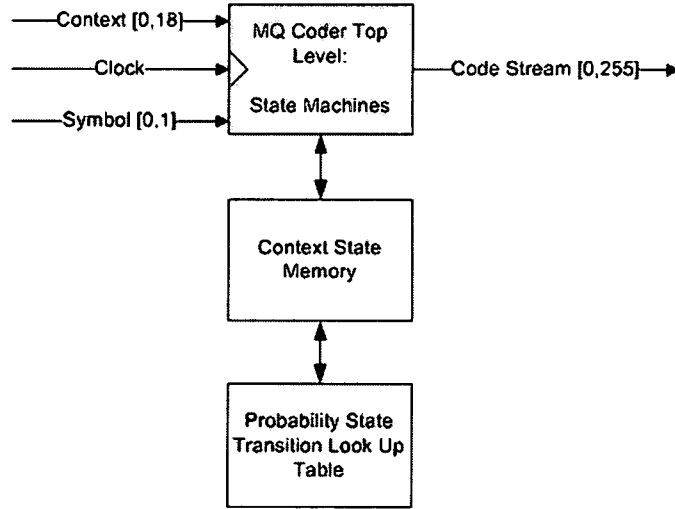


Figure 5.4: Overview of Hardware Architecture

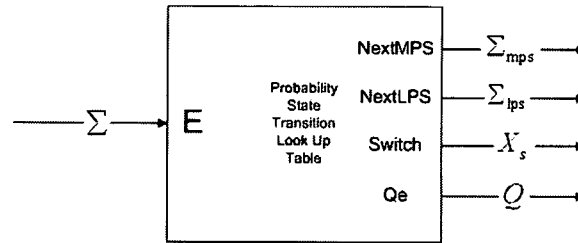


Figure 5.5: Probability State Transition Table Module Interface

probability state Σ is input into the module, which then causes the four outputs to transition to the appropriate state based on Table C.5.

5.2.2 Context State Memory

The context state memory is the next module up in the hierarchy. This module is of sequential design and therefore clocked. This module relates to Table C.4 as it is initialized to these values for Σ_κ and s_κ . Once initialized, it stores the current Σ_κ

and S_κ , which evolve for each of the 19 contexts as symbols are encoded. In addition, it performs the actual probability state of transition of Σ_κ based on whether the current symbol is an MPS or LPS. The interface for this module has six inputs and two outputs which interface with the state machines discussed in the next section. The interface to the Context State Memory module is represented graphically in figure 5.6. The first input shown is the clock, which is the same frequency as the state machines

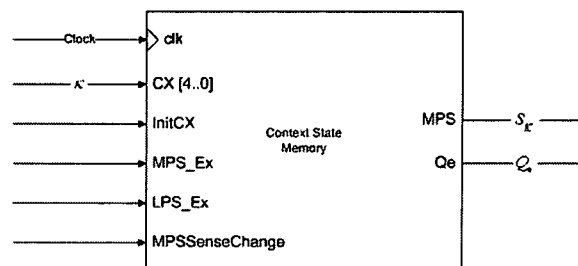


Figure 5.6: Context State Memory Module Interface

clock. The second input is the input context κ . This is the value which Tier-I of the JPEG 2000 encoder passes into the MQ coder. This controls which of the 19 context state memories will be output by this module. The third input (InitCX) causes the Context State Memory module to reset to the values in Table C.4. The next two inputs control the transition of probabilities for MPS and LPS respectively. These inputs are held at an active low state until either an MPS or LPS probability state transition is required. Depending on which transition is required, it will be held at an active high for one clock cycle. The final input is the MPSSenseChange which behaves in the same manner.

The memory in the module is set up as an array of STD_LOGIC_VECTORs, which are 6 bits wide. The STD_LOGIC_VECTOR array elements store the current probability state value Σ_{κ} for each of the 19 contexts (κ). As the context value is input from Tier I, the current value for Σ is passed to the Probability State Transition Module where the probability is looked up and passed back to this module which then passes it to the state machines without being registered in this module.

5.2.3 The State Machines

Up to this point the hardware descriptions have been "support modules" for the actual encoding process. This section explains the state machines which control the encoding process.

The MQ Coder has two intertwined state machines which act similar to sub-routines in software. The state machines are developed from the C prototype code by examining the logical branches and variables dependent on previous operations within the algorithm. Operations which are independent of one another are performed within the same clock cycle to eliminate as many clock cycles as possible.

To allow the state machines to behave as sub-routines, each state machine is aware the other's state. When the *primary* state machine moves to a particular state, it causes one of the *dependent* state machines to transition through its states. The primary state machine will then wait for the dependent state machine to reach a particular state before continuing. By analogy, this structure creates the equivalent of a blocking software function. Due to the MQ-coder algorithm's dependency on previously coded data, the blocking configuration is the required mode of operation.

Initialization State Machine

The initialization of the state machine takes place before each code block is coded. The initialization is handled in a separate state machine which initializes the MQ coder's registers to the values in Figure C.2 and the context state memory to the values shown in Table C.4. A state machine is chosen to perform this operation due to the need to strobe the `initCX` line for one clock cycle.

Encoder State Machine

The Encoder state machine is the heart of the encoder. The other modules and state machines are really support for this state machine. It performs the algorithm given in Figure C.1. Once the registers are initialized, and the state context memories are set to their initial values, this state machine can begin encoding context and symbol pairs. The context value produced by Tier I is passed to the `STD_LOGIC_VECTOR k`, along with the symbol (1 or 0) is passed to the `STD_LOGIC` value `x`. Once the context and symbol is set, encoding begins by strobing the `EncodeBit` input.

This state machine remains in the idle state (`E_ID`) until the `EncodeBit` input is raised high for at least one clock cycle. As the encoder moves out of the idle state it calculates the new value for `A` and determines if the current bit is an MPS or an LPS. At this point the algorithm splits into two separate paths. Both paths are similar, except for the probability state transitions, and that when coding an MPS, `A` must be checked to see if renormalization is required. When encoding an LPS, renormalization is always required, thus this check is not needed.

When renormalization occurs, for both the MPS and LPS paths, the value of \bar{t} ($\bar{t} = 0$) is examined to see if a byte is ready for removal from the C register. If a byte is ready, the Encoder state machine hands the work off to the Transfer Byte State Machine, discussed in the next section. The Transfer Byte state machine is activated when the Encoder state machine arrives into either state E_MPS_REN_1_WAIT_TB_2 or state E_LPS_REN_1_WAIT_TB_2. The Encoder state machine waits in this state until the Transfer Byte State Machine indicates it is finished by moving into the state TB_2. Once this occurs, the Encoding state machine moves to the next state as determined by the logic. Once either the MPS or LPS path is completed for a given symbol, the state machine returns to the idle state and raises the **EncoderReady** flag indicating that the encoder is ready to encode another context and symbol pair. Figures 5.7 is a state diagram of the Encoding state machine. Figure 5.8 is a state transition table of the Encoding state machine, which shows the state transitions and the necessary conditions to transition from state to state.

Transfer Byte State Machine (TBSM)

The TBSM handles moving the encoded bytes from the C register into the output byte buffer \bar{T} (\bar{T}). This also requires that the bit stuffing protocol is followed, as discussed in Chapter 4. Bit stuffing prevents the encoder from producing consecutive bits greater than 0xFF8F. The output of the TBSM is held in the byte buffer \bar{T} . The actual output data is handled by a small piece of helper logic. This helper logic is activate by strobing **PutByte** for one clock. This logic moves the output byte held in \bar{T} into the **ByteOut** output buffer. This logic also ensures that the first output from the encoder does not result in the **ByteReady** flag from being raised. The **ByteReady** flag must not be raised until the second byte is output from \bar{T} . If the **ByteReady**

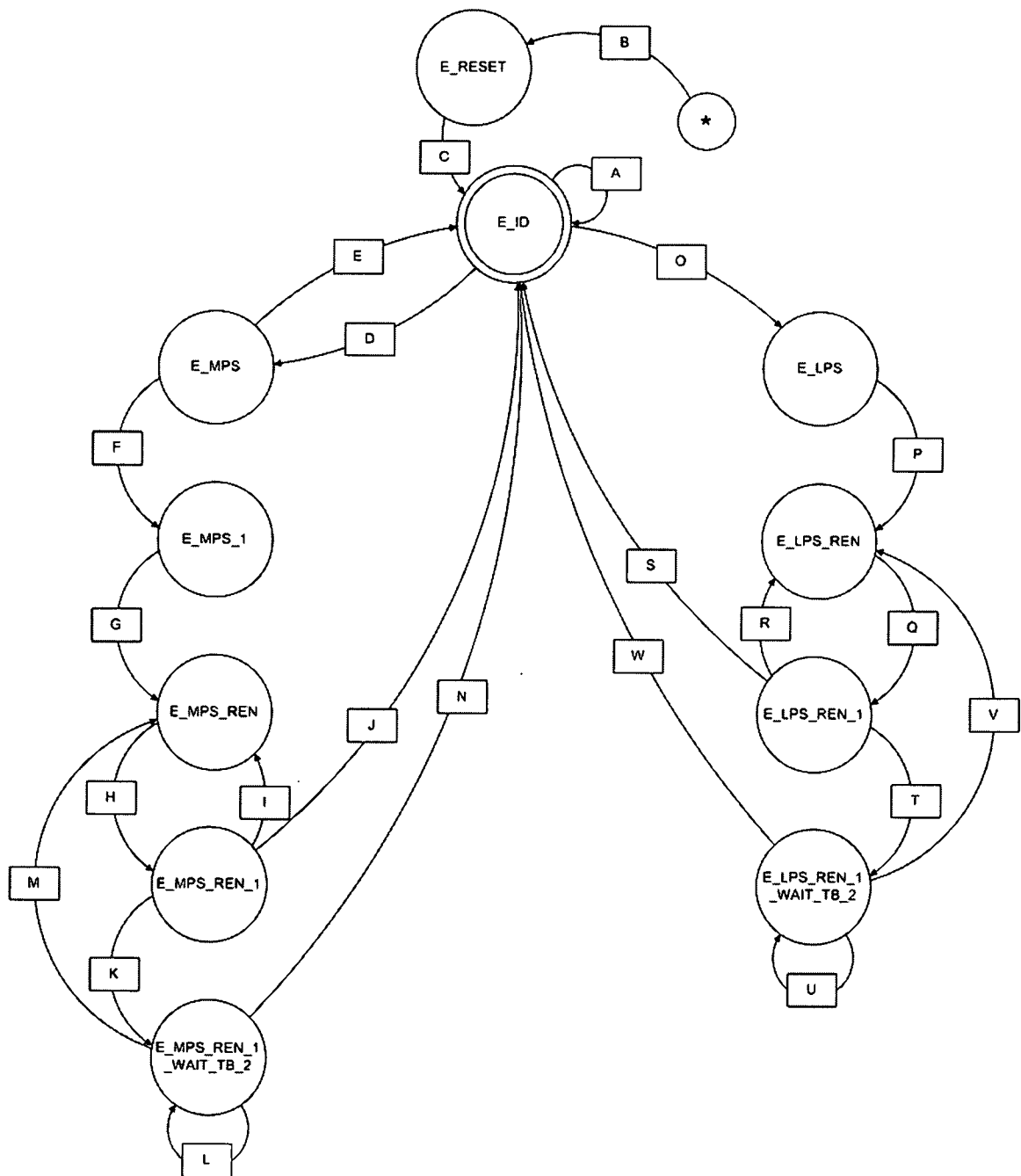


Figure 5.7: Encode State Machine

MQ Coder - Encoder (E) State Transition Table				
Current State	Condition to move to next state	Operation(s) Performed in Current State	Next State	Transition to Next State
*	rst = '1'		E_RESET	B
E_RESET	*	TB <= TB_ID FL <= FL_ID E <= E_ID mps_ex <= '0' lps_ex <= '0' EncoderReady <= '1'	E_ID	C
E_ID	EncodeBit != '1'	x_reg <= x k_reg <= k EncoderReady <= '1'	E_ID	A
	EncodeBit = '1' And x_reg = mps	A <= A - pbar EncoderReady <= '0'	E_MPS	D
	EncodeBit = '1' And x_reg != mps	A <= A - pbar EncoderReady <= '0'	E_LPS	O
E_MPS	A >= 2 ¹⁵	C <= C + pbar EncoderReady <= '1'	E_ID	E
	A < 2 ¹⁵		E_MPS_1	F
E_MPS_1	A < pbar	A <= pbar mps_ex <= '1'	E_MPS_REN	G
	A >= pbar	C <= C + pbar mps_ex <= '1'		
E_MPS_REN	*	A = A << 1 C = C << 1 t_bar <= t_bar - 1 mps_ex <= '0'	E_MPS_REN_1	H
E_MPS_REN_1	t_bar = 0		E_MPS_REN_1_WAIT_TB_2	K
	t_bar != 0 AND A >= 2 ¹⁵	EncoderReady <= '1'	E_ID	J
	t_bar != 0 AND A < 2 ¹⁵		E_MPS_REN	I
E_MPS_REN_1_WAIT_TB_2	TB != TB_2		E_MPS_REN_1_WAIT_TB_2	L
	A < 2 ¹⁵		E_MPS_REN	M
	A >= 2 ¹⁵	EncoderReady <= '1'	E_ID	N
E_LPS	A < pbar	lps_ex <= '1' mpssensechange <= '1' C <= C + pbar	E_LPS_REN	P
	A >= pbar	lps_ex <= '1' mpssensechange <= '1' A <= pbar		
E_LPS_REN	*	A <= A << 1 C <= C << 1 mpssensechange <= '0' t_bar <= t_bar - 1 lps_ex <= '0'	E_LPS_REN_1	Q
E_LPS_REN_1	t_bar != 0 AND A < 2 ¹⁵		E_LPS_REN	R
	t_bar != 0 AND A >= 2 ¹⁵ AND A < 2 ¹⁵	EncoderReady <= '1'	E_ID	S
	t_bar = 0		E_LPS_REN_1_WAIT_TB_2	T
E_LPS_REN_1_WAIT_TB_2	*		E_LPS_REN_1_WAIT_TB_2	U
	TB = TB_2 AND A < 2 ¹⁵		E_LPS_REN	V
	A >= 2 ¹⁵	EncoderReady <= '1'	E_ID	W

Figure 5.8: Encoding State Machine State Transition Table

flag is raised after the first byte is output, the first byte in the code stream would be erroneous.

As with the Encoder state machine, the TBSM (whose states are always prefixed by TB_) resides in the idle state (TB_ID) until its services are required (See Figure 5.9 for a graphical representation of this state machine). This state machine services both the Encoder state machine and the Flush state machine. It can be activated by any of four possible states. As the TBSM it moves out of the idle state, it determines if bit stuffing is required by checking if the output buffer Tbar is equal to 0xFF. At this point, the bit stuffing and non bit stuffing paths separate.

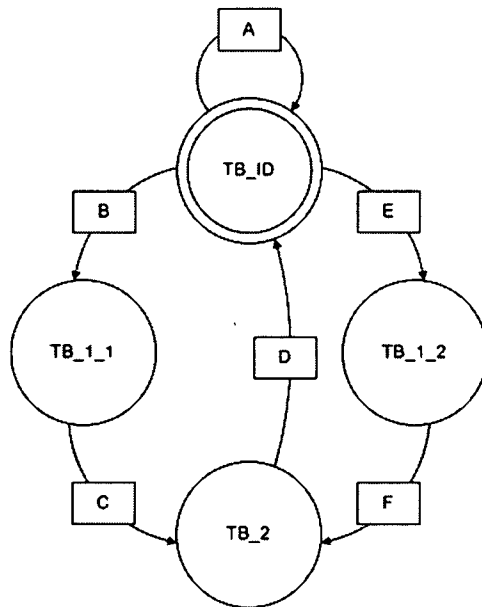


Figure 5.9: Transfer Byte State Machine

When bit stuffing occurs, the byte buffer Tbar is immediately output to make room for the new byte. The renormalization counter tbar is then set to 7 which ensures that the next byte does not exceed 0x8F. This also ensures that the next byte does not produce a carry bit, which could possibly propagate to bytes that have

already been removed from the MQ coder. If bit stuffing is not required, the carry bit C(27) is added to the byte buffer Tbar. Now that the byte buffer has possibly increased by 1, it must be checked again to see if bit stuffing is required. If bit stuffing is now required, a similar procedure as described earlier is followed. If bit stuffing is not required, tbar is set to 8, allowing the next output byte to reach the full value of 0xFF.

Once either the bit stuffing, or non bit stuffing paths are complete, they end at a common state of TB_2. When this state is reached, it indicates to either the Encode state machine, or the Flush state Machine that the TBSM has finished, and they can move out of their respective wait states. See Figure 5.10 for a complete listing of the states and the transition conditions.

Flushing State Machine

Once Encoding is finished as determined by Tier I of the JPEG 2000 encoder, some bytes remaining in the C register of the MQ coder. Thus, to complete the code stream, these bytes must be flushed. This state machine is shown in 5.11.

This state machine is fairly linear with the possibility of looping up to once through its states, depending on the initial value of tbar. As with the encoder state machine, the Flushing state machine also hands off the byte transferring process to the TBSM. This is implemented with the same intertwined trigger and wait state as implemented in the Encoder. One construct not used to this point is a barrel shifter. Up to now, all shifting operations have taken place one bit at a time. Depending on the available FPGA resources, this construct may take up more space than would be required for a sequential shifting operation, but guarantees the shifting is completed in one clock. If a particular implementation is low on FPGA resources, this could be a possible

MQ Coder - Transfer Byte (TB) State Transition Table				
Current State	Condition to move to next state	Operation(s) Performed In Current State	Next State	Transition to Next State
TB_ID	FL != FL_1_2 AND FL != FL_1_1 AND E != E_LPS_REN_1_WAIT_TB_2 AND E != E_MPS_REN_1_WAIT_TB_2		TB_ID	A
	FL = FL_1_2 OR FL = FL_1_1 OR E = E_LPS_REN_1_WAIT_TB_2 OR E = E_MPS_REN_1_WAIT_TB_2 AND Tbar = 0xFF	PutByte <= '1' Tbar_temp <= Tbar	TB_1_1	B
	FL = FL_1_2 OR FL = FL_1_1 OR E = E_LPS_REN_1_WAIT_TB_2 OR E = E_MPS_REN_1_WAIT_TB_2 AND Tbar != 0xFF	Tbar <= Tbar + C(27) C(27) <= '0' PutByte <= '1' Tbar_temp <= Tbar	B_1_2	E
TB_1_1	*	PutByte <= '0' Tbar <= C(27 DOWNT0 20) C(27 DOWNT0 20) <= "00000000" t_bar <= "00111"	TB_2	C
TB_1_2	Tbar = 0xFF	Tbar <= C(27 DOWNT0 20) C(27 DOWNT0 20) <= "00000000" t_bar <= "00111"	TB_2	F
	Tbar != 0xFF	Tbar <= C(26 DOWNT0 19) C(26 DOWNT0 19) <= "00000000" t_bar <= "01000"		
TB_2	*		TB_ID	D

Figure 5.10: Transfer Byte State Transition Table.

place to reduce the required resources. Figure 5.12 is the state transition table for the Flushing state machine.

5.2.4 Results

The hardware MQ coder performs as required, producing the same encoded bit-stream as the software prototype. These results as previously discussed are obtained

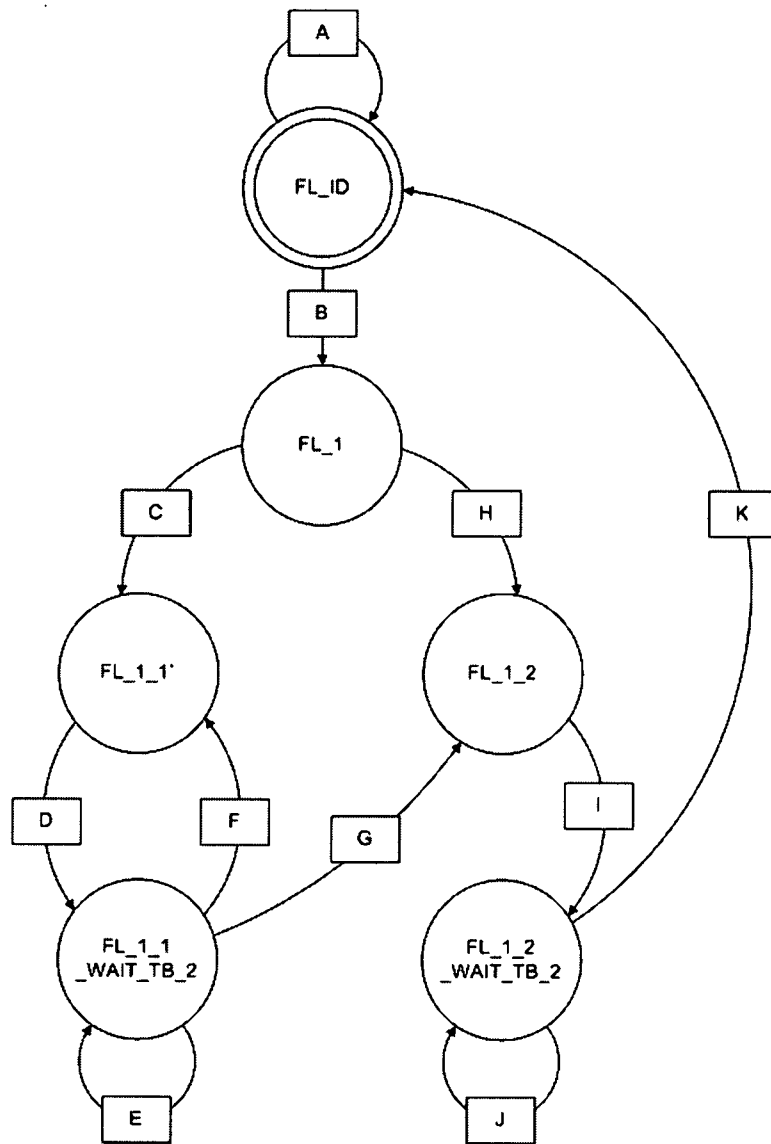


Figure 5.11: Encoder Flush State Machine

MQ Coder - Flush (FL) State Transition Table				
Current State	Condition to move to next state	Operation(s) Performed In Current State	Next State	Transition to Next State
FL_ID	flushMQ != '1'	nbits <= "01100" - t_bar;	FL_ID	A
	flushMQ = '1'	EncoderReady <= '0'	FL_1	B
FL_1	nbits > 0 AND nbits < 13	C <= C << t_bar	FL_1_1	C
	nbits >= 13		FL_1_2	H
FL_1_1	*	nbits <= nbits - t_bar	FL_1_1_WAIT_TB_2	D
FL_1_1_WAIT_TB_2	TB != TB_2	nbits <= nbits - t_bar	FL_1_1_WAIT_TB_2	E
	TB = TB_2 AND nbits > 0 AND nbits < 13	C <= C << t_bar	FL_1_1	F
	TB = TB_2 AND nbits > 13		FL_1_2	G
FL_1_2	*		FL_1_2_WAIT_TB_2	I
FL_1_2_WAIT_TB_2	TB != TB_2		FL_1_2_WAIT_TB_2	J
	TB = TB_2		FL_ID	K

Figure 5.12: Encoder Flush State Transition Table

by encoded the same set of data on both the software and hardware in which both produced identical outputs.

5.2.5 Performance

After the functionality of the MQ-coder is verified, performance metrics are required to determine if the effort and expense of transitioning the MQ coder to hardware is a wise decision.

The method chosen to measure the performance of the MQ coder in both hardware and software is to measure the average time each encoder requires to encode one symbol. In the software, a high performance timer is used to measure the time required to encode each symbol. In hardware, the number of clocks required to encode one symbol is recorded. Counting clocks is an easy method of measure the performance of a digital design. It also has the added advantage of scaling easily

depending on the clock frequency being used for a particular application. Clock counting also allows the performance of a design to be determined in simulation rather than in actual hardware. This method however does assume the time required to transfer data into and out of the hardware domain to be zero. The clock counting for in this instance is accomplished using a VHDL test bench.

After the data is recorded using the test bench and high performance timers, the average of each is taken to determine the overall performance of the two systems. It is found that the average symbol encoding time for the software running on a Pentium M 1.6 GHz processor is $1.26\mu\text{s}$ while the hardware is able to encode one symbol in an average of 30.14ns at a clock frequency of 100MHz. This is a speedup of over 41 times. The 30.14ns required to encoded the symbol translates to an average of just over three clocks per symbol.

In an effort to understand these numbers better, the encoding time per bit is graphed for each of the encoders. Notice that on each of these graphs that the spikes which take a larger amount of time decrease as the encoder encodes more data. In reference to Figure 5.14, there are three dominate clock counts 3, 6, and 9. The smallest number of clock cycles (3) occurs when a symbol is encoded and neither renormalization or a byte needs to be output. When a symbol requires 6 clock cycles, this represents a renormalization occurring. The largest number of clocks (9) occurs when a byte is output from the encoder. The hardware graph in Figure 5.14 clearly show the MQ coder adapting to the data and output fewer and fewer bytes in relation to the number of symbols encoded.

The distinctions between the different encoding processes are not as clearly defined in the software graph in Figure 5.13 as they are in the hardware graph. It is however

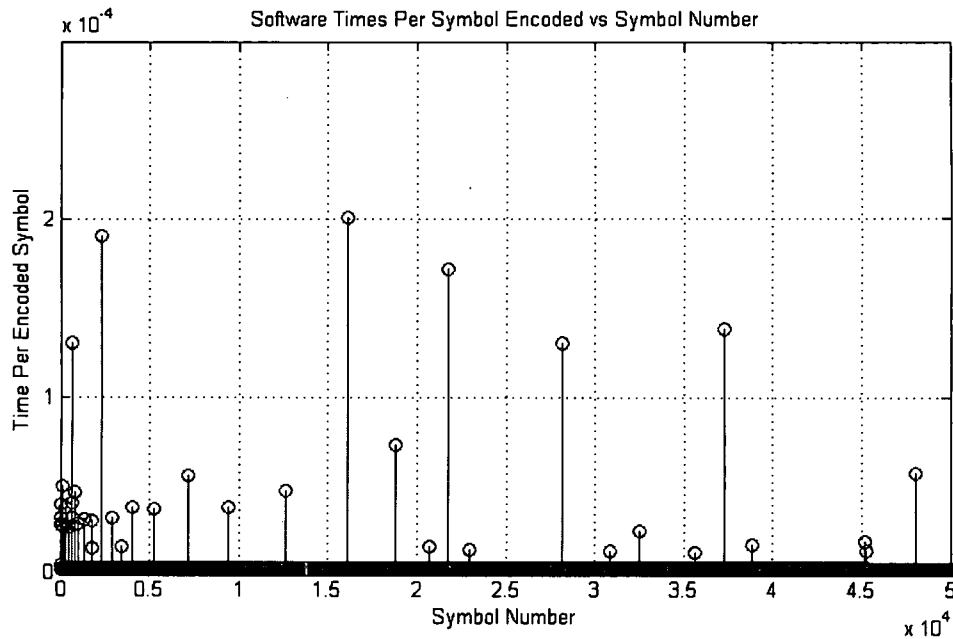


Figure 5.13: Software Encoding Times Per Symbol Encoded

apparent that a correlation between the two processes exists. Figure 5.15 shows both the software and hardware for closer examination of the correlation between the two encoders. It is clear that both encoders are correlated with respect to the relative amount of time required to encode a relative symbol. In other words, the encoding time required "spikes" as the same context and symbol pair are encoded.

5.2.6 Design Tradeoffs

A few design tradeoffs were made during the design of the hardware MQ coder. For example, if one state machine is used over the intertwining of 2 state machines, a small amount of speedup may occur. This speedup at most would only be one clock, during the transition between the two state machines. However, by performing the hand off of processing, the transfer byte logic is only required once rather than

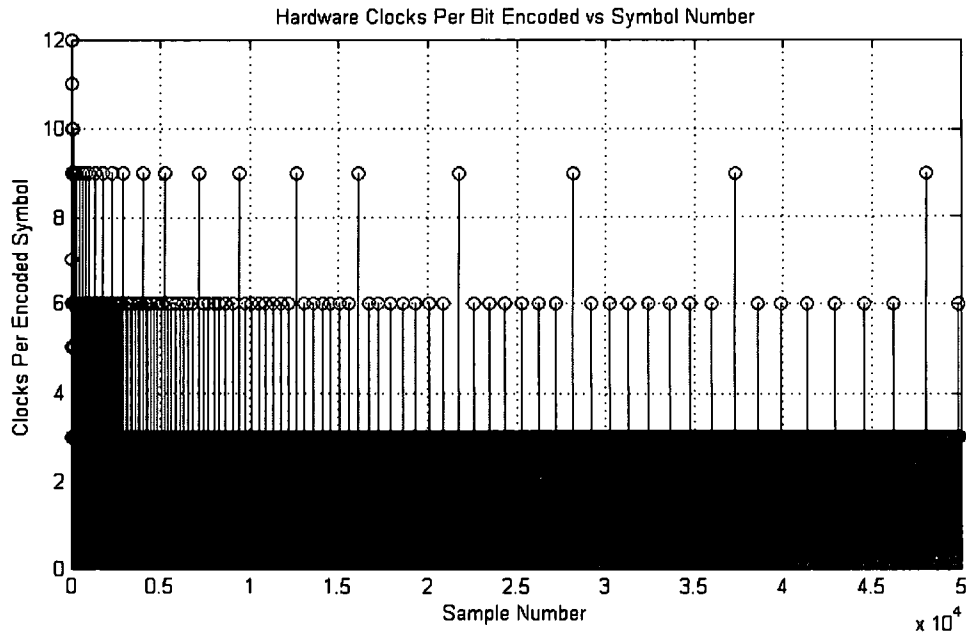


Figure 5.14: Hardware Clocks Per Symbol Encoded

four times, (twice for the Encoder State Machine, and twice for the Flush State Machine) consuming more FPGA fabric. This tradeoff is worthwhile because only a small fraction of the symbols that are encoded will result in a byte being output. Of the 50,000 context and symbol pairs encoded in the graphs above, only about 0.05 percent of the time, is the Transfer Byte State Machine activated. The accounts for about 0.05 percent of the total clock cycles. This percentage will very depending on the data being encoded.

5.2.7 Speedup

As discussed in the last section, the hardware speeds up the average symbol encoding time by over 41 times with a clock frequency of 100MHz when compared to

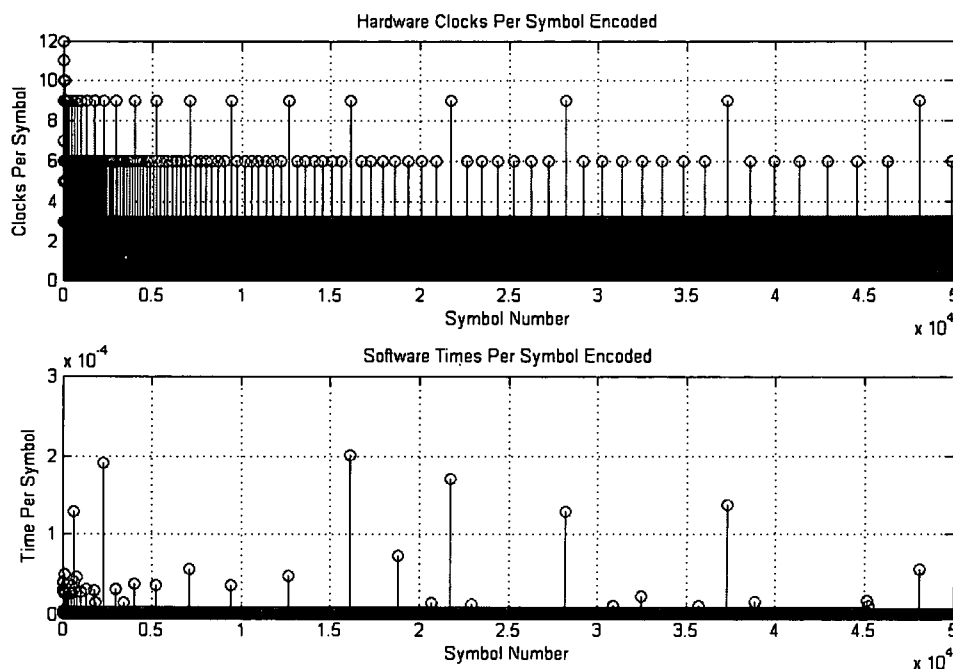


Figure 5.15: Software Hardware Comparison

the average symbol encoding time of a Pentium M 1.6GHz general purpose processor. A clock frequency of 100MHz should not be hard to obtain in advanced FPGAs. To further evaluate the maximum performance of this design, it is synthesized for an FPGA with a constraint of 200MHz as the minimum required clock frequency. This forced the synthesizer and fitter to "try harder" to meet the constraint. With this constraint in place, an advanced FPGA is able to achieve an estimated clock frequency of 185MHz where, the average symbol encoding time is 16.3ns; a speed up over the software of approximately 71 times.

CHAPTER 6

Conclusions & Future Work

6.1 Conclusion

In this thesis, the implementation of the Hardware MQ coder provides significant speedup over a software version using a similar algorithm. The design when synthesized consumes a reasonable amount of FPGA resources in relation to the complexity of the algorithm. A designer using this Hardware MQ coder in their system must decide if the increased performance is suitable for their application.

6.2 Future Work

The slowest portion of the Hardware MQ coder is the look up tables, which is due to their combinational nature. It is possible that the maximum clock frequency of the design could be increased propagation delay associated with this portion of the design could be reduced.

Other future areas of work, not relating directly to the hardware MQ coder include placing the Tier I encoding system into hardware. This would allow for significant parallel processing of code blocks, limiting the possible performance enhancements primarily to FPGA resources and the data bandwidth which feeds the Tier I hardware. Placing Tier I in hardware would also give the designer the opportunity to perform

distortion estimates in hardware, further reducing the burden on the JPEG 2000 software.

APPENDIX A

C Language MQ-Coder Implementation

MQCoder.cpp

```
// Written by The Center for Collaborative Computing
//University of Dayton Research Institute
// Written by Dave Mundy April 20, 2005.
// Copyright (c) 2005 - 2006 , Dave Mundy
// All rights reserved.

//MQCoder.cpp
//Header file: MQCoder.h
// Written by Dave Mundy April 20, 2005.

#include "book_mq_3.h"
#include "config.h"
#include <memory.h>
#include <stdlib.h>
#include <stdio.h>

unsigned int A; //MQ coder A register
unsigned long C; //MQ coder C register
int tbar; //MQ coder Shifts of C counter
int Tbar;
unsigned int pbar;
int L; //MQ coder Output Byte index
unsigned char *B = 0;
int BPmax = 0;
unsigned char CX;

//##### Register Definitions #####
//
//Register A:      1 a a a a a a a a a a a a a a       (16-bit)
//
//                | MSB of C   |
//Register C:      x b b b b b b b s s s c c c c c c c c c c c c c c c c c c
//                | | Part. Code | |    | |
//        Carry Bit---|          ---   |         Active Region of C     |
//                      |              |
//                      Spacer Bits ----|
//
//#####

//Probability State transition table for MQ-coder
//As per Table 2.1 - Taubman
PST pst_table[]=
{
    {&pst_table[1], &pst_table[1], 1, 0x5601, (float)0.475, 0},           //0
    {&pst_table[2], &pst_table[6], 0, 0x3401, (float)0.292, 1},          //1
    {&pst_table[3], &pst_table[9], 0, 0x1801, (float)0.132, 2},          //2
    {&pst_table[4], &pst_table[12], 0, 0x0AC1, (float)0.0593, 3},         //3
    {&pst_table[5], &pst_table[29], 0, 0x0521, (float)0.0283, 4},        //4
    {&pst_table[38], &pst_table[33], 0, 0x0221, (float)0.0117, 5},       //5
```

```

    {&pst_table[7], &pst_table[6], 1, 0x5601, (float)0.475, 6}, //6
    {&pst_table[8], &pst_table[14], 0, 0x5401, (float)0.463, 7}, //7
    {&pst_table[9], &pst_table[14], 0, 0x4801, (float)0.397, 8}, //8
    {&pst_table[10], &pst_table[14], 0, 0x3801, (float)0.309, 9}, //9
    {&pst_table[11], &pst_table[17], 0, 0x3001, (float)0.265, 10}, //10
    {&pst_table[12], &pst_table[18], 0, 0x2401, (float)0.199, 11}, //11
    {&pst_table[13], &pst_table[20], 0, 0x1C01, (float)0.155, 12}, //12
    {&pst_table[29], &pst_table[21], 0, 0x1601, (float)0.121, 13}, //13
    {&pst_table[15], &pst_table[14], 1, 0x5601, (float)0.475, 14}, //14
    {&pst_table[16], &pst_table[14], 0, 0x5401, (float)0.463, 15}, //15
    {&pst_table[17], &pst_table[15], 0, 0x5101, (float)0.447, 16}, //16
    {&pst_table[18], &pst_table[16], 0, 0x4801, (float)0.397, 17}, //17
    {&pst_table[19], &pst_table[17], 0, 0x3801, (float)0.309, 18}, //18
    {&pst_table[20], &pst_table[18], 0, 0x3401, (float)0.292, 19}, //19
    {&pst_table[21], &pst_table[19], 0, 0x3001, (float)0.265, 20}, //20
    {&pst_table[22], &pst_table[19], 0, 0x2801, (float)0.221, 21}, //21
    {&pst_table[23], &pst_table[20], 0, 0x2401, (float)0.199, 22}, //22
    {&pst_table[24], &pst_table[21], 0, 0x2201, (float)0.188, 23}, //23
    {&pst_table[25], &pst_table[22], 0, 0x1C01, (float)0.155, 24}, //24
    {&pst_table[26], &pst_table[23], 0, 0x1801, (float)0.132, 25}, //25
    {&pst_table[27], &pst_table[24], 0, 0x1601, (float)0.121, 26}, //26
    {&pst_table[28], &pst_table[25], 0, 0x1401, (float)0.110, 27}, //27
    {&pst_table[29], &pst_table[26], 0, 0x1201, (float)0.0993, 28}, //28
    {&pst_table[30], &pst_table[27], 0, 0x1101, (float)0.0938, 29}, //29
    {&pst_table[31], &pst_table[28], 0, 0x0AC1, (float)0.0593, 30}, //30
    {&pst_table[32], &pst_table[29], 0, 0x09C1, (float)0.0499, 31}, //31
    {&pst_table[33], &pst_table[30], 0, 0x08A1, (float)0.0476, 32}, //32
    {&pst_table[34], &pst_table[31], 0, 0x0521, (float)0.0283, 33}, //33
    {&pst_table[35], &pst_table[32], 0, 0x0441, (float)0.0235, 34}, //34
    {&pst_table[36], &pst_table[33], 0, 0x02A1, (float)0.0145, 35}, //35
    {&pst_table[37], &pst_table[34], 0, 0x0221, (float)0.0117, 36}, //36
    {&pst_table[38], &pst_table[35], 0, 0x0141, (float)0.00692, 37}, //37
    {&pst_table[39], &pst_table[36], 0, 0x0111, (float)0.00588, 38}, //38
    {&pst_table[40], &pst_table[37], 0, 0x0085, (float)0.00287, 39}, //39
    {&pst_table[41], &pst_table[38], 0, 0x0049, (float)0.00157, 40}, //40
    {&pst_table[42], &pst_table[39], 0, 0x0025, (float)0.000797, 41}, //41
    {&pst_table[43], &pst_table[40], 0, 0x0015, (float)0.000453, 42}, //42
    {&pst_table[44], &pst_table[41], 0, 0x0009, (float)0.000194, 43}, //43
    {&pst_table[45], &pst_table[42], 0, 0x0005, (float)0.000108, 44}, //44
    {&pst_table[45], &pst_table[43], 0, 0x0001, (float)0.000022, 45}, //45
    {&pst_table[46], &pst_table[46], 0, 0x5601, (float)0.475, 46}, //46
};

//MQ context state initialization table
//As per Table 12.1 -Taubman
MQCS MQCS_INIT_table[]=
{
    {&pst_table[4], 0, (float)0.0283, 0}, //0
    {&pst_table[0], 0, (float)0.475, 1}, //1
    {&pst_table[0], 0, (float)0.475, 2}, //2
    {&pst_table[0], 0, (float)0.475, 3}, //3

```



```

    {&pst_table[0], 0, (float)0.475, 4}, //4
    {&pst_table[0], 0, (float)0.475, 5}, //5
    {&pst_table[0], 0, (float)0.475, 6}, //6
    {&pst_table[0], 0, (float)0.475, 7}, //7
    {&pst_table[0], 0, (float)0.475, 8}, //8
    {&pst_table[3], 0, (float)0.0593, 9}, //9
    {&pst_table[0], 0, (float)0.475, 10}, //10
    {&pst_table[0], 0, (float)0.475, 11}, //11
    {&pst_table[0], 0, (float)0.475, 12}, //12
    {&pst_table[0], 0, (float)0.475, 13}, //13
    {&pst_table[0], 0, (float)0.475, 14}, //14
    {&pst_table[0], 0, (float)0.475, 15}, //15
    {&pst_table[0], 0, (float)0.475, 16}, //16
    {&pst_table[0], 0, (float)0.475, 17}, //17
    {&pst_table[46], 0, (float)0.475, 18}, //18
};

//MQ context state table.
//This table is initialized in MQEncode_init() and MQDecode_init()
MQCS MQCx[19];

//----- Input/Output Buffer -----
//Accesable functions:
//      void setIOBuffPtr(unsigned char *Buffer)
//      unsigned char *Buffer:  Pointer to the output buffer
//                               for the MQ-encoder, or the input
//                               buffer for the MQ-decoder.
void setIOBuffPtr(unsigned char *Buffer)
{
    B = Buffer;
}

int GetBuffLength(void)
{
    return L;
}
//----- End Input/Output Buffer -----

##### MQ Encoder #####
//Accesable functions:
//      void MQEncode_init()
//      void MQEncode(unsigned char D, unsigned char k)
//      unsigned char D:  The "decision" bit. The only allowable
//                        value is 1 or 0
//      unsigned char k:  The context state associate with D.
//      void FLUSHregister()
//
//Usage:
//      (1) Before any bits are encoded, first the Output buffer must
//      be set using the setIOBuffPtr(unsigned char *Buffer)
//      (2) Next, the MQEncode_init() method must be called.

```

```

//      (3) The bits can now be encoded using the
//      MQEncode(unsigned char D, unsigned char k) method.
//      (4) After all of the bits are encoded, the stream must be
//      terminated using the FLUSHregister() method.

//----- MQ Encoder -----
void MQEncode_init()
{
    A = 0x8000;
    C = 0;
    tbar = 12;
    Tbar = 0;
    L = -1; //avoid transferring Tbar to the byte buffer before it is first filled
    memcpy(MQCx, MQCS_INIT_table, 19 * sizeof(MQCS));
}

void PutByte()
{
    if(L >= 0)
    {
        *(B+L) = (unsigned char)Tbar;
    }
    L = L + 1;
}

void TransferByte()
{
    if(Tbar == 0xFF) //can't propagate any carry past Tbar; need bit stuff
    {
        PutByte();
        //Mask then shift
        Tbar = ((C & (unsigned long)SELECT_CMSB_MASK) >> CMSB_SHIFT);
        C = C & (unsigned long)CLEAR_CMSB_MASK;
        tbar = 7;
    }
    else //don't need to bit stuff
    {
        //propagate any carry bit from C into Tbar
        Tbar = Tbar + ((C & (unsigned long)SELECT_CCARRY_MASK) >> CCARRY_SHIFT);
        C = C & (unsigned long)CLEAR_CCARRY_MASK;
        PutByte();
        if(Tbar == 0xFF)
        {
            //mask then shift
            Tbar = ((C & (unsigned long)SELECT_CMSB_MASK) >> CMSB_SHIFT);
            C = (C & (unsigned long)CLEAR_CMSB_MASK); //clear cpartial
            tbar = 7;
        }
        else
        {
            //mask then shift

```

```

        Tbar = ((C & (unsigned long)SELECT_CPARTIAL_MASK) >> CPARTIAL_SHIFT);
        C = (C & (unsigned long)CLEAR_CPARTIAL_MASK); //clear cpartial
        tbar = 8;
    }
}
}

void FLUSHregister()
{
    int nbits;
    nbits = 27 - 15 - tbar;
    C = (1<<tbar) * C;
    while(nbits > 0)
    {
        TransferByte();
        nbits = nbits - tbar;
        C = (1<<tbar) * C;
    }
    TransferByte();
}

void MQEncode(unsigned char x, unsigned char k)
{
    pbar = MQCx[k].SigmaSubK->pbar;
    A = A - pbar;
    if(x == MQCx[k].sk)//coding an MPS
    {
        //No renormalizaion and hence no conditional exchange
        if(A >= (unsigned int)TWO_TO_THE_FIFTEEN)
        {
            C=C + pbar;
        }
        else
        {
            if(A < pbar) //conditional exchange
            {
                A = pbar;
            }
            else
            {
                C = C + pbar;
            }
            MQCx[k].SigmaSubK = MQCx[k].SigmaSubK->NMPS;
            do //perform renormalization shift
            {
                A = 2*A;
                C = 2*C;
                tbar = tbar - 1;
                if(tbar == 0)
                {

```

```

        TransferByte();
    }
    }while(A < (unsigned int)TWO_TO_THE_FIFTEEN);
}
}
else //coding an LPS, renormalization is inevitable
{
    if(A < pbar) //conditional exchange
    {
        C = C + pbar;
    }
    else
    {
        A = pbar;
    }
    MQCx[k].sk = MQCx[k].sk ^ MQCx[k].SigmaSubK->Xs;
    MQCx[k].SigmaSubK = MQCx[k].SigmaSubK->NLPS;
    do //perform renormalization shift
    {
        A = 2*A;
        C = 2*C;
        tbar = tbar - 1;
        if(tbar == 0)
        {
            TransferByte();
        }
    }while(A < (unsigned int)TWO_TO_THE_FIFTEEN);
}

}
//##### END MQ Encoder #####

```

APPENDIX B

VHDL Language MQ-Coder Implementation

MQ.vhd

--Copyright (C) 2006 Dave Mundy,
--University of Dayton Research Institute
--All Rights Reserved.

--File: MQ.VHD

library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

ENTITY MQ IS

PORT(k : IN STD_LOGIC_VECTOR (4 DOWNT0 0);
 x : IN STD_LOGIC;
 EncodeBit : IN STD_LOGIC;
 clk : IN STD_LOGIC;
 rst : IN STD_LOGIC;
 init : IN STD_LOGIC;
 flushMQ : IN STD_LOGIC;
 ReadByte : IN STD_LOGIC;
 ByteOut : OUT STD_LOGIC_VECTOR(7 DOWNT0 0);
 ByteReady : OUT STD_LOGIC;
 EncoderReady : OUT STD_LOGIC

);

END MQ;

ARCHITECTURE MQC OF MQ IS

----- state machine variable definitions -----

TYPE sm_E IS(E_Reset, E_ID, E_MPS, E_MPS_1
 ,E_MPS_REN, E_MPS_REN_1, E_MPS_REN_1_WAIT_TB_2
 , E_LPS, E_LPS_REN, E_LPS_REN_1, E_LPS_REN_1_WAIT_TB_2);

SIGNAL E : sm_E;

TYPE sm_FL IS(FL_ID, FL_1, FL_1_1, FL_1_1_WAIT_TB_2, FL_1_2, FL_1_2_WAIT_TB_2);

SIGNAL FL : sm_FL;

TYPE sm_TB IS(TB_ID, TB_1_1, TB_1_2, TB_2);

SIGNAL TB : sm_TB;

TYPE sm_INIT IS(INIT_ID, INIT_1);

SIGNAL MQ_INIT : sm_INIT;

----- END state machine variable definitions END -----

----- SIGNAL Declarations -----

SIGNAL t_bar : STD_LOGIC_VECTOR(4 DOWNT0 0);

SIGNAL Tbar : STD_LOGIC_VECTOR(7 DOWNT0 0);

SIGNAL A : STD_LOGIC_VECTOR(15 DOWNT0 0);

SIGNAL C : STD_LOGIC_VECTOR(27 DOWNT0 0);

SIGNAL initCX : STD_LOGIC;

```

SIGNAL mps_ex          : STD_LOGIC;
SIGNAL lps_ex          : STD_LOGIC;
SIGNAL mps              : STD_LOGIC;
SIGNAL mpssensechange  : STD_LOGIC;
SIGNAL pbar             : STD_LOGIC_VECTOR(15 DOWNT0 0);
SIGNAL nbits            : STD_LOGIC_VECTOR(4 DOWNT0 0);
SIGNAL FirstByte       : STD_LOGIC;
SIGNAL PutByte         : STD_LOGIC;
SIGNAL Tbar_temp       : STD_LOGIC_VECTOR(7 DOWNT0 0);
SIGNAL OutputByteReady : STD_LOGIC;
----- END SIGNAL Declarations END -----

COMPONENT MQ_PST_LUT IS
port(
    clk          : IN  STD_LOGIC;    --clock
    CX           : IN  STD_LOGIC_VECTOR(4 DOWNT0 0); --context input
    initCX       : IN  STD_LOGIC;    --initialize context tables
    MPS_Ex       : IN  STD_LOGIC;    --perform MPS Exchange on Current Context
    LPS_Ex       : IN  STD_LOGIC;    --perform LPS Exchange on Current Context
    MPSSenseChange : IN  STD_LOGIC;  --perform MPS Sense Change on Current Context
    MPS          : OUT STD_LOGIC;    --current MPS for current context
    Qe           : OUT STD_LOGIC_VECTOR(15 DOWNT0 0) --current probability state
);
END COMPONENT;
BEGIN

    DUT : mq_pst_lut
    PORT MAP (
        clk  => clk ,
        CX => k_reg,
        initCX => initcx,
        MPS_ex  => mps_ex ,
        LPS_ex  => lps_ex ,
        MPSSenseChange => mpssensechange ,
        MPS => mps,
        Qe => pbar  ) ;

    ByteReady <= OutputByteReady;

    PROCESS(clk, init, rst, EncodeBit)
    BEGIN
        IF(clk'EVENT AND clk = '1') THEN
            IF(rst = '1') THEN
                --initialize probability state transition look up table;
                --initialize all state machines
                E <= E_Reset; --[B]
            ELSE
                ----- state machine for MQEncode(D, k) -----
                CASE E IS
                    WHEN E_Reset =>

```

```

        TB <= TB_ID;
        FL <= FL_ID;
        E <= E_ID; --[C]
        mps_ex <= '0';
        lps_ex <= '0';
        EncoderReady <= '1';
    WHEN E_ID =>
        IF(EncodeBit = '1') THEN
            A <= A - pbar;
            EncoderReady <= '0';
            IF(x_reg = mps) then --coding an MPS
E <= E_MPS; --[D]
        ELSE
            --coding an LPS
            E <= E_LPS; --[M]
        END IF;

        ELSE
            x_reg <= x;
            k_reg <= k;
            EncoderReady <= '1';
            E <= E_ID; --[A]
        END IF;
    WHEN E_MPS =>
        IF(A >= "1000000000000000") THEN
            C <= C + pbar;
            EncoderReady <= '1';
            E <= E_ID;
        ELSE
            E <= E_MPS_1;
        END IF;
    WHEN E_MPS_1 =>
        IF(A < pbar ) THEN --conditional exchange
            A <= pbar;
            mps_ex <= '1';
            E <= E_MPS_REN;
        ELSE
            C <= C + pbar;
            mps_ex <= '1';
            E <= E_MPS_REN;
        END IF;
    WHEN E_MPS_REN => --Perform Renormalization Shifts
        A <= TO_STDLOGICVECTOR(to_bitvector(A) sll 1);
        C <= TO_STDLOGICVECTOR(to_bitvector(C) sll 1);
        t_bar <= t_bar - 1;
        mps_ex <= '0';
        E <= E_MPS_REN_1;
    WHEN E_MPS_REN_1 =>
        IF(t_bar = 0) THEN
            --Transfer Byte()
            E <= E_MPS_REN_1_WAIT_TB_2;
        ELSE
            IF(A < "1000000000000000") THEN--while a < 2^15

```



```

        E <= E_MPS_REN;
    ELSE
        EncoderReady <= '1';
        E <= E_ID;
    END IF;
END IF;
WHEN E_MPS_REN_1_WAIT_TB_2 =>
    IF(TB = TB_2) THEN
        IF(A < "1000000000000000") THEN--while a < 2^15
            E <= E_MPS_REN;
        ELSE
            EncoderReady <= '1';
            E <= E_ID;
        END IF;
    ELSE
        E <= E_MPS_REN_1_WAIT_TB_2;
    END IF;
WHEN E_LPS =>
    IF(A < pbar ) THEN
        lps_ex <= '1';
        mpssensechange <= '1';
        C <= C + pbar;
        E <= E_LPS_REN;
    ELSE
        lps_ex <= '1';
        mpssensechange <= '1';
        A <= pbar;
        E <= E_LPS_REN;
    END IF;
WHEN E_LPS_REN => --perform renorm
    A <= TO_STDLOGICVECTOR(to_bitvector(A) sll 1);
    C <= TO_STDLOGICVECTOR(to_bitvector(C) sll 1);
    mpssensechange <= '0';
    t_bar <= t_bar - 1;
    lps_ex <= '0';
    E <= E_LPS_REN_1;
WHEN E_LPS_REN_1 =>
    IF(t_bar = 0) THEN
        --Transfer Byte()
        E <= E_LPS_REN_1_WAIT_TB_2;
    ELSE
        IF(A < "1000000000000000") THEN--while a < 2^15
            E <= E_LPS_REN;
        ELSE
            EncoderReady <= '1';
            E <= E_ID;
        END IF;
    END IF;
WHEN E_LPS_REN_1_WAIT_TB_2 =>
    IF(TB = TB_2) THEN
        IF(A < "1000000000000000") THEN--while a < 2^15

```

```

        E <= E_LPS_REN;
    ELSE
        EncoderReady <= '1';
        E <= E_ID;
    END IF;
ELSE
    E <= E_LPS_REN_1_WAIT_TB_2;
END IF;
END CASE;
END IF;
----- END state machine for MQEncode(D, k) END -----

----- state machine for TransferByte(D, k) -----

CASE TB IS
    WHEN TB_ID =>
        IF(FL = FL_1_2 OR FL = FL_1_1
            OR E = E_LPS_REN_1_WAIT_TB_2
            OR E = E_MPS_REN_1_WAIT_TB_2) THEN
            IF(Tbar = "11111111") THEN --bit stuff
                PutByte <= '1';
                Tbar_temp <= Tbar;
                TB <= TB_1_1;
            ELSE -- no bit stuff
                Tbar <= Tbar + C(27);
                C(27) <= '0';
                PutByte <= '1';
                Tbar_temp <= Tbar;
                TB <= TB_1_2;
            END IF;
        ELSE
            TB <= TB_ID;
        END IF;
    WHEN TB_1_1 => --bit stuff
        PutByte <= '0';
        Tbar <= C(27 DOWNT0 20);
        C(27 DOWNT0 20) <= "00000000";
        t_bar <= "00111";
        TB <= TB_2; --END
    WHEN TB_1_2 => --no bit stuff
        PutByte <= '0';
        IF(Tbar = "11111111") THEN
            Tbar <= C(27 DOWNT0 20);
            C(27 DOWNT0 20) <= "00000000";
            t_bar <= "00111";
            TB <= TB_2; --END
        ELSE
            Tbar <= C(26 DOWNT0 19);
            C(26 DOWNT0 19) <= "00000000";
            t_bar <= "01000";
            TB <= TB_2; --END
        END IF;
    END CASE;

```

```

        END IF;
    WHEN TB_2 =>
        TB <= TB_ID;
    end case;

----- END state machine for TransferByte(D, k) END -----

----- State Machine for FLUSHregister() -----
CASE FL IS
    WHEN FL_ID =>
        IF(flushMQ = '1') THEN
            FL <= FL_1;
            EncoderReady <= '0';
        ELSE
            nbits <= "01100" - t_bar;
            FL <= FL_ID;
        END IF;
    WHEN FL_1 =>
        C <= TO_STDLOGICVECTOR(to_bitvector(C) sll conv_integer(t_bar));
        --nbits should never be greater than 12,
        --IF so, the number IS actually negative
        IF(nbits > "00000" AND nbits < "01101") THEN
            FL <= FL_1_1;
        ELSE
            FL <= FL_1_2;
        END IF;
    WHEN FL_1_1 =>
        --Transfer Byte()
        nbits <= nbits - t_bar;
        FL <= FL_1_1_WAIT_TB_2;
    WHEN FL_1_1_WAIT_TB_2 =>
        IF(TB = TB_2) THEN
            C <= TO_STDLOGICVECTOR(to_bitvector(C) SLL CONV_INTEGER(t_bar));
            IF(nbits > "00000" AND nbits < "01101") THEN
                FL <= FL_1_1;
            ELSE
                FL <= FL_1_2;
            END IF;
        ELSE
            nbits <= nbits - t_bar;
            FL <= FL_1_1_WAIT_TB_2;
        END IF;
    WHEN FL_1_2 =>
        --Transfer Byte()
        FL <= FL_1_2_WAIT_TB_2;
    WHEN FL_1_2_WAIT_TB_2 =>
        IF(TB = TB_2) THEN
            FL <= FL_ID;
        ELSE
            FL <= FL_1_2_WAIT_TB_2;

```

```

        END IF;
    END CASE;
----- END State Machine for FLUSHregister() END -----

CASE MQ_INIT IS
    WHEN INIT_ID =>
        IF(init = '1') THEN
            initCX <= '1';
            A <= "100000000000000000";
            C <= "00000000000000000000000000000000";
            t_bar <= "01100";
            Tbar <= "00000000";
            FirstByte <= '1';
            ByteOut <= "00000000";
            OutputByteReady <= '0';
            mpssensechange <= '0';
            PutByte <= '0';
            Tbar_temp <= "00000000";
            MQ_INIT <= INIT_1;
        ELSE
            MQ_INIT <= INIT_ID;
        END IF;
    WHEN INIT_1 =>
        initCX <= '0';
        MQ_INIT <= INIT_ID;
END CASE;

IF(PutByte = '1' AND FirstByte = '0') THEN
    OutputByteReady <= '1';
    ByteOut <= Tbar;
elsif(PutByte = '1' AND FirstByte = '1') THEN
    FirstByte <= '0';
END IF;

IF(OutputByteReady = '1' AND ReadByte = '1') THEN
    OutputByteReady <= '0';
END IF;

END IF; --End Synchronous Clock
END PROCESS;
END MQC;

```

MQ_PST.vhd

--Copyright (C) 2006 Dave Mundy,
--University of Dayton Research Institute
--All Rights Reserved.

--File: MQ_PST.VHD

```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;  
use ieee.std_logic_arith.all;
```

```
-----  
  
entity MQ_PST is  
port(  
    E           :    in  std_logic_vector (5 downto 0);  
    NextMPS     :    out std_logic_vector (5 downto 0);  
    NextLPS     :    out std_logic_vector (5 downto 0);  
    Switch      :    out  std_logic;  
    Qe          :    out std_logic_vector (15 downto 0)  
);  
end MQ_PST;
```

```
architecture PSTQe of MQ_PST is  
begin
```

```
    with E select  
    NextLPS<= conv_std_logic_vector(1,6) when conv_std_logic_vector(0,6) ,  
              conv_std_logic_vector(6,6) when conv_std_logic_vector(1,6) ,  
              conv_std_logic_vector(9,6) when conv_std_logic_vector(2,6) ,  
              conv_std_logic_vector(12,6) when conv_std_logic_vector(3,6) ,  
              conv_std_logic_vector(29,6) when conv_std_logic_vector(4,6) ,  
              conv_std_logic_vector(33,6) when conv_std_logic_vector(5,6) ,  
              conv_std_logic_vector(6,6) when conv_std_logic_vector(6,6) ,  
              conv_std_logic_vector(14,6) when conv_std_logic_vector(7,6) ,  
              conv_std_logic_vector(14,6) when conv_std_logic_vector(8,6) ,  
              conv_std_logic_vector(14,6) when conv_std_logic_vector(9,6) ,  
              conv_std_logic_vector(17,6) when conv_std_logic_vector(10,6) ,  
              conv_std_logic_vector(18,6) when conv_std_logic_vector(11,6) ,  
              conv_std_logic_vector(20,6) when conv_std_logic_vector(12,6) ,  
              conv_std_logic_vector(21,6) when conv_std_logic_vector(13,6) ,  
              conv_std_logic_vector(14,6) when conv_std_logic_vector(14,6) ,  
              conv_std_logic_vector(14,6) when conv_std_logic_vector(15,6) ,  
              conv_std_logic_vector(15,6) when conv_std_logic_vector(16,6) ,  
              conv_std_logic_vector(16,6) when conv_std_logic_vector(17,6) ,  
              conv_std_logic_vector(17,6) when conv_std_logic_vector(18,6) ,  
              conv_std_logic_vector(18,6) when conv_std_logic_vector(19,6) ,  
              conv_std_logic_vector(19,6) when conv_std_logic_vector(20,6) ,  
              conv_std_logic_vector(19,6) when conv_std_logic_vector(21,6) ,  
              conv_std_logic_vector(20,6) when conv_std_logic_vector(22,6) ,
```

```

conv_std_logic_vector(21,6) when conv_std_logic_vector(23,6) ,
conv_std_logic_vector(22,6) when conv_std_logic_vector(24,6) ,
conv_std_logic_vector(23,6) when conv_std_logic_vector(25,6) ,
conv_std_logic_vector(24,6) when conv_std_logic_vector(26,6) ,
conv_std_logic_vector(25,6) when conv_std_logic_vector(27,6) ,
conv_std_logic_vector(26,6) when conv_std_logic_vector(28,6) ,
conv_std_logic_vector(27,6) when conv_std_logic_vector(29,6) ,
conv_std_logic_vector(28,6) when conv_std_logic_vector(30,6) ,
conv_std_logic_vector(29,6) when conv_std_logic_vector(31,6) ,
conv_std_logic_vector(30,6) when conv_std_logic_vector(32,6) ,
conv_std_logic_vector(31,6) when conv_std_logic_vector(33,6) ,
conv_std_logic_vector(32,6) when conv_std_logic_vector(34,6) ,
conv_std_logic_vector(33,6) when conv_std_logic_vector(35,6) ,
conv_std_logic_vector(34,6) when conv_std_logic_vector(36,6) ,
conv_std_logic_vector(35,6) when conv_std_logic_vector(37,6) ,
conv_std_logic_vector(36,6) when conv_std_logic_vector(38,6) ,
conv_std_logic_vector(37,6) when conv_std_logic_vector(39,6) ,
conv_std_logic_vector(38,6) when conv_std_logic_vector(40,6) ,
conv_std_logic_vector(39,6) when conv_std_logic_vector(41,6) ,
conv_std_logic_vector(40,6) when conv_std_logic_vector(42,6) ,
conv_std_logic_vector(41,6) when conv_std_logic_vector(43,6) ,
conv_std_logic_vector(42,6) when conv_std_logic_vector(44,6) ,
conv_std_logic_vector(43,6) when conv_std_logic_vector(45,6) ,
conv_std_logic_vector(46,6) when conv_std_logic_vector(46,6) ,
conv_std_logic_vector(0,6) when others;

```

with E select

```

NextMPS<= conv_std_logic_vector(1,6) when conv_std_logic_vector(0,6),
conv_std_logic_vector(2,6) when conv_std_logic_vector(1,6),
conv_std_logic_vector(3,6) when conv_std_logic_vector(2,6),
conv_std_logic_vector(4,6) when conv_std_logic_vector(3,6),
conv_std_logic_vector(5,6) when conv_std_logic_vector(4,6),
conv_std_logic_vector(6,6) when conv_std_logic_vector(5,6),
conv_std_logic_vector(7,6) when conv_std_logic_vector(6,6),
conv_std_logic_vector(8,6) when conv_std_logic_vector(7,6),
conv_std_logic_vector(9,6) when conv_std_logic_vector(8,6),
conv_std_logic_vector(10,6) when conv_std_logic_vector(9,6),
conv_std_logic_vector(11,6) when conv_std_logic_vector(10,6),
conv_std_logic_vector(12,6) when conv_std_logic_vector(11,6),
conv_std_logic_vector(13,6) when conv_std_logic_vector(12,6),
conv_std_logic_vector(14,6) when conv_std_logic_vector(13,6),
conv_std_logic_vector(15,6) when conv_std_logic_vector(14,6),
conv_std_logic_vector(16,6) when conv_std_logic_vector(15,6),
conv_std_logic_vector(17,6) when conv_std_logic_vector(16,6),
conv_std_logic_vector(18,6) when conv_std_logic_vector(17,6),
conv_std_logic_vector(19,6) when conv_std_logic_vector(18,6),
conv_std_logic_vector(20,6) when conv_std_logic_vector(19,6),
conv_std_logic_vector(21,6) when conv_std_logic_vector(20,6),
conv_std_logic_vector(22,6) when conv_std_logic_vector(21,6),
conv_std_logic_vector(23,6) when conv_std_logic_vector(22,6),
conv_std_logic_vector(24,6) when conv_std_logic_vector(23,6),

```

```

conv_std_logic_vector(25,6) when conv_std_logic_vector(24,6),
conv_std_logic_vector(26,6) when conv_std_logic_vector(25,6),
conv_std_logic_vector(27,6) when conv_std_logic_vector(26,6),
conv_std_logic_vector(28,6) when conv_std_logic_vector(27,6),
conv_std_logic_vector(29,6) when conv_std_logic_vector(28,6),
conv_std_logic_vector(30,6) when conv_std_logic_vector(29,6),
conv_std_logic_vector(31,6) when conv_std_logic_vector(30,6),
conv_std_logic_vector(32,6) when conv_std_logic_vector(31,6),
conv_std_logic_vector(33,6) when conv_std_logic_vector(32,6),
conv_std_logic_vector(34,6) when conv_std_logic_vector(33,6),
conv_std_logic_vector(35,6) when conv_std_logic_vector(34,6),
conv_std_logic_vector(36,6) when conv_std_logic_vector(35,6),
conv_std_logic_vector(37,6) when conv_std_logic_vector(36,6),
conv_std_logic_vector(38,6) when conv_std_logic_vector(37,6),
conv_std_logic_vector(39,6) when conv_std_logic_vector(38,6),
conv_std_logic_vector(40,6) when conv_std_logic_vector(39,6),
conv_std_logic_vector(41,6) when conv_std_logic_vector(40,6),
conv_std_logic_vector(42,6) when conv_std_logic_vector(41,6),
conv_std_logic_vector(43,6) when conv_std_logic_vector(42,6),
conv_std_logic_vector(44,6) when conv_std_logic_vector(43,6),
conv_std_logic_vector(45,6) when conv_std_logic_vector(44,6),
conv_std_logic_vector(45,6) when conv_std_logic_vector(45,6),
conv_std_logic_vector(46,6) when conv_std_logic_vector(46,6),
conv_std_logic_vector(0,6) when others;

```

with E select

```

Switch <= '1' when conv_std_logic_vector(0,6) ,
          '0' when conv_std_logic_vector(1,6) ,
          '0' when conv_std_logic_vector(2,6) ,
          '0' when conv_std_logic_vector(3,6) ,
          '0' when conv_std_logic_vector(4,6) ,
          '0' when conv_std_logic_vector(5,6) ,
          '1' when conv_std_logic_vector(6,6) ,
          '0' when conv_std_logic_vector(7,6) ,
          '0' when conv_std_logic_vector(8,6) ,
          '0' when conv_std_logic_vector(9,6) ,
          '0' when conv_std_logic_vector(10,6) ,
          '0' when conv_std_logic_vector(11,6) ,
          '0' when conv_std_logic_vector(12,6) ,
          '0' when conv_std_logic_vector(13,6) ,
          '1' when conv_std_logic_vector(14,6) ,
          '0' when conv_std_logic_vector(15,6) ,
          '0' when conv_std_logic_vector(16,6) ,
          '0' when conv_std_logic_vector(17,6) ,
          '0' when conv_std_logic_vector(18,6) ,
          '0' when conv_std_logic_vector(19,6) ,
          '0' when conv_std_logic_vector(20,6) ,
          '0' when conv_std_logic_vector(21,6) ,
          '0' when conv_std_logic_vector(22,6) ,
          '0' when conv_std_logic_vector(23,6) ,
          '0' when conv_std_logic_vector(24,6) ,

```

```

'0' when conv_std_logic_vector(25,6) ,
'0' when conv_std_logic_vector(26,6) ,
'0' when conv_std_logic_vector(27,6) ,
'0' when conv_std_logic_vector(28,6) ,
'0' when conv_std_logic_vector(29,6) ,
'0' when conv_std_logic_vector(30,6) ,
'0' when conv_std_logic_vector(31,6) ,
'0' when conv_std_logic_vector(32,6) ,
'0' when conv_std_logic_vector(33,6) ,
'0' when conv_std_logic_vector(34,6) ,
'0' when conv_std_logic_vector(35,6) ,
'0' when conv_std_logic_vector(36,6) ,
'0' when conv_std_logic_vector(37,6) ,
'0' when conv_std_logic_vector(38,6) ,
'0' when conv_std_logic_vector(39,6) ,
'0' when conv_std_logic_vector(40,6) ,
'0' when conv_std_logic_vector(41,6) ,
'0' when conv_std_logic_vector(42,6) ,
'0' when conv_std_logic_vector(43,6) ,
'0' when conv_std_logic_vector(44,6) ,
'0' when conv_std_logic_vector(45,6) ,
'0' when conv_std_logic_vector(46,6) ,
'0' when others;

```

--Lookup Qe(E)

with E select

```

Qe <=
X"5601" when conv_std_logic_vector(0,6) ,
X"3401" when conv_std_logic_vector(1,6) ,
X"1801" when conv_std_logic_vector(2,6) ,
X"0AC1" when conv_std_logic_vector(3,6) ,
X"0521" when conv_std_logic_vector(4,6) ,
X"0221" when conv_std_logic_vector(5,6) ,
X"5601" when conv_std_logic_vector(6,6) ,
X"5401" when conv_std_logic_vector(7,6) ,
X"4801" when conv_std_logic_vector(8,6) ,
X"3801" when conv_std_logic_vector(9,6) ,
X"3001" when conv_std_logic_vector(10,6),
X"2401" when conv_std_logic_vector(11,6),
X"1C01" when conv_std_logic_vector(12,6),
X"1601" when conv_std_logic_vector(13,6),
X"5601" when conv_std_logic_vector(14,6),
X"5401" when conv_std_logic_vector(15,6),
X"5101" when conv_std_logic_vector(16,6),
X"4801" when conv_std_logic_vector(17,6),
X"3801" when conv_std_logic_vector(18,6),
X"3401" when conv_std_logic_vector(19,6),
X"3001" when conv_std_logic_vector(20,6),
X"2801" when conv_std_logic_vector(21,6),
X"2401" when conv_std_logic_vector(22,6),
X"2201" when conv_std_logic_vector(23,6),
X"1C01" when conv_std_logic_vector(24,6),

```



```
X"1801" when conv_std_logic_vector(25,6),
X"1601" when conv_std_logic_vector(26,6),
X"1401" when conv_std_logic_vector(27,6),
X"1201" when conv_std_logic_vector(28,6),
X"1101" when conv_std_logic_vector(29,6),
X"0AC1" when conv_std_logic_vector(30,6),
X"09C1" when conv_std_logic_vector(31,6),
X"08A1" when conv_std_logic_vector(32,6),
X"0521" when conv_std_logic_vector(33,6),
X"0441" when conv_std_logic_vector(34,6),
X"02A1" when conv_std_logic_vector(35,6),
X"0221" when conv_std_logic_vector(36,6),
X"0141" when conv_std_logic_vector(37,6),
X"0111" when conv_std_logic_vector(38,6),
X"0085" when conv_std_logic_vector(39,6),
X"0049" when conv_std_logic_vector(40,6),
X"0025" when conv_std_logic_vector(41,6),
X"0015" when conv_std_logic_vector(42,6),
X"0009" when conv_std_logic_vector(43,6),
X"0005" when conv_std_logic_vector(44,6),
X"0001" when conv_std_logic_vector(45,6),
X"5601" when conv_std_logic_vector(46,6),
X"0000" when others;
```

```
end PSTQe;
```

MQ_PST_LUT.vhd

--Copyright (C) 2006 Dave Mundy,
--University of Dayton Research Institute
--All Rights Reserved.

--File: MQ_PST_LUT.VHD

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity MQ_PST_LUT is
port(clk : in std_logic;
CX : in std_logic_vector (4 downto 0);
initCX : in std_logic;
MPS_Ex : in std_logic;
LPS_Ex : in std_logic;
MPSSenseChange : in std_logic;
MPS : out std_logic;
Qe : out std_logic_vector(15 downto 0)
);
end MQ_PST_LUT;

architecture PST_LUT of MQ_PST_LUT is

component MQ_PST is
port(
E : in std_logic_vector (5 downto 0);
NextMPS : out std_logic_vector (5 downto 0);
NextLPS : out std_logic_vector (5 downto 0);
Switch : out std_logic;
Qe : out std_logic_vector (15 downto 0)
);
end component;

signal RegE : std_logic_vector(5 downto 0);
signal RegNextMPS : std_logic_vector(5 downto 0);
signal RegNextLPS : std_logic_vector(5 downto 0);
signal RegSwitch : std_logic;
signal RegQe : std_logic_vector(15 downto 0);
signal RegCurCX : std_logic;

signal RegCX : std_logic_vector(5 downto 0);
signal RegMPS : std_logic;

signal SigE : std_logic_vector(5 downto 0);
signal SigNextMPS : std_logic_vector(5 downto 0);
signal SigNextLPS : std_logic_vector(5 downto 0);

```

signal SigSwitch      :      std_logic;
signal SigQe          :      std_logic_vector(15 downto 0);
signal SigMPS         :      std_logic;
signal ASDF           :      std_logic_vector(5 downto 0);

type IofCXMem is array (0 to 18) of std_logic_vector(5 downto 0);
signal CX_State: IofCXMem;
signal CX_Switch: std_logic_vector(18 downto 0);

begin

    PST : MQ_PST
    port map(
        E      =>      ASDF,
        NextMPS =>      SigNextMPS,
        NextLPS =>      SigNextLPS,
        Switch  =>      SigMPS,
        Qe      =>      Qe
    );

    ASDF <= CX_State(conv_integer(CX));
    process(clk,initCX)
    begin

        if(clk'event and clk = '1' ) then

            if(initCX = '1') then
                CX_State(0) <= conv_std_logic_vector(4,6);
                CX_State(1) <= conv_std_logic_vector(0,6);
                CX_State(2) <= conv_std_logic_vector(0,6);
                CX_State(3) <= conv_std_logic_vector(0,6);
                CX_State(4) <= conv_std_logic_vector(0,6);
                CX_State(5) <= conv_std_logic_vector(0,6);
                CX_State(6) <= conv_std_logic_vector(0,6);
                CX_State(7) <= conv_std_logic_vector(0,6);
                CX_State(8) <= conv_std_logic_vector(0,6);
                CX_State(9) <= conv_std_logic_vector(3,6);
                CX_State(10) <= conv_std_logic_vector(0,6);
                CX_State(11) <= conv_std_logic_vector(0,6);
                CX_State(12) <= conv_std_logic_vector(0,6);
                CX_State(13) <= conv_std_logic_vector(0,6);
                CX_State(14) <= conv_std_logic_vector(0,6);
                CX_State(15) <= conv_std_logic_vector(0,6);
                CX_State(16) <= conv_std_logic_vector(0,6);
                CX_State(17) <= conv_std_logic_vector(0,6);
                CX_State(18) <= conv_std_logic_vector(46,6);

                CX_Switch(0) <= '0';
                CX_Switch(1) <= '0';
                CX_Switch(2) <= '0';
                CX_Switch(3) <= '0';
            end if;
        end if;
    end process;

```

```

CX_Switch(4) <= '0';
CX_Switch(5) <= '0';
CX_Switch(6) <= '0';
CX_Switch(7) <= '0';
CX_Switch(8) <= '0';
CX_Switch(9) <= '0';
CX_Switch(10) <= '0';
CX_Switch(11) <= '0';
CX_Switch(12) <= '0';
CX_Switch(13) <= '0';
CX_Switch(14) <= '0';
CX_Switch(15) <= '0';
CX_Switch(16) <= '0';
CX_Switch(17) <= '0';
CX_Switch(18) <= '0';

else
  if(MPS_Ex = '1' and LPS_Ex = '0') then
    CX_State(conv_integer(CX)) <= SigNextMPS;
  end if;
  if(MPS_Ex = '0' and LPS_Ex = '1') then
    CX_State(conv_integer(CX)) <= SigNextLPS;
  end if;
  if(MPSSenseChange = '1') then
    CX_Switch(conv_integer(CX)) <= SigMPS
      xor CX_Switch(conv_integer(CX));
  end if;
end if;
end if;

end process;
MPS <= CX_Switch(conv_integer(CX));
end PST_LUT;

```

APPENDIX C

JPEG2000 and MQ Coder Reference

C.1 JPEG 2000 Reference

κ^{sig}	LL, LH blocks			HL blocks			HH blocks	
	κ^{h}	κ^{v}	κ^{d}	κ^{h}	κ^{v}	κ^{d}	κ^{d}	$\kappa^{\text{h}} + \kappa^{\text{v}}$
8	2	X	X	X	2	X	≥ 3	X
7	1	≥ 1	X	≥ 1	1	X	2	≥ 1
6	1	0	≥ 1	0	1	≥ 1	2	0
5	1	0	0	0	1	0	1	≥ 2
4	0	2	X	2	0	X	1	1
3	0	1	X	1	0	X	1	0
2	0	0	≥ 2	0	0	≥ 2	0	≥ 2
1	0	0	1	0	0	1	0	1
0	0	0	0	0	0	0	0	0

Table C.1: Context Values for κ^{sig} Used as Context or to Form Context for All Passes

$\overleftarrow{\sigma}$	κ^{sig}	κ^{mag}
0	0	15
0	> 0	16
1	X	17

Table C.2: Context Values for κ^{mag} Used in Magnitude Refinement Pass

$\bar{\chi}^h$	$\bar{\chi}^v$	κ^{sign}	χ^{flip}
1	1	14	1
1	0	13	1
1	-1	12	1
0	1	11	1
0	0	10	1
0	-1	11	-1
-1	1	12	-1
-1	0	13	-1
-1	-1	14	-1

Table C.3: Context Values for κ^{sign} and χ^{flip} Used in Sign Coding

C.2 MQ Coder Reference

κ	Context Name	Σ_{κ}	s_{κ}
0	κ^{sig}	4	0
1 – 8	κ^{sig}	0	0
9	κ^{run}	3	0
10 – 14	κ^{sign}	0	0
15 – 17	κ^{mag}	0	0
18	κ^{uni}	46	0

Table C.4: MQ Context State Initialization

Σ	Transition		X_s	Estimate $Q(\text{hex})$	Σ	Transition		X_s	Estimate $Q(\text{hex})$
	Σ_{mps}	Σ_{lps}				Σ_{mps}	Σ_{lps}		
0	1	1	1	x5601	24	25	22	0	x1C01
1	2	6	0	x3401	25	26	23	0	x1801
2	3	9	0	x1801	26	27	24	0	x1601
3	4	12	0	x0AC1	27	28	25	0	x1401
4	5	29	0	x0521	28	29	26	0	x1201
5	38	33	0	x0221	29	30	27	0	x1101
6	7	6	1	x5601	30	31	28	0	x0AC1
7	8	14	0	x5401	31	32	29	0	x09C1
8	9	14	0	x4801	32	33	30	0	x08A1
9	10	14	0	x3801	33	34	31	0	x0521
10	11	17	0	x3001	34	35	32	0	x0441
11	12	18	0	x2401	35	36	33	0	x02A1
12	13	20	0	x1C01	36	37	34	0	x0221
13	29	21	0	x1601	37	38	35	0	x0141
14	15	14	1	x5601	38	39	36	0	x0111
15	16	14	0	x5401	39	40	37	0	x0085
16	17	15	0	x5101	40	41	38	0	x0049
17	18	16	0	x4801	41	42	39	0	x0025
18	19	17	0	x3801	42	43	40	0	x0015
19	20	18	0	x3401	43	44	41	0	x0009
20	21	19	0	x3001	44	45	42	0	x0005
21	22	19	0	x2801	45	45	43	0	x0001
22	23	20	0	x2401	46	46	46	0	x5601
23	24	21	0	x2201					

Table C.5: MQ-Coder Probability State Transition Table

1	$Q \leftarrow Q(\Sigma_k)$	From lookup Table C.5
2	$A \leftarrow A - Q$	Find next span of A
3	IF $x = s_k$	Coding an MPS
4	IF $A \geq 2^{15}$	Do not need to renormalize
5	$C \leftarrow C + Q$	
6	ELSE	Need to renormalize
7	IF $A < Q$	Need conditional Exchange
8	$A \leftarrow Q$	Swap sub-span
9	ELSE	
10	$C \leftarrow C + Q$	No conditional exchange
11	$\Sigma_k \leftarrow \Sigma_{MPS}(\Sigma_k)$	Transition probability estimate
12	DO	Perform renormalization
13	$A \leftarrow 2A$	Left shift
14	$C \leftarrow 2C$	
15	$\bar{t} \leftarrow \bar{t} - 1$	Renormalization count
16	IF $\bar{t} = 0$	
17	Transfer_Byte(\bar{T}, C, L, \bar{t})	Output byte
18	WHILE $A < 2^{15}$	
19	ELSE	Coding an LPS
20	IF $A < Q$	Need Conditional Exchange
21	$C \leftarrow C + Q$	Swap sub-span
22	ELSE	
23	$A \leftarrow Q$	No conditional exchange
24	$s_k \leftarrow s_k \oplus X_s(\Sigma_k)$	Change MPS if $X_s(\Sigma_k) = 1$
25	$\Sigma_k \leftarrow \Sigma_{LPS}(\Sigma_k)$	Transition probability estimate
26	DO	Perform renormalization
27	$A \leftarrow 2A$	Left shift
28	$C \leftarrow 2C$	
29	$\bar{t} \leftarrow \bar{t} - 1$	Renormalization count
30	IF $\bar{t} = 0$	
31	Transfer_Byte(\bar{T}, C, L, \bar{t})	Output Byte
32	WHILE $A < 2^{15}$	

Figure C.1: MQ Encoder Main Routine (MQ-Encode(x, κ))[15, p.646]

1	$A \leftarrow x8000$	Initialize A to 2^{15}
2	$C \leftarrow 0$	
3	$\bar{t} \leftarrow 12$	Initialize renormalization count to account for spacer bits
4	$\bar{T} \leftarrow 0$	Initialize the output byte as zero
5	$L \leftarrow -1$	Initialize the byte count to -1, this prevents the first byte from being output before a potential carry is applied

Figure C.2: MQ Encoder Initialization Routine[15, p.478]

1	IF $\bar{T} = \text{xFF}$	Must bit stuff
2	Put_Byte(\bar{T}, L)	Output byte buffer
3	$\bar{T} \leftarrow C^{msbs}$	Fill output buffer
4	$C^{msbs} \leftarrow 0$	Clear C Register
5	$\bar{i} \leftarrow 7$	Next byte will only be renormalized up to 7 times. This keeps the carry
6	ELSE	from propagating past \bar{T}
7	$\bar{T} \leftarrow \bar{T} + C^{carry}$	No need to bit stuff
8	$C^{carry} \leftarrow 0$	Apply any carry bit to the output buffer
9	Put_Byte(\bar{T}, L)	Clear Carry Bit
10	IF $\bar{T} = \text{xFF}$	Output Byte Buffer
11	$\bar{T} \leftarrow C^{msbs}$	The application of the carry bit has caused a bit stuff necessary
12	$C^{msbs} \leftarrow 0$	Fill output buffer
13	$\bar{i} \leftarrow 7$	Clear C Register
14	ELSE	Next byte will only be renormalized up to 7 times. This keeps the carry
15	$\bar{T} \leftarrow C^{partial}$	from propagating past \bar{T}
16	$C^{partial} \leftarrow 0$	Carry bit has not caused a bit stuff
17	$\bar{i} \leftarrow 8$	Fill output buffer with everything except the carry bit
		Clear C register
		Next byte will be renormalized 8 times

Figure C.3: MQ Encoder Transfer Byte Routine (Transfer-Byte(\bar{T}, C, L, \bar{i}))[15, p.479]

1	IF $L \geq 0$	If at least 1 byte has been previously output from \bar{T}
2	$B_L \leftarrow \bar{T}$	Output the byte from the buffer
3	$L \leftarrow L + 1$	Increment the output byte count.

Figure C.4: MQ Encoder Put Byte Routine (Put-Byte(\bar{T}, L))[15, p.479]

1	$n^{bits} \leftarrow 27 - 15 - \bar{i}$	Determine the number of bits to flush from C
2	$C \leftarrow 2^{\bar{i}} C$	Move the next 8 bits into $C^{partial}$
3	WHILE $n^{bits} > 0$	While there are bits left to flush
4	Transfer_Byte(\bar{T}, C, L, \bar{i})	Output byte
5	$n^{bits} \leftarrow n^{bits} - \bar{i}$	Calculate the number of bits remaining to flush
6	$C \leftarrow 2^{\bar{i}} C$	Move the next 8 bits into $C^{partial}$
7	Transfer_Byte(\bar{T}, C, L, \bar{i})	Final Output byte

Figure C.5: MQ Encoder Flush Routine[15, p.496]

BIBLIOGRAPHY

- [1] Tinku Acharya and Ping-Sing Tsai. *JPEG2000 Standard for Image Compression*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2005.
- [2] Peter J. Ashenden. *The vhdl cookbook*. Internet, 1990.
- [3] Eric J. Balster. "VIDEO COMPRESSION AND RATE CONTROL METHODS BASED ON THE WAVELET TRANSFORM". 2004.
- [4] Mr. Martin Boliek, editor. *JPEG 2000 Part I Final Committee Draft Version 1.0*. Joint Pictures Expert Group, ISO/IEC, 2000.
- [5] K.C. Chang. *Digital Design and Modeling with VHDL and Synthesis*. Wiley, 1997.
- [6] Diego Santa Cruz and Touradj Ebrahimi. A study of jpeg 2000 still image coding versus other standards. ISO/IEC JTC1/SC29/WG1, The Proceedings of EUSIPCO 2000.
- [7] Majid Rabbani Diego Santa Cruz. The jpeg2000 still-image compression standard. Slides in PDF Format.
- [8] Michael P. Flaherty. A study of the design and real-time implementation of a semi-generic integer-to-integer discrete wavelet transform. Master's thesis, University of Dayton, 300 College Park, Dayton, OH 45440, December 2006.
- [9] Gaetano Impoco. Jpeg2000 - a short tutorial, April 2001.
- [10] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.
- [11] W. B. Pennebaker and J. L. Mitchell. Probability estimation for the q-coder. *IBM J. RES. Develop*, 32(6):737 – 751, November 1988.
- [12] Richard E. Woods Rafael C. Gonzalez. *Digital Image Processing*. Addison-Wesley Publishing Company, Inc., 1993.
- [13] Thomas D. Schneider. *Information theory primer*. 2005.
- [14] Paul R. Schumacher. An efficient jpeg2000 tier-1 coder hardware implementation for real-time video processing. *IEEE Transactions on Consumer Electronics*, 49(4):780–786, November 2003.
- [15] David S. Taubman and Michael W. Marcellin. *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, Boston, Mass., 2002.

- [16] Krishnaraj Varma and Amy Bell. Jpeg2000choices and tradeoffs for encoders. *IEEE Signal Processing Magazine*, pages 70–75, November 2004.
- [17] James S. Walker. "*A Primer On Wavelets and their Scientific Applications*". Chapman & Hall/CRC, Boca Raton, FL, 1999.

R00259319Z