

2009

## A parallel architecture of JPEG2000 Tier I encoding for FPGA implementation

Luke A. Hogrebe  
*University of Dayton*

Follow this and additional works at: [https://ecommons.udayton.edu/graduate\\_theses](https://ecommons.udayton.edu/graduate_theses)

---

### Recommended Citation

Hogrebe, Luke A., "A parallel architecture of JPEG2000 Tier I encoding for FPGA implementation" (2009).  
*Graduate Theses and Dissertations*. 3344.  
[https://ecommons.udayton.edu/graduate\\_theses/3344](https://ecommons.udayton.edu/graduate_theses/3344)

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](mailto:mschlangen1@udayton.edu), [ecommons@udayton.edu](mailto:ecommons@udayton.edu).

A PARALLEL ARCHITECTURE OF JPEG2000 TIER I  
ENCODING FOR FPGA IMPLEMENTATION

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

Luke A. Hogrebe, B.S.

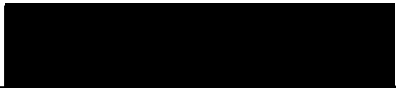
UNIVERSITY OF DAYTON

Dayton, Ohio

August 2009


A PARALLEL ARCHITECTURE OF JPEG2000 TIER I  
ENCODING FOR FPGA IMPLEMENTATION

APPROVED BY:



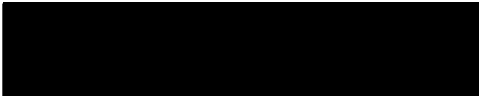
---

Eric Balster, Ph.D.  
Advisor Committee Chairman  
Electrical & Computer Engineering




---

Frank Scarpino, Ph.D.  
Committee Member  
Electrical & Computer Engineering




---

Ralph Barrera, D.E.  
Committee Member  
Electrical & Computer Engineering




---

Donald Moon, Ph.D.  
Department Chairman  
Electrical & Computer Engineering



---

Malcolm Daniels, Ph.D.  
Interim Dean  
School of Engineering



---

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

## ABSTRACT

### A PARALLEL ARCHITECTURE OF JPEG2000 TIER I ENCODING FOR FPGA IMPLEMENTATION

Name: Luke A. Hogrebe  
University of Dayton, 2009

Advisor: Eric Balster

In the past image compression techniques have primarily competed to provide the smallest file size while maintaining the best possible image quality. While this is still true for image compression, new applications have introduced the need to expand upon the features of existing compression methods. The leading international compression standard, JPEG, has met the needs of digital imaging applications for years and continues to do so for most non-specialized applications. However, with the growing numbers and resolutions of images in existence, JPEG is showing signs of its age. As a result, JPEG2000 has been introduced as a feature rich image compression standard. JPEG2000 offers powerful features such as high quality at low bit-rates, bit-rate control, and file-stream scalability in terms of resolution, quality, components, and spatial region. While JPEG2000 is not yet remotely close to being utilized in the breath of applications as JPEG, many specialized applications, such as medical imaging and persistent surveillance, are able to take advantage of JPEG2000's added feature set. However, many of these specialized applications deal with extremely high image resolutions and numbers. Compressing large quantities of image data presents

a challenge for JPEG2000, in that JPEG2000 is substantially more complex than JPEG and therefore takes longer to compress images. However, JPEG2000 possesses the ability to compress small sections of an image in parallel. This thesis investigates the implementation of the most computationally intensive portion of JPEG2000, referred to as Tier I, in a parallel architecture on Field Programmable Gate Arrays (FPGAs) as a solution to real-time compression requirements for high resolution imagery. Results show that when compared to a highly optimized all-software implementation built upon Intel's Integrated Performance Primitives running on a single CPU core, the hardware acceleration can introduce raw Tier I processing speed improvements on the order of 90%. Integration of the hardware version of Tier I with the rest of the Intel based software shows over a 40% improvement in overall image compression rates.

## ACKNOWLEDGMENTS

I would like to thank everyone involved in making my graduate experience not only possible, but incredibly enjoyable as well. In particular I would like to thank:

- **Dr. Frank Scarpino:** For all of your patience, dedication, advice, and insight in both engineering and life matters. Your motto, "It's all about the education," will be remembered and embraced as I continue my graduate studies. Also, thank you for providing me with the opportunity to work with such a bright and talented group. My graduate experience has been more worthwhile than I ever anticipated.
- **Dr. Eric Balster:** For serving as my advisor and keeping me on schedule to graduate. You always look out for your students' best interests, and I thank you for helping me.
- **Kerry Hill, Al Scarpelli, and the Air Force Research Laboratory at Wright-Patterson Air Force Base:** For all of your confidence, lab space and resources, and financial support. Your support for students in the Reconfigurable Computing Laboratory is definitely appreciated, I am extremely grateful for the opportunity to have been a part of the group.
- **David Walker:** For your patience and help in obtaining a functional hardware design. I could have not asked for a better person to work with. You made the many hours spent debugging much easier and more productive.
- **Thaddeus Marrara, Nick Vican, Ken Simone, and Bill Turri:** For all of your engineering guidance and support. You all are always ready and willing to answer my many questions. I would also like to thank Nick Vican and Bill Turri for code contributions that significantly helped get the hardware design off the ground in its early stages.
- **David Lucking and Ben Fortener:** For always being there to answer questions. Your knowledge of JPEG2000 has been particularly appreciated.
- **David Mundy:** For your software implementation of JPEG2000 and hardware

implementation of the MQ Coder. Your hard work made this thesis possible.

- **Dr. Barrera and Dr. Moon:** For serving on my thesis committee.

## TABLE OF CONTENTS

	Page
Approval . . . . .	ii
Abstract . . . . .	iii
Acknowledgments . . . . .	v
List of Tables . . . . .	ix
List of Figures . . . . .	x
 Chapters:	
1. Introduction . . . . .	1
1.1 JPEG2000 Features . . . . .	5
1.2 Thesis Objective and Organization . . . . .	10
2. The JPEG and JPEG2000 Compression Standards . . . . .	11
2.1 The JPEG Compression Standard . . . . .	11
2.2 The JPEG2000 Compression Standard . . . . .	16
2.2.1 Tiling . . . . .	17
2.2.2 DC Level Shift . . . . .	18
2.2.3 Color Transform . . . . .	19
2.2.4 Discrete Wavelet Transform . . . . .	20
2.2.5 Quantization . . . . .	25
2.2.6 Embedded Block Coding With Optimized Truncation . . . . .	26
3. JPEG2000 Tier I Hardware Architecture . . . . .	41
3.0.7 Top Level . . . . .	43
3.0.8 Tier I Controller . . . . .	51
3.0.9 Tier I Top . . . . .	55

3.0.10	Stripe FIFO . . . . .	60
3.0.11	Stripe RAM . . . . .	60
3.0.12	K Calculator . . . . .	61
3.0.13	PSDX RAM . . . . .	64
3.0.14	Sign Loader . . . . .	66
3.0.15	Tier I Sub . . . . .	69
3.0.16	Stripe Bits Controller . . . . .	76
3.0.17	State Tables Controller . . . . .	77
3.0.18	The Coding Passes . . . . .	90
3.0.19	Decision Bit and Context FIFO . . . . .	102
3.0.20	MQ Coder . . . . .	102
3.0.21	MQ Coder Output and Trailer Packer . . . . .	103
3.0.22	Packed Output FIFO . . . . .	106
4.	Hardware Performance . . . . .	108
5.	Future Work and Conclusions . . . . .	122
5.1	Future Work . . . . .	122
5.2	Conclusions . . . . .	123
	Bibliography . . . . .	125

## LIST OF TABLES

Table	Page
2.1 Lossy 9/7 Filter Coefficients . . . . .	24
2.2 Lossless 5/3 Filter Coefficients . . . . .	25
2.3 Software Timing Results for a 1k×1k Color Tile at 10:1 Compression	27
2.4 <b>LL/LH</b> Sub-band Zero Coding Context Lookup Table . . . . .	32
2.5 <b>HL</b> Sub-band Zero Coding Context Lookup Table . . . . .	33
2.6 <b>HH</b> Sub-band Zero Coding Context Lookup Table . . . . .	33
2.7 Sign Coding Context and $\hat{\chi}$ Lookup Table . . . . .	35
2.8 Magnitude Refinement Context Lookup Table . . . . .	36
4.1 Altera Stratix III EP3SL150 FPGA Resource Utilization for 2, 16, and 31 Tier I Cores . . . . .	109
4.2 Hardware Timing for <i>peppers</i> . . . . .	112
4.3 Software Timing for <i>peppers</i> . . . . .	112
4.4 Hardware Timing for <i>pentagon</i> . . . . .	112
4.5 Software Timing for <i>pentagon</i> . . . . .	112
4.6 Percentage Time Improvements for the Hardware Tier I Implementa- tion vs Software for <i>peppers</i> . . . . .	121
4.7 Percentage Time Improvements for the Hardware Tier I Implementa- tion vs Software for <i>pentagon</i> . . . . .	121

## LIST OF FIGURES

Figure	Page
1.1 Principal stages of a general compression/decompression system. . . .	2
1.2 Standard 512 x 512 uncompressed grayscale <i>peppers</i> image. . . . .	7
1.3 Examples of compression distortion of <i>peppers</i> at .1 bpp using JPEG (left) and JPEG2000 (right). . . . .	7
1.4 Example of JPEG2000 quality scalability. . . . .	8
1.5 Example of JPEG2000 resolution scalability. . . . .	9
2.1 JPEG compression blocking artifacts. . . . .	14
2.2 MCU consisting of six data blocks for 4:2:0 subsampling format with interleaving. One data block is composed of an 8x8 group of DCT coefficients. . . . .	15
2.3 Scanning order of AC DCT coefficients for run-length coding for a data block. . . . .	17
2.4 JPEG2000 compression block diagram. . . . .	18
2.5 Original image (left); Result of row processing and downsampling (center); Result of column processing and downsampling to form 4 sub-bands (right). . . . .	21
2.6 1 level of wavelet decomposition of <i>peppers</i> . . . . .	22
2.7 Forward DWT filter processing diagram for one decomposition level. .	23
2.8 Forward DWT filter processing diagram for three decomposition levels.	23
2.9 2 levels of wavelet decomposition of <i>peppers</i> . . . . .	23
2.10 Code-block partitioning for <i>peppers</i> using a code-block size of 32x32 samples. Each code-block can be sent to a different instance of a Tier I encoder. . . . .	28
2.11 Bit-plane images for 8 bit grayscale <i>peppers</i> . . . . .	29
2.12 Depiction of the order stripes are processed for a code-block. . . . .	31
2.13 Depiction of a stripe neighborhood and the neighborhood of a stripe bit being coded. . . . .	32
2.14 Run-length coding (RLC) example. . . . .	34
2.15 Block diagram for the Significance Propagation Pass. . . . .	37
2.16 Block diagram for the Magnitude Refinement Pass. . . . .	38
2.17 Block diagram for the Cleanup Pass. . . . .	39

3.1	Hierarchy of the JPEG2000 Tier I encoder architecture. . . . .	42
3.2	Depiction of how code-blocks are stored in the <i>Primary Input FIFO</i> . .	44
3.3	<i>mymodule</i> state machine and other processes. . . . .	45
3.4	<i>mymodule</i> and <i>Tier I Controller</i> interface diagram with multiplexors and demultiplexors combined for Group 1 and 2 signals. . . . .	49
3.5	<i>Tier I Controller</i> state machine and other processes. . . . .	52
3.6	<i>Tier I Top</i> components and major interconnect signals. . . . .	56
3.7	<i>Tier I Top</i> state machine and other processes. . . . .	57
3.8	Depiction of how code-block stripes are stored in the <i>Tier I Stripe FIFOs</i> . .	61
3.9	Diagram showing how bit-plane <i>K</i> is found by the <i>K Calculator</i> . . . .	62
3.10	<i>K Calculator</i> state machine. . . . .	62
3.11	<i>PSDX RAM</i> memory organization for one state table. . . . .	65
3.12	<i>PSDX RAM</i> example access of stripes 1 and 33. . . . .	65
3.13	<i>PSDX RAM</i> memory organization. . . . .	66
3.14	<i>Sign Loader</i> state machine. . . . .	67
3.15	Example of where the first stripe's sign bits are loaded in terms of <i>PSDX RAM</i> addresses for a code-block. . . . .	68
3.16	<i>Tier I Sub</i> components and major interconnect signals. . . . .	70
3.17	Detailed diagram for module connections between several <i>Tier I Sub</i> components (shaded region of Figure 3.16). . . . .	71
3.18	<i>Tier I Sub</i> state machine for the <i>State Tables Controller</i> and coding pass interface management. . . . .	72
3.19	<i>Tier I Sub</i> state machine for <i>MQ Coder</i> interface management and byte counting process. . . . .	74
3.20	Depiction of how <i>Stripe RAM</i> locations correspond to code-block stripes. .	77
3.21	Depiction of how the <i>Stripe Bits Controller</i> accesses code-block stripes and the state table information provided to a coding pass. . . . .	78
3.22	<i>State Tables Controller</i> state machine. . . . .	80
3.23	Depiction of the state table registers, the write register, and the state table <i>PSDX RAM</i> before any bit-plane has been processed. The sign table may be non-zero. . . . .	82
3.24	Depiction of the <i>State Tables Controller</i> PRELOAD operations. . . .	83
3.25	Depiction of the <i>State Tables Controller</i> first loading operations for a bit-plane. . . . .	84
3.26	Depiction of the <i>State Tables Controller</i> first writing operations for a bit-plane. . . . .	85
3.27	Depiction of the <i>State Tables Controller</i> second loading operations for a bit-plane. . . . .	86
3.28	Depiction of the <i>State Tables Controller</i> second writing operations for a bit-plane. . . . .	86

3.29	Depiction of the <i>State Tables Controller</i> first read operations for the first flush for a bit-plane. . . . .	88
3.30	Depiction of the <i>State Tables Controller</i> first write operations for the first flush for a bit-plane. . . . .	88
3.31	Depiction of the <i>State Tables Controller</i> second read operations for the first flush for a bit-plane. . . . .	89
3.32	Depiction of the <i>State Tables Controller</i> second write operations for the first flush for a bit-plane. . . . .	89
3.33	Depiction of the how the <i>State Tables Controller</i> determines a bit-plane has been processed. . . . .	90
3.34	<i>Significance Propagation Pass</i> state machine. . . . .	92
3.35	<i>Magnitude Refinement Pass</i> state machine. . . . .	95
3.36	<i>Cleanup Pass</i> state machine. . . . .	98
3.37	Depiction of how data is stored in the <i>Decision Bit and Context FIFO</i> . . . . .	103
3.38	<i>MQ Coder Output and Trailer Packer</i> state machine. <code>PACK_BYTE_14:1</code> states are inferred. . . . .	104
3.39	Depiction of how data is stored in the <i>Packed Output FIFO</i> . . . . .	107
4.1	$2k \times 2k$ <i>pentagon</i> . . . . .	111
4.2	$6k \times 6k$ <i>peppers</i> . . . . .	111
4.3	Hardware and software timing comparisons for <i>peppers</i> $2k \times 2k$ . . . .	113
4.4	Hardware and software timing comparisons for <i>pentagon</i> $2k \times 2k$ . . . .	113
4.5	Hardware and software timing comparisons for <i>peppers</i> $4k \times 4k$ . . . .	114
4.6	Hardware and software timing comparisons for <i>pentagon</i> $4k \times 4k$ . . . .	114
4.7	Hardware and software timing comparisons for <i>peppers</i> $6k \times 6k$ . . . .	115
4.8	Hardware and software timing comparisons for <i>pentagon</i> $6k \times 6k$ . . . .	115
4.9	Hardware and software timing comparisons for <i>peppers</i> $8k \times 8k$ . . . .	116
4.10	Hardware and software timing comparisons for <i>pentagon</i> $8k \times 8k$ . . . .	116
4.11	Hardware and software timing comparisons for <i>peppers</i> $10k \times 10k$ . . . .	117
4.12	Hardware and software timing comparisons for <i>pentagon</i> $10k \times 10k$ . . . .	117
4.13	Hardware and software Tier I timing comparisons for <i>peppers</i> . . . . .	118
4.14	Hardware and software Tier I timing comparisons for <i>pentagon</i> . . . . .	119
4.15	Hardware and software timing comparisons for beginning-to-end compression for <i>peppers</i> . . . . .	120
4.16	Hardware and software timing comparisons for beginning-to-end compression for <i>pentagon</i> . . . . .	120

## CHAPTER 1

### Introduction

The use of digital imagery has become so vast in both specialized applications and everyday life that it is nearly impossible to imagine an advancing technological world in which digital images are not significantly involved. Current applications ranging from medical diagnosis, remote sensing, automated inspection, digital archives, mobile devices, to the Internet all have come to rely extensively if not solely on the ability to utilize digital images. The substantial and growing use of digital images and the increasing resolutions of digital cameras drive the need to store the imagery efficiently. A modest 5 megapixel monochrome image with 8 bits of precision for each pixel requires 5 MB of storage memory without any compression. Two hundred pictures consumes 1 GB of storage space; thus storing images in their raw pixel format is wasteful of storage memory. Compression is the means by which pixel data is transformed so that the image is represented by a fewer number of bits while residing in storage (or when transmitted to reduce bandwidth consumption). In other words, it is a method for reducing redundant data while the image is stored and transmitted [1, pg 5]. The overall transformation can be carried out such that the original image is obtained exactly following decompression, or the resulting image after decompression sustains some degree of distortion compared to the original image,

although with the benefit of a smaller compressed file size. The block diagram for a general compression/decompression system is shown in Figure 1.1.

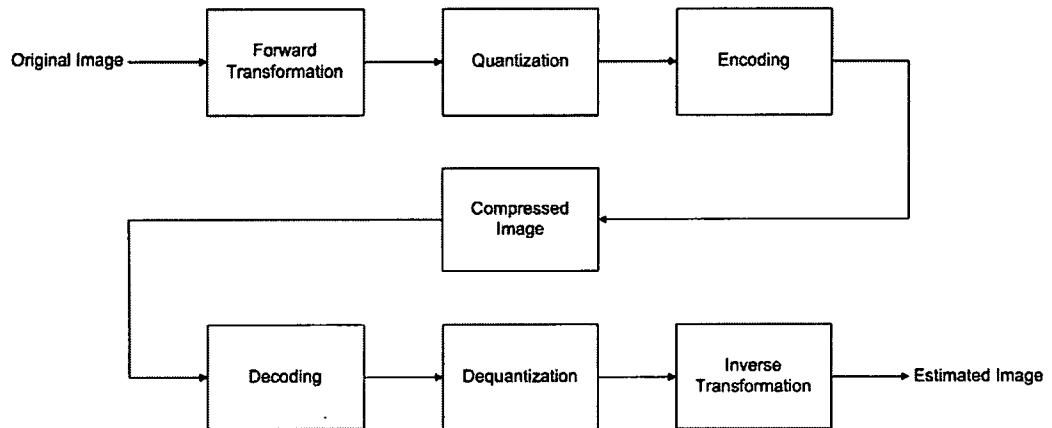


Figure 1.1: Principal stages of a general compression/decompression system.

The forward transform decreases redundancy in the original image [7, pg 421]. That is to say, in a typical natural image the information held by a pixel is highly correlated to that of the surrounding pixels. A pixel is predicted with reasonably high confidence based on its neighbors, meaning the overall contribution of valuable information from that pixel is relatively small [7, pg 415]. The forward transform decorrelates pixel data in the spatial domain, and in the cases of the JPEG and JPEG2000 compression techniques, maps the data to frequency based domains.<sup>1</sup> The objective is to force a small number of transformed samples to contain as much information as possible, so the remaining data can be reduced through quantization [7, pg 470]. The forward transform is generally considered lossless, meaning the inverse transform process will produce the exact same original image. However, if floating

<sup>1</sup>The Discrete Wavelet Transform used in JPEG2000 actually produces transformed data that is both frequency and spatially based.

point numbers are used for the transform operations (as opposed to integers), the transform is not considered to be truly lossless.

The quantization stage of an image compression system is the reason why high compression ratios can be achieved. Quantization is the process of throwing data away so that the compressed image can be represented by a fewer number of bits. Quantization is irreversible, i.e. a lossy operation, since data is lost to obtain the compression gain. An application in which an exact representation of the original image is required, such as in areas of medical imaging [9], the quantization stage is omitted. In most cases, however, at least some degree of quantization is acceptable. Ultimately, the amount of quantization is dictated by the desired quality of the decompressed image for the end user. If the end user desires high compression ratios, then copious quantization can be applied if the decompressed image quality is within tolerable limits.

The advantage of transforming an image to a frequency based domain in JPEG and JPEG2000 is that information can be easily discarded without the human eye detecting a significant impact. Quantizing the data takes advantage of psychovisual redundancy because transformed samples that carry little information are eliminated, allowing for compression gains without any substantial visual quality loss [7, pg 417]. As more and more data is thrown out, however, distortion inevitably becomes perceptible.

The last stage in obtaining a compressed image is the encoding process, in which the (non-)quantized data is mapped to a series of symbols that are encoded. Encoding the symbols involves mapping them to codewords based on their probabilities of occurrence. For variable-length coding, symbols that arise frequently are represented

by codewords of shorter length (i.e. fewer bits) than infrequent symbols. The average number of bits required to represent a particular group of symbols is defined as the entropy. Equation 1.1 shows how entropy is calculated for a group of symbols, called an alphabet, where  $p(a_i)$  represents the probability of occurrence of a particular symbol out of a possible  $N$ . These probabilities can be derived from the source data itself, but in practice they are instead provided by a model, or an estimate of the source data. Entropy is an important part of information theory since it places a bound on the maximum compression obtainable for a model. Overall, the encoding stage accomplishes the task of removing statistical redundancies in the data and is commonly referred to as entropy encoding [1, pg 16]. Entropy encoding is a lossless procedure.

$$E = - \sum_{i=1}^N p(a_i) \log_2 p(a_i) \quad (1.1)$$

After entropy encoding, the final code-stream is assembled, and a compressed image is obtained. To reacquire a displayable image, the decompression stages simply invert the compression process through decoding, dequantization, and an inverse transform. The decoding stage produces (non-)quantized transform data. If the compression is lossy, the dequantization stage makes assumptions about what data has been thrown away and may add information back to the transformed data. Since quantization is irreversible, however, the original transformed data is impossible to gain back exactly. Finally, the inverse transform provides a displayable image. As Figure 1.1 shows, the final output after decompression is only an estimate of the original image if the data is quantized during compression. For lossless compression, the final output is the original image.

Considering the growing numbers and sizes of digital images in existence today, the drive to improve compression methods is ever present. The compression technique at the core of this thesis is JPEG2000. The JPEG2000 compression engine has recently been developed to meet the needs of more demanding digital imaging applications, and is on its way to becoming more widely accepted as a standard for image compression. The next section discusses the advantages of using JPEG2000, particularly compared to its predecessor, JPEG.

## **1.1 JPEG2000 Features**

For any application requiring image compression, considerations must be made as to how the data will be transformed, i.e. represented by a fewer number of bits, for eventual storage or transmission. Image quality, file size, computational intensity, and features of the compression scheme are all critical factors for consideration. One of the most common methods of image compression in existence is JPEG, largely due to the fact that JPEG is the first international still image compression standard for grayscale and color images [1, pg 55]. Created by the Joint Photographic Experts Group in the late 1980s and finalized in 1992, JPEG now ranges in a very wide array of applications from digital cameras to the Internet. JPEG's ultimate successor, JPEG2000, is the latest image compression method conceived by the JPEG committee. While JPEG2000's current use is still far from mainstream, the new standard provides many advantages over the aging JPEG standard [10].

One of JPEG2000's most prominent attributes is visual quality at low bit-rates, that is bit-rates below .25 bits per pixel (bpp) [1, pg 139]. The bit-rate for an 8 bit

monochrome image can be calculated by:

$$\text{bit-rate} = \frac{\text{size of compressed file (bits)}}{\text{image width} \times \text{image height (pixels)}} \quad (1.2)$$

JPEG introduces significant distortion at low bit-rates. In particular, the small blocks an image is divided into for processing become distinctly visible. JPEG2000 processing differs so that the small blocking artifacts are avoided. Simply stated, as compression ratios increase the subjective quality of JPEG2000 images is better than JPEG [2, 12]. A justifying example is shown in Figure 1.3 comparing JPEG to JPEG2000 at a bit-rate of .1 bpp. The original image is shown in Figure 1.2 as a degradation reference [17]. Thus, for the same quality as a JPEG image, a JPEG2000 image is better suited for transmission because it utilizes the available bandwidth more effectively due to its smaller file size; i.e. there is less data to transmit for the same quality as a JPEG image. In real-time surveillance applications where imagery must be transmitted from the source and distributed to many viewers at even moderate rates, maximizing bandwidth efficiency is particularly important. Considering the other end of the spectrum, at high bit-rates where JPEG and JPEG2000 images are visually identical, JPEG2000 obtains a 20% average compression gain over JPEG [15, pg 402].

Another desirable trait of JPEG2000 from an image distribution perspective is the ability to acquire pieces of the code-stream progressively. The JPEG2000 code-stream can be organized such that images are received by increasing quality (pixel accuracy) or resolution (Figures 1.4 and 1.5, respectively). That is to say, the JPEG2000 code-stream is highly scalable. Images can be requested from mobile multimedia devices, such as smartphones, and the devices will only receive the resolution suitable for their screen at the highest quality. Even though only part of the JPEG2000 file is acquired,

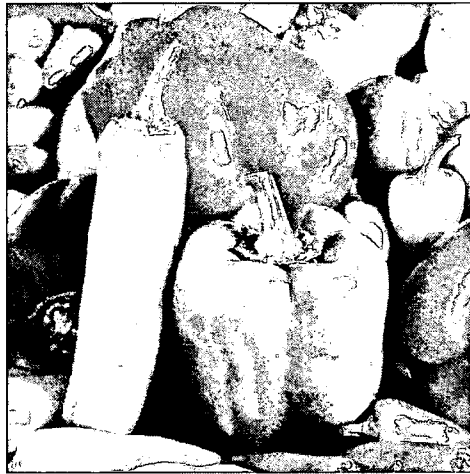


Figure 1.2: Standard 512 x 512 uncompressed grayscale *peppers* image.



Figure 1.3: Examples of compression distortion of *peppers* at .1 bpp using JPEG (left) and JPEG2000 (right).

it is fully decodable. Especially where bandwidth and device memory are limited, the ability to only request a desired resolution and quality is highly advantageous. In addition, the device could be designed so that a low quality image at the desired resolution displays as the first bits of the file are obtained. Upon downloading more of the file the image would refresh clearer. The principle applies as well to resolution progression, which could easily be applicable for use on the Internet. An image could be downloaded first at a low resolution to produce a thumbnail image and refreshed with a subsequent higher resolution image upon selection by the user [8]. The code-stream scalability of JPEG2000 is truly summarized by the phrase *compress once but decompress in many ways* [16].



Figure 1.4: Example of JPEG2000 quality scalability.

Also significant is the ability to randomly access the code-stream to extract desired spatial areas for display. This feature coincides with the aforementioned progression options in that a decodable image can be obtained upon receiving only sections of the JPEG2000 file. To save bandwidth, memory, and decoding time, image viewers can request only a region at a specified quality and resolution level. Similarly, a Region of Interest (ROI) option exists such that a user defined region of the image retains

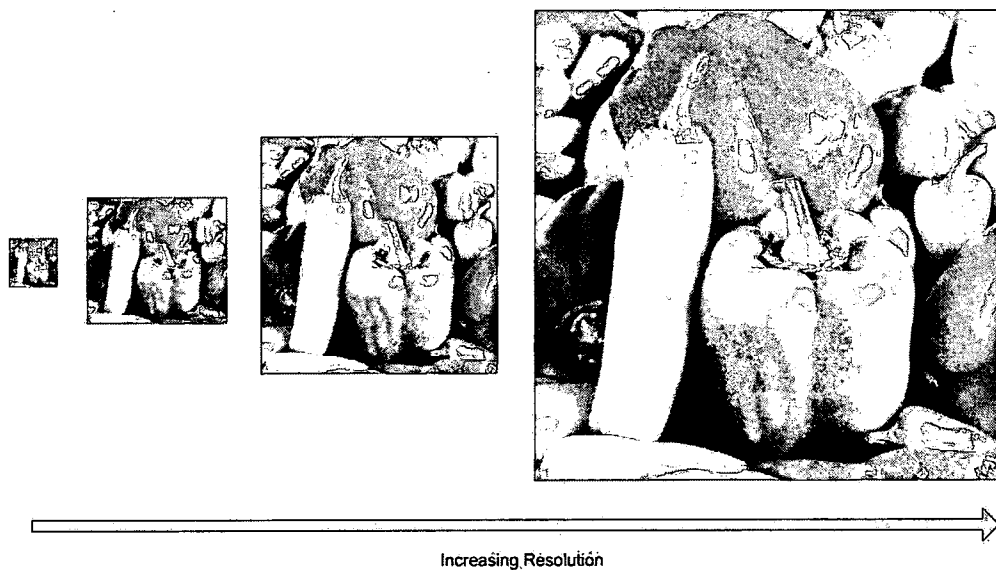


Figure 1.5: Example of JPEG2000 resolution scalability.

higher quality than the rest of the image. An ROI could be useful if file size limits are rigid and the end user wants a certain area of the image to possess maximum quality.

Other features include lossy and lossless coding modes without drastic variations between implementations and robustness to code-stream errors. Desired file sizes can be specified through the use of a rate-distortion tracking method. JPEG2000 allows for up to 38 bits of precision per image component,  $2^{14}$  components, and image sizes of  $(2^{32} - 1) \times (2^{32} - 1)$  as well. For applications using imagery such as satellite or multispectral, these bounds provide great flexibility in how images are both processed and stored. JPEG2000 also has specifications for video compression, called Motion JPEG2000, and the Internet, called JPEG2000 Internet Protocol (JPIP). One last major feature of JPEG2000 is the ability to tile an image to reduce memory

requirements. For further descriptions of these JPEG2000 benefits, refer to [15, pg 402] and [1, pg 139].

## **1.2 Thesis Objective and Organization**

The objective of this thesis is to develop a parallel architecture targeted for FPGA implementation for the most computationally intensive portion of the JPEG2000 encoding scheme, referred to as Tier I, for a wide area persistent surveillance application. Largely because of the advantages mentioned above, JPEG2000 is the chosen compression engine of the surveillance system. Using JPEG2000, however, does not come without a price. The computational intensity of JPEG2000 presents a challenge for compressing very high resolution imagery (greater than 100 megapixels) at even moderate frame rates. Fortunately, JPEG2000 lends itself to the parallel processing of images, making JPEG2000 a suitable candidate for FPGA implementation.

Chapter 2 of this thesis provides an overview of the principle components of the JPEG and JPEG2000 compression algorithms. Chapter 3 explains the currently functioning JPEG2000 Tier I hardware module by module. Chapter 4 reveals how well the hardware performs, and Chapter 5 presents future work regarding the compressor along with conclusions.

## CHAPTER 2

### The JPEG and JPEG2000 Compression Standards

#### 2.1 The JPEG Compression Standard

Both JPEG and JPEG2000 algorithmically follow the progression of data transformations, quantization, and encoding. However, the means by which these processes are accomplished differ significantly. This section outlines the principle components of 8 bit lossy baseline JPEG compression. Baseline JPEG refers to the methodology of transforming  $8 \times 8$  blocks of an image by means of the Discrete Cosine Transform (DCT), reducing the precision of the transformed data using quantization tables, and entropy encoding the resulting data according to the Huffman coding technique [14, pg 109]. For most applications, baseline JPEG's support of 8 bit data per image component is sufficient. Image components form color spaces (coordinate systems for colors), such as the red-green-blue (RGB) color space or the luminance-chrominance (LC) color space. A color bitmap consists of three color components, namely red, green, and blue. Corresponding points from each color plane form a picture element (pixel). Assuming that a pixel is represented by 24 bits (8 bits for each component), a pixel can take on a color shade mapped to a number 0 through  $2^{24} - 1$ , or 16,777,215. Grayscale images consist of only one 8 bit component representing 256 shades of gray.

For color images, a color transform is first applied to convert the color space from RGB to LC, presumably YCbCr. Chrominance (color) information is perceived more

modestly by the human eye than luminance (intensity) information and can be subsampled while having minimal visual impact on the reconstructed image [1, pg 61]. Isolating redundant color information and subsampling the color components reduces the amount of data that must be compressed. Subsampling by a factor of two in the horizontal and vertical directions produces a 4:2:0 color subsampling format. Equation 2.1 shows the RGB to YCbCr color transformation, and Equation 2.2 provides the inverse transformation. Both equations assume the use of 8 bit unsigned data. For grayscale imagery there is only one component, so the color space transformation is obviously omitted.

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.29900 & 0.58700 & 0.11400 \\ -0.16874 & -0.33126 & 0.50000 \\ 0.50000 & -0.41869 & -0.08131 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \quad (2.1)$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0.0 & 1.40210 \\ 1.0 & -0.34414 & -0.71414 \\ 1.0 & 1.77180 & 0.0 \end{pmatrix} \begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} - \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \quad (2.2)$$

If the source image is to be compressed with data loss<sup>2</sup> the DCT is used to represent the image in terms of frequency and amplitude varying cosine functions. Prior to DCT processing, each pixel is level shifted to create a two's complement signed representation. For 8 bit unsigned data this is accomplished by subtracting 128 from each component sample. The DCT then localizes most of the energy (generally) of the image into low frequency regions. As a result high frequency (edge detail) information can be readily discarded to boost compression gain by truncating the high frequency DCT coefficients. JPEG protocol calls for an image to be broken into sections of 8×8

<sup>2</sup>One of the criticisms of JPEG is that the lossless encoding mode, JPEG-LS, differs significantly from the lossy.

pixels, called data blocks, for DCT processing. Equation 2.3 shows the forward DCT mapping for an image sample,  $f(x,y)$ .

$$F(u,v) = \frac{2}{\sqrt{MN}} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x,y) \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2M}\right] \quad (2.3)$$

for  $u = 0, 1, \dots, N-1$  and  $v = 0, 1, \dots, M-1$ , where

$$C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } k = 0 \\ 1 & \text{otherwise.} \end{cases} \quad (2.4)$$

Equation 2.5 provides the inverse transformation for a DCT coefficient,  $F(u,v)$ .

$$f(x,y) = \frac{2}{\sqrt{MN}} \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} C(u)C(v)F(u,v) \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2M}\right] \quad (2.5)$$

for  $x = 0, 1, \dots, N-1$  and  $y = 0, 1, \dots, M-1$ , where  $C(\cdot)$  is as defined in Equation 2.4.

Following DCT operations the image data is in order for quantization. Quantization is the process of reducing the precision of the DCT coefficients so that fewer bits are required to encode them, resulting in a smaller compressed file. Quantization introduces data loss, which is accentuated visually by the borders of the small  $8 \times 8$  data blocks as the coefficients are quantized more and more heavily. Figure 2.1 demonstrates the effects of heavy quantization on an image. Because the borders of the data blocks become readily visible at high compression ratios, the distortion introduced is commonly referred to as blocking artifacts.



Figure 2.1: JPEG compression blocking artifacts.

Quantization is accomplished by means of a uniform quantizer, given by Equation 2.6 where  $F(u, v)$  is the DCT coefficient to be quantized and  $Q(u, v)$  is an element from a user selectable quantization table. Part of the JPEG standard provides reference tables for quantization values [21], but it is not required that they are used. Because flexibility is given in carrying out quantization, a quality parameter has been devised by the Independent JPEG Group for adjusting the reference quantization values provided by the JPEG standard. More information regarding the quality factor can be found in [1, pg 68].

$$F_q(u, v) = \text{round} \left( \frac{F(u, v)}{Q(u, v)} \right) \quad (2.6)$$

The order in which component data is organized and processed after quantization is based on data block groupings called Minimum Coded Units (MCUs). The MCU structure may differ depending on the component subsampling format as well as if

the components are processed with interleaving. Interleaving means that instead of processing entire components at a time, data blocks from each component are encoded successively. An MCU for interleaving mode will consist of data blocks from all of the components. For a 4:2:0 subsampling format with interleaving, four data blocks from the Y component are encoded first, followed by one data block from the Cb component and then one data block from the Cr component. These six data blocks form an MCU. The group of four data blocks in the Y component are processed in raster scan order (left to right, top to bottom), and each group of four data blocks for subsequent MCUs are fetched in raster scan order as well. Data blocks from the Cb and Cr components are also processed in raster scan order. Figure 2.2 shows an MCU for YCbCr components of the 4:2:0 color subsampling format with component interleaving. If using non-interleaving mode all MCUs are composed of one data block from one component, and the data blocks are processed in raster scan order.

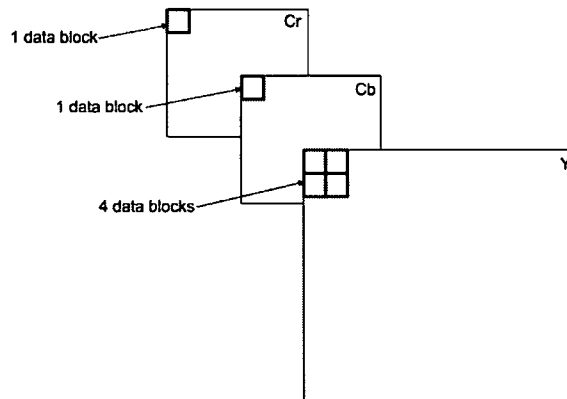


Figure 2.2: MCU consisting of six data blocks for 4:2:0 subsampling format with interleaving. One data block is composed of an  $8 \times 8$  group of DCT coefficients.

Following quantization, data blocks are entropy encoded using the Huffman coding scheme. Entropy encoding means that given a set of symbols, the symbols of higher probability are represented by fewer bits than less frequently appearing symbols. The majority of symbols in JPEG are formed by run-length codes of the AC DCT coefficients. Run-length coding reduces strings of zeros to simply the number of zeros in the string, which is used to indicate where the non-zero elements are. The AC scan pattern shown in Figure 2.3 reveals the order coefficients are examined for run-length coding. The DC coefficients are encoded differentially, meaning the DC value encoded for the current data block is dependent on the previous data block's DC coefficient. The differentials are mapped to symbols that are then Huffman coded. Following Huffman coding the final compressed JPEG file is assembled with appropriate header and marker information.

This discussion has outlined the most significant parts of baseline JPEG lossy encoding. Areas not explicitly discussed include file-stream syntax and the four modes of operation: sequential DCT-based, progressive DCT-based, lossless, and hierarchical. For more information regarding JPEG compression, refer to [1, 14, 21].

## **2.2 The JPEG2000 Compression Standard**

Despite its success, JPEG still leaves room for considerable expansion and improvement. Most of its features are underutilized by mainstream users. In fact, studies have shown that some 90% of JPEG users do not surpass use of the baseline mode [1, pg 137]. In addition, users are continuously looking to reduce compressed file sizes while retaining image quality. For equivalent high compression ratios, JPEG2000 provides much better visual quality than JPEG (Figure 1.3). This improvement is

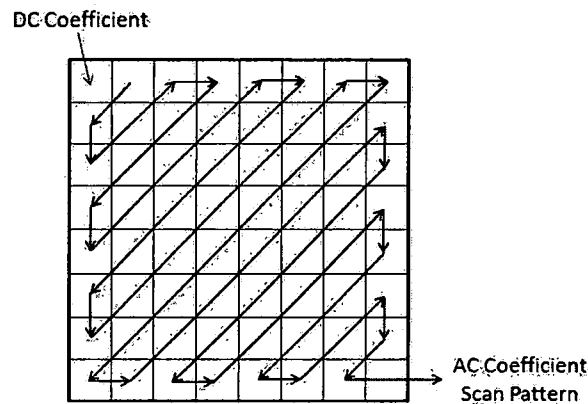


Figure 2.3: Scanning order of AC DCT coefficients for run-length coding for a data block.

due to the superior methods used to transform and encode the data. Figure 2.4 presents the overview of the JPEG2000 compression process. The following section describes what can be considered baseline JPEG2000, or Part I of the JPEG2000 standard. Additions to the standard provide more flexibility than what is discussed [12]; however, the intent is to provide an overview of the core processing concepts behind JPEG2000.

### 2.2.1 Tiling

Arguably JPEG's leading drawback is the blocking distortion introduced at high compression ratios. JPEG2000, on the other hand, avoids block distortion on small sections of data. Instead, the entire image is allowed to undergo transformation by the Discrete Wavelet Transform (DWT), hence circumventing interior boundary artifacts altogether. If memory or processing time restrictions prohibit compressing the whole image at once, the image can be partitioned into non-overlapping tiles to be encoded independently; such as in wide area persistent surveillance applications

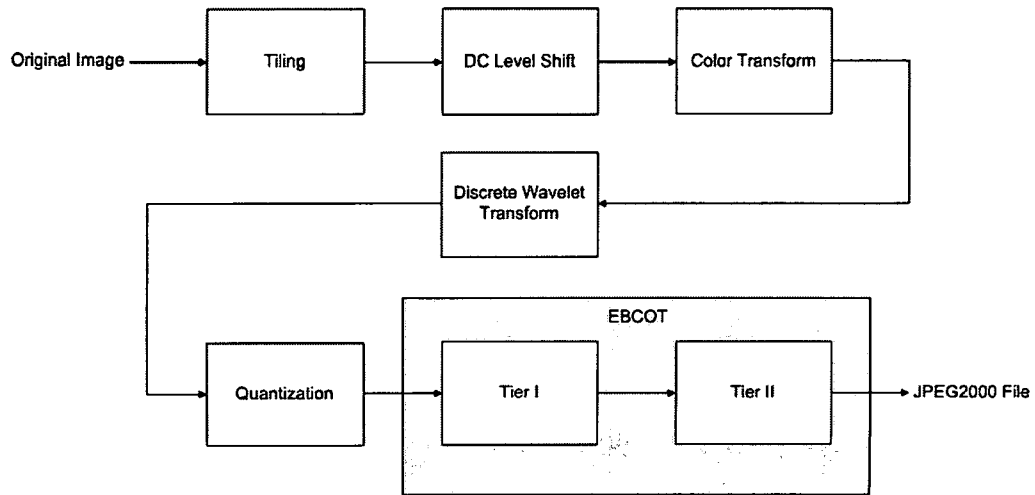


Figure 2.4: JPEG2000 compression block diagram.

where imagery may be greater than 100 megapixels. For example, a 100 megapixel image can be divided into 100 1k×1k tiles. However, tiling an image *does* have the undesired effect of introducing boundary artifacts for lossy compression [1, pg 151]. As mentioned, the reasons for tiling include memory reduction for processing and the option for parallel tile encoding to provide time savings. For more information regarding tiling, such as a description of the JPEG2000 image canvas and dealing with partial tiles, refer to [15, pg 449] and [22].

### 2.2.2 DC Level Shift

Following tiling procedures image components are subject to a DC level shift, also called a level offset. Component values are shifted so that they are distributed around zero. The shift is accomplished by subtracting  $2^{(\text{component bit depth})-1}$  from each component sample. Equation 2.7 shows the mapping of an image sample,  $I(x,y)$ , to a DC shifted value, where  $i$  references the component and  $\text{bit depth}^i$  refers to component  $i$ 's bit depth. Equation 2.8 shows the resulting range of the DC shifted image samples.

For an 8 bit component, the range of values shift from 0:255 to -128:127.

$$I_{DC \text{ shifted}}(x, y) \leftarrow I(x, y) - 2^{(bit \text{ depth}^i)-1} \quad (2.7)$$

$$-2^{(bit \text{ depth}^i)-1} \leq I_{DC \text{ shifted}}(x, y) < 2^{(bit \text{ depth}^i)-1} \quad (2.8)$$

The DWT introduces the appeal to incorporate DC level shifting because it high-pass filters the majority of the image data. The high-pass filtered data is centered around zero regardless of level offsetting, while the data only subject to low-pass filtering is not. Although considerations could be made to account for unsigned low-pass filtered samples (i.e. the DC level shift is not mandatory), for maximum decoder compatibility it is recommended the DC level shift be used [15, pg 418].

### 2.2.3 Color Transform

Like JPEG, JPEG2000 allows a color transform to decorrelate color planes to improve compression efficiency. The color planes must be the first three indexed components and have identical dimensions and bit depths. The color transform produces a luminance component and two chrominance components. For lossless compression the components are denoted YCbCr following the Reversible Color Transform (RCT) as shown in Equation 2.9. The RCT is only used in conjunction with the DWT's lossless 5/3 wavelet [22]. Equation 2.10 provides the inverse RCT.

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.25 & 0.5 & 0.25 \\ 0.00 & -1.0 & 1.00 \\ 1.00 & -1.0 & 0.00 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \quad (2.9)$$

$$\begin{aligned}
G &= Y - .25Cb - .25Cr \\
\begin{pmatrix} R \\ B \end{pmatrix} &= \begin{pmatrix} 1.0 & 0.0 & 1.0 \\ 1.0 & 1.0 & 0.0 \end{pmatrix} \begin{pmatrix} G \\ Cb \\ Cr \end{pmatrix}
\end{aligned} \tag{2.10}$$

For lossy compression the Irreversible Color Transform (ICT) is used to convert RGB to YCbCr color space and only with the DWT's 9/7 wavelet [22]. As shown in Equation 2.11 the matrix coefficients are non-integers, which introduces rounding errors and makes the transform non-reversible. The same is true for the inverse ICT in Equation 2.12.

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.29900 & 0.58700 & 0.11400 \\ -0.16875 & -0.33126 & 0.50000 \\ 0.50000 & -0.41869 & -0.08131 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{2.11}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0.00000 & 1.40200 \\ 1.0 & -0.34413 & -0.71414 \\ 1.0 & 1.77200 & 0.00000 \end{pmatrix} \begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} \tag{2.12}$$

#### 2.2.4 Discrete Wavelet Transform

The data decorrelating transformation of JPEG2000 is the Discrete Wavelet Transform (DWT). The DWT decomposes an image into spatial frequency subbands and resolutions; in practical terms, by high and low pass filtering the image in its horizontal and vertical dimensions multiple times. Figure 2.6 displays the result of one level of decomposition, denoted  $N_L = 1$ , which contains the data for two resolution levels. Figure 2.6 is obtained by first low pass filtering the image horizontally (across its rows) and downsampling by a factor of 2 along the columns. The process is repeated on

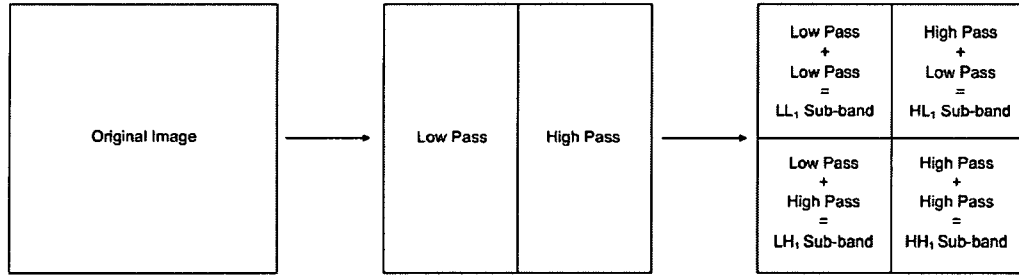


Figure 2.5: Original image (left); Result of row processing and downsampling (center); Result of column processing and downsampling to form 4 subbands (right).

the original image using a high pass filter, resulting in two image halves (Figure 2.5, center). Both image halves are then low pass filtered vertically (along their columns) and downsampled by a factor of 2 along their rows. The process is repeated using a high pass filter, and the new image quarters are merged together to form what is pictured in Figure 2.6. The far right image in Figure 2.5 shows the labeling of the different frequency feature quadrants, called subbands. The  $LL_1$  subband contains a coarse approximation (low pass filtered and downsampled) of the original image and is considered another resolution level. Therefore, DWT processing is often referred to as multi-resolution analysis. Typically for natural images, 5 levels of wavelet decomposition provide the best compression performance [20], taking into account the tile size being used. The  $HL_1$ ,  $LH_1$ , and  $HH_1$  subbands contain the additional (residual) information needed to reassemble the original image. Figure 2.7 summarizes the process just discussed to obtain one wavelet decomposition level, where  $h_L$  represents the low pass filter and  $h_H$  represents the high pass filter. Extending the concept of decomposition levels, the  $LL_1$  subband can reduce its entropy further by undergoing the process that was applied to the entire image [11]. Figure 2.8 shows a simplified

way of visualizing multi-resolution analysis. A DWT'd image will have  $N_L + 1$  image resolutions and  $3N_L + 1$  subbands. Figure 2.9 shows the result of two wavelet decomposition levels for *peppers*, with subbands labeled accordingly.



Figure 2.6: 1 level of wavelet decomposition of *peppers*.

One significant advantage of using the DWT is that different resolutions are inherently built into the transformed image. Therefore, different applications can access images suitable for their display device from one JPEG2000 file. Particularly in applications where images are distributed over a network, the ability to send only the resolution required for the application saves available network bandwidth. Another advantage is that the DWT is applied on a tile by tile basis to avoid small scale blocking artifacts. Instead, as compression ratios increase for lossy compression, the DWT distortion is seen as a blurring across the entire tile. However, for tiled images,

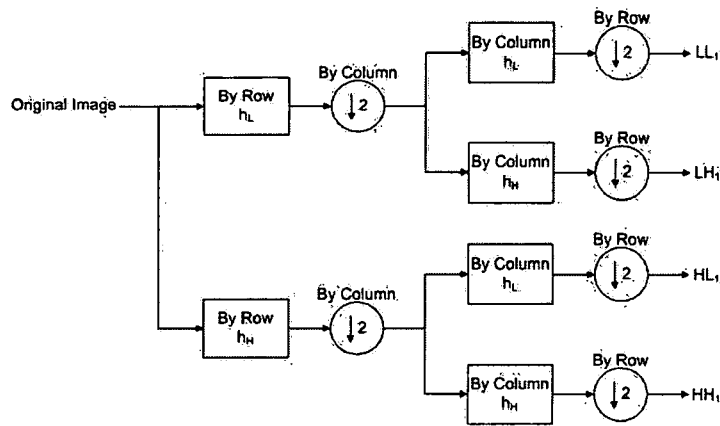


Figure 2.7: Forward DWT filter processing diagram for one decomposition level.

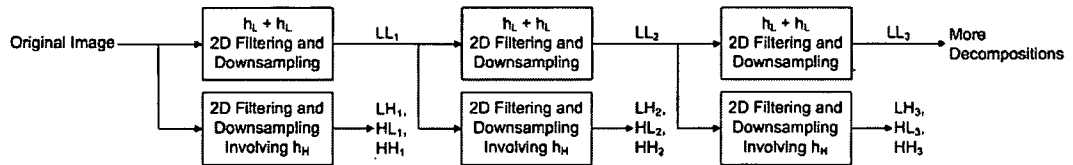


Figure 2.8: Forward DWT filter processing diagram for three decomposition levels.

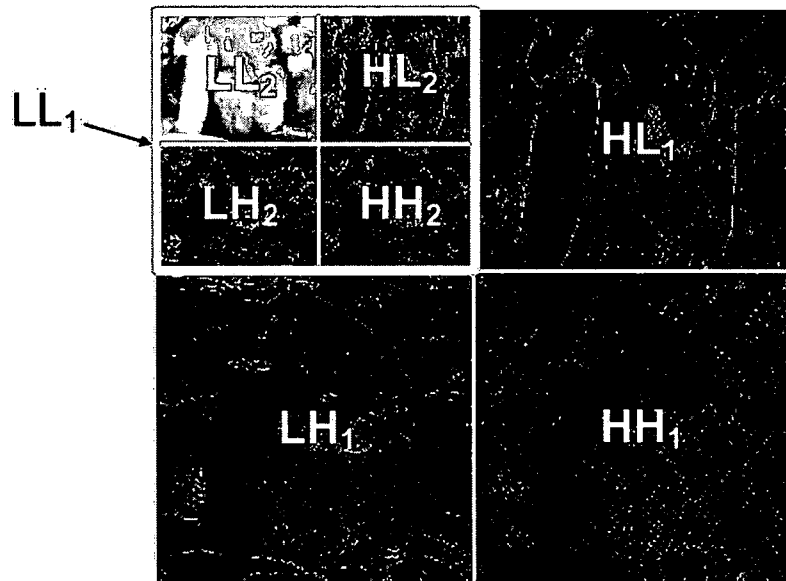


Figure 2.9: 2 levels of wavelet decomposition of *peppers*.

tile boundaries become visible as compression ratios increase. Use of the DWT is not mandatory and may be omitted such as when compressing binary images [15, pg 429].

JPEG2000 (Part I) uses one of two wavelet filter types, depending on whether the compression is lossy or lossless. For lossy compression, the Daubechies 9/7 filter is employed. For lossless compression, the 5/3 filter coefficients are used. Both sets of filters are referenced by two numbers. The first is the number of low pass filter taps in the analysis filter, and the second is the number of high pass analysis filter taps. The analysis filters are those that accomplish the forward wavelet transformation. Synthesis filters perform the task of the inverting the transformation. As found in [3], the 9-7 and 5-3 wavelet filter coefficients are listed in Tables 2.1 and 2.2.

Table 2.1: Lossy 9/7 Filter Coefficients

i	Analysis Filter Coefficients		Synthesis Filter Coefficients	
	Lowpass Filter $h_L(i)$	Highpass Filter $h_H(i)$	Lowpass Filter $h_L(i)$	Highpass Filter $h_H(i)$
0	0.6029490182363579	1.115087052456994	1.115087052456994	0.6029490182363579
$\pm 1$	0.2668641184428723	-0.5912717631142470	0.5912717631142470	-0.2668641184428723
$\pm 2$	-0.07822326652898785	-0.05754352622849957	-0.05754352622849957	-0.07822326652898785
$\pm 3$	-0.01686411844287495	0.09127176311424948	-0.09127176311424948	0.01686411844287495
$\pm 4$	0.02674875741080976			0.02674875741080976

Presented in this section has been an overview of DWT processing methodology, as well as examples of the effects of DWT processing on image data. Information such as the underlying theory of the DWT and gain bits are beyond the scope of this overview. In addition, procedures for computing DWTs other than vanilla convolution-based filtering exist. One prominent method is called lifting-based filtering [3, 13]. The

Table 2.2: Lossless 5/3 Filter Coefficients

i	Analysis Filter Coefficients		Synthesis Filter Coefficients	
	Lowpass Filter $h_L(i)$	Highpass Filter $h_H(i)$	Lowpass Filter $h_L(i)$	Highpass Filter $h_H(i)$
0	6/8	1	1	6/8
$\pm 1$	2/8	-1/2	1/2	-2/8
$\pm 2$	-1/8			-1/8

methodology presented in this section filters data and then discards half of it through downsampling. Processing data that is immediately thrown away is wasteful in terms of computing resources and time, so the lifting approach modifies the algorithm to eliminate the unnecessary processing. For more information on the DWT and computational methods, refer to [6, 7, 15, 19].

### 2.2.5 Quantization

In order to obtain respectable compression gains, the precision of the wavelet coefficients in each of the subbands is reduced to decrease the amount of data that is entropy encoded [1, pg 152]. Encoding less data results in a smaller compressed code-stream; hence the compression gain. For lossless compression quantization is not performed, allowing for only an average between 2:1 and 3:1 compression ratio [4].

Quantization is irreversible since data is lost without the ability for exact recovery. The data precision reduction occurs by dividing each of the wavelet coefficients by a subband specific parameter called the quantization step size,  $\Delta_b$ , where the subscript  $b$  refers to the coefficient's subband. A quantized coefficient,  $q_b(u, v)$ , is found using

Equation 2.13.

$$q_b(u, v) = \text{sign}[\text{coeff}_b(u, v)] \cdot \text{floor} \left[ \frac{|\text{coeff}_b(u, v)|}{\Delta_b} \right] \quad (2.13)$$

$$\Delta_b = 2^{R_b - \epsilon_b} \left( 1 + \frac{\mu_b}{2^{11}} \right) \quad (2.14)$$

The quantization step size depends on three variables as presented in Equation 2.14.  $R_b$  refers to the nominal dynamic range of subband  $b$ , or the total number of bits required to represent the coefficient taking into consideration the gain bits added from DWT processing [7, pg 508].  $\mu_b$  and  $\epsilon_b$  are called the mantissa and exponent of subband  $b$ , respectively.  $\mu_b$  exists in the range  $0 \leq \mu_b < 2^{11}$  and  $\epsilon_b$  in the range  $0 \leq \epsilon_b < 2^5$  [15, pg 437]. The mantissa and exponent are specified in code-stream markers and are either specified for each subband (expounded quantization) or only for the  $LL_{N_L}$  subband (derived quantization). For more information regarding quantization, refer to [15, pg 436] and [22].

## 2.2.6 Embedded Block Coding With Optimized Truncation

After quantization the wavelet coefficients are entropy encoded (lossless). The overall manner in which data is entropy encoded in JPEG2000 is referred to as Embedded Block Coding With Optimized Truncation (EBCOT) [15, pg 333]. The computational intensity of EBCOT contributes to JPEG2000's substantially greater demand for processing resources compared to JPEG. Encoding comparisons have shown that JPEG2000 is on the order of thirty times more computationally intensive than baseline JPEG [1, pg 227]. The largest contributor to the processing demand is EBCOT's method of encoding in a stage called Tier I. The second portion of EBCOT is referred to as Tier II and is responsible for assembling the scalable code-stream.

Tier II processing can also include fine truncation of the bit-stream to obtain user specified file sizes.

### **Tier I: Candidate for Hardware Acceleration**

Tier I consists of two parts: a bit-plane level symbol generator and a binary arithmetic coder. These stages consume the most processing time in JPEG2000 encoding. A non-commercial single threaded JPEG2000 software encoder produces the timing results in Table 2.3 for compressing a 1k×1k color tile with a 10:1 compression ratio and 5 levels of wavelet decomposition. Tier I consumes 68.65% of the total processing time, making it a desirable target for processing acceleration.

Table 2.3: Software Timing Results for a 1k×1k Color Tile at 10:1 Compression

Function	Time (ms)	Percentage of Total Time (%)
DC Level Shift	4.421	0.94
Color Transform	5.353	1.14
DWT (5 decompositions)	95.421	20.39
Quantization	34.478	7.37
Tier I	321.256	68.65
Tier II	1.349	0.29
Misc Overhead	5.707	1.22
Total Time	467.985	-

Tier I is a feasible candidate for hardware acceleration due to the way JPEG2000 partitions tile data. Prior to Tier I encoding, the (non-)quantized wavelet coefficients are divided into sections called code-blocks. Typical code-block sizes are 64×64 or 32×32 samples, though image quality does improve with increasing code-block size [18]. Code-blocks do not span across subbands. Instead, if a code-block boundary does not line up exactly with a subband boundary, a partial code-block is used. Figure

2.10 shows how code-block boundaries are formed for *peppers*, where code-block and subband boundaries align. More details and restrictions regarding code-block sizes and boundaries can be found in [22].

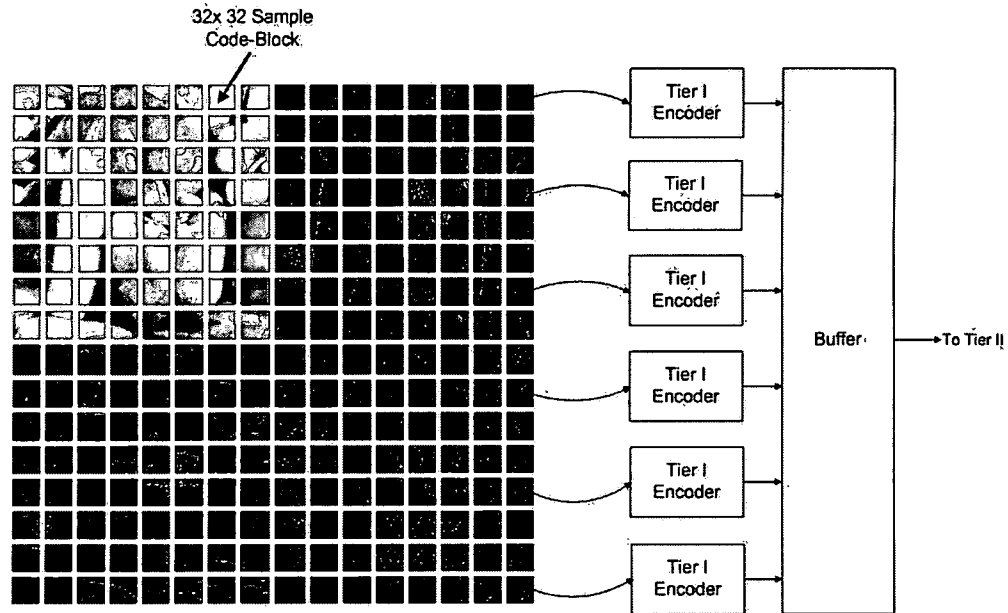


Figure 2.10: Code-block partitioning for *peppers* using a code-block size of  $32 \times 32$  samples. Each code-block can be sent to a different instance of a Tier I encoder.

Code-blocks are independently encoded, meaning code-blocks do not influence one another during Tier I processing. Not only does independent encoding aid in error robustness [15, pg 403], but in addition, code-blocks can be processed in parallel. The ability to encode sections of a tile simultaneously forms the basis of a hardware accelerated approach to JPEG2000 encoding. Using the well known parallel processing capabilities of FPGAs, a Tier I parallel encoding architecture is presented in Chapter 3.

## Tier I: Fractional Bit-plane Coding

Code-blocks can be “sliced” horizontally between each bit to create bit-planes. Figure 2.11 demonstrates the concept of bit-planes for the 8 bit grayscale *peppers* image (without any DWT processing). It can be seen, for example, that all of the most significance bits for each pixel are grouped into a two dimensional array—the most significant bit-plane. Bit-planes can be thought of as binary images.

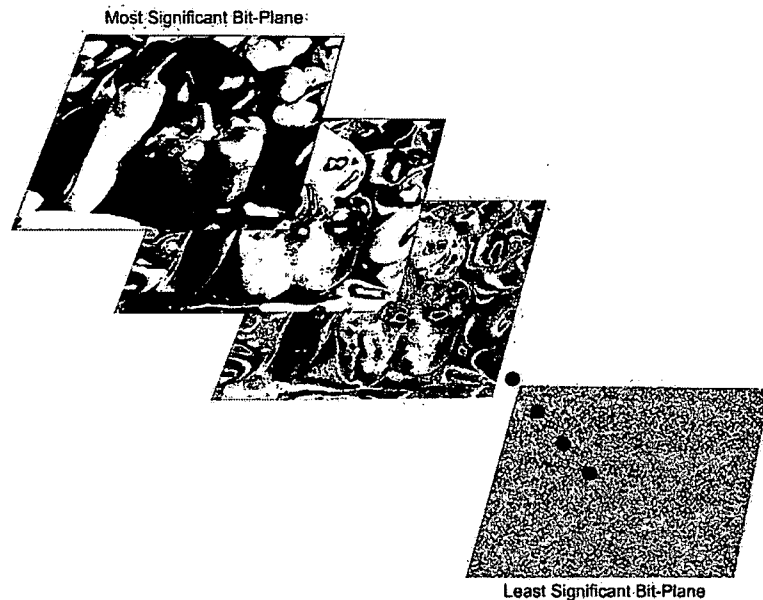


Figure 2.11: Bit-plane images for 8 bit grayscale *peppers*.

Before any bit-plane coding begins, the (non-)quantized wavelet coefficients must be converted to a sign-magnitude representation as opposed to twos complement representation. Scanning downward from the most significant magnitude bit-plane, the first non-all-zero bit-plane, referred to as  $K$ , is then found. Beginning with  $K$ , the bit-planes are processed by a sequence of coding passes—Significance Propagation Pass

(SPP), Magnitude Refinement Pass (MRP), and Cleanup Pass (CUP). The coding passes operate on bit-planes  $K:1$ , with bit-plane 1 referring to the least significant bit-plane [15, pg 485]. With the exception of bit-plane  $K$ , the order of coding pass processing is first the SPP, followed by the MRP, and the CUP last. On bit-plane  $K$  the CUP is the only coding pass to operate. While the SPP and MRP could operate on  $K$ , they would produce no output and hence only consume processing time [1, pg 172]. The purpose of the coding passes is to generate context labels and binary decision values as inputs for the binary arithmetic coder.<sup>3</sup> Each bit of a bit-plane is coded only once by a single coding pass, which is referred to as fractional bit-plane coding [1, pg 164]. Bits are processed in groups of four, called stripes. Figure 2.12 shows the order of stripe processing for each code-block and ultimately for each bit-plane. Bits within each stripe are processed from top to bottom.

Associated with the coding passes are three state variable tables— $\sigma$ ,  $\sigma_D$ , and  $\pi$ . These tables are used by the coding passes to track which bits have already been coded and how to assign the context and decision bit for the current stripe bit being processed. The size of the tables matches the code-block size, and bordering samples are assumed to be zero. All of the tables are initialized to zero when starting to process a code-block. State table  $\pi$  is re-initialized to zeros after each bit-plane is processed, while  $\sigma$  and  $\sigma_D$  are zeroed following completion of processing for the entire code-block. Locations within the context tables correlate directly with samples in the associated code-block. Therefore,  $\sigma(m,n)$ ,  $\sigma_D(m,n)$ , and  $\pi(m,n)$  all contain

<sup>3</sup>Documents may differ in the coding type they assign to context values. For example, one document may assign run-length coding to context 17 while another assigns it to 9. However, as long as the MQ coder operates consistently with the coding pass table (e.g. context 17 means run-length coding both to the coding passes and the MQ coder) there is no effect on the resulting bit-stream.

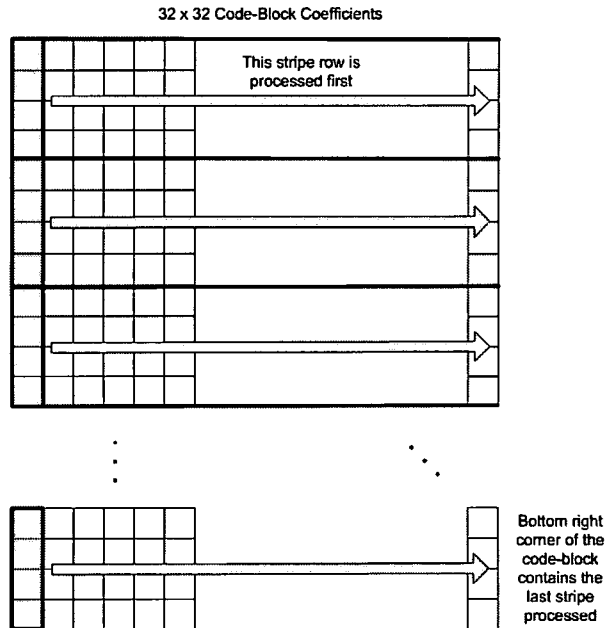


Figure 2.12: Depiction of the order stripes are processed for a code-block.

information corresponding to code-block location  $(m,n)$ .  $\sigma$  represents a coefficient's significance, that is, whether or not a one bit has been coded for the sample. When the first one of a coefficient is coded  $\sigma$  is set to one. If a bit has been coded in the MRP, then  $\sigma_D$  (delayed significance) is set to one for that sample. State variable  $\pi$  (coding pass membership) is set to one if zero coding has occurred for the current stripe bit during the SPP.

### Zero Coding

JPEG2000 fractional bit-plane coding consists of four possible coding modes—zero coding, run-length coding, sign coding, and magnitude refinement coding. Zero coding occurs in both the SPP and CUP only for bits that are not currently significant. Zero coding is initiated differently for the two passes, but the method regardless uses the significance values of the surrounding neighborhood of the bit (Figure 2.13) to

determine the context. Based on the subband of the code-block, the sum of the neighboring horizontal significance values, the sum of the neighboring vertical significance values, and the sum of all four neighboring diagonal significance values a context is determined. The tables for selecting the proper contexts are shown in Tables 2.4, 2.5, and 2.6, with  $X$  representing "don't care" values. The decision bit generated is equal to the value of the current stripe bit.

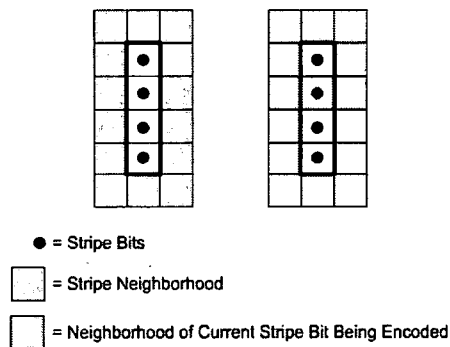


Figure 2.13: Depiction of a stripe neighborhood and the neighborhood of a stripe bit being coded.

Table 2.4: **LL/LH** Sub-band Zero Coding Context Lookup Table

Neighborhood Significance Sums			Context Label
Horizontal Sum	Vertical Sum	Diagonals Sum	CX
2	$X$	$X$	8
1	$\geq 1$	$X$	7
1	0	$\geq 1$	6
1	0	0	5
0	2	$X$	4
0	1	$X$	3
0	0	$\geq 2$	2
0	0	1	1
0	0	0	0

Table 2.5: **HL** Sub-band Zero Coding Context Lookup Table

Neighborhood Significance Sums			Context Label
Horizontal Sum	Vertical Sum	Diagonals Sum	CX
$X$	2	$X$	8
$\geq 1$	1	$X$	7
0	1	$\geq 1$	6
0	1	0	5
2	0	$X$	4
1	0	$X$	3
0	0	$\geq 2$	2
0	0	1	1
0	0	0	0

Table 2.6: **HH** Sub-band Zero Coding Context Lookup Table

Neighborhood Significance Sums		Context Label
Horizontal & Vertical Sum	Diagonals Sum	CX
$X$	$\geq 3$	8
$\geq 1$	2	7
0	2	6
$\geq 2$	1	5
1	1	4
0	1	3
$\geq 2$	0	2
1	0	1
0	0	0

### Run-Length Coding

Run-length coding occurs only in the CUP and most frequently in quantized high frequency subbands that are mostly zero valued. In the event that the CUP codes the first bit of a stripe, if the stripe has no significant bits as well as no significant bits

in its neighborhood (Figure 2.13), then run-length coding takes place. If the stripe contains all zero bits, a context of 17 and a decision bit of zero are generated. If the stripe contains any ones a context of 17 is generated along with a decision bit of one. The location of the first one bit in the stripe is coded as two decision bits, each with a context of 18. Because a stripe consists of only four bits, the indices of the bits can be represented by the integers zero through three. Therefore, to run-length code the stripe 0110, the (context, decision) pairs are (17,1), followed by (18,0), and (18,1) last. Figure 2.14 clarifies this example.

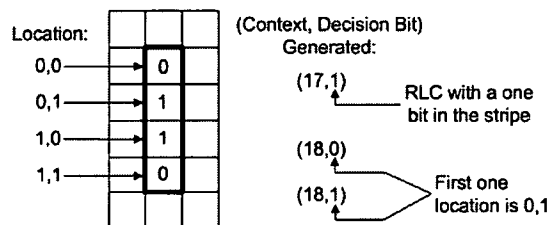


Figure 2.14: Run-length coding (RLC) example.

## Sign Coding

Sign coding occurs in the SPP or CUP, and only when a bit becomes significant. Sign coding, as its name implies, codes the sign of the sample with a context and decision bit. When a sign is coded, the significance and sign values of the associated stripe bit's horizontal and vertical neighbors are used to determine the context and another parameter called  $\hat{\chi}$ . Equations 2.15 and 2.16 show how the parameters are determined for referencing the sign lookup table provided in Table 2.7. The decision bit for sign coding is found by exclusively ORing the  $\hat{\chi}$  value with the sign bit of the stripe bit being coded (Equation 2.17).

$$H = \min[1, \max(-1, \sigma[m, n-1] \times (1 - 2\chi[m, n-1]) + \sigma[m, n+1] \times (1 - 2\chi[m, n+1]))] \quad (2.15)$$

$$V = \min[1, \max(-1, \sigma[m-1, n] \times (1 - 2\chi[m-1, n]) + \sigma[m+1, n] \times (1 - 2\chi[m+1, n]))] \quad (2.16)$$

$$\text{Decision Bit} = \hat{\chi} \otimes \chi \quad (2.17)$$

Table 2.7: Sign Coding Context and  $\hat{\chi}$  Lookup Table

H	V	$\hat{\chi}$	CX
1	1	0	13
1	0	0	12
1	-1	0	11
0	1	0	10
0	0	0	9
0	-1	1	10
-1	1	1	11
-1	0	1	12
-1	-1	1	13

### Magnitude Refinement Coding

The fourth type of coding used to generate contexts and decision bits is magnitude refinement coding, which only occurs in the MRP. Both the delayed significance state variable for the stripe bit being coded and the significance values of the stripe bit's eight neighbors are used to determine the context. Table 2.8 shows how the context is chosen. The decision bit is equal to the value of the stripe bit.

Table 2.8: Magnitude Refinement Context Lookup Table

$\sigma_D$	Sum of 8 Neighbor $\sigma$ 's	Context Label
1	$X$	16
0	$\geq 1$	15
0	0	14

### Significance Propagation Pass

The Significance Propagation Pass (SPP) first operates on bit-plane K-1. Figure 2.15 shows the block diagram for the SPP. For the current stripe bit, if its associated significance is zero ( $\sigma = 0$ ) and at least one of its eight neighbors has a significance of one ( $\sigma = 1$  for at least one neighbor), zero coding will commence; otherwise, the next bit of the stripe is examined. If the stripe bit is one, the coefficient's sign bit is then coded and the coefficient is marked as significant (set  $\sigma = 1$ ).

### Magnitude Refinement Pass

The Magnitude Refinement Pass (MRP), like the SPP, also first operates on bit-plane K-1. Figure 2.16 shows the block diagram for the MRP. For the current stripe bit, if it is significant ( $\sigma = 1$ ) and has not already been zero coded by the SPP ( $\pi = 0$ ), the bit undergoes magnitude refinement coding; otherwise, the next bit of the stripe is examined. The delayed significance state variable is also set to one ( $\sigma_D = 1$ ) to indicate magnitude refinement coding has occurred for the coefficient. The next bit of the stripe is then examined.

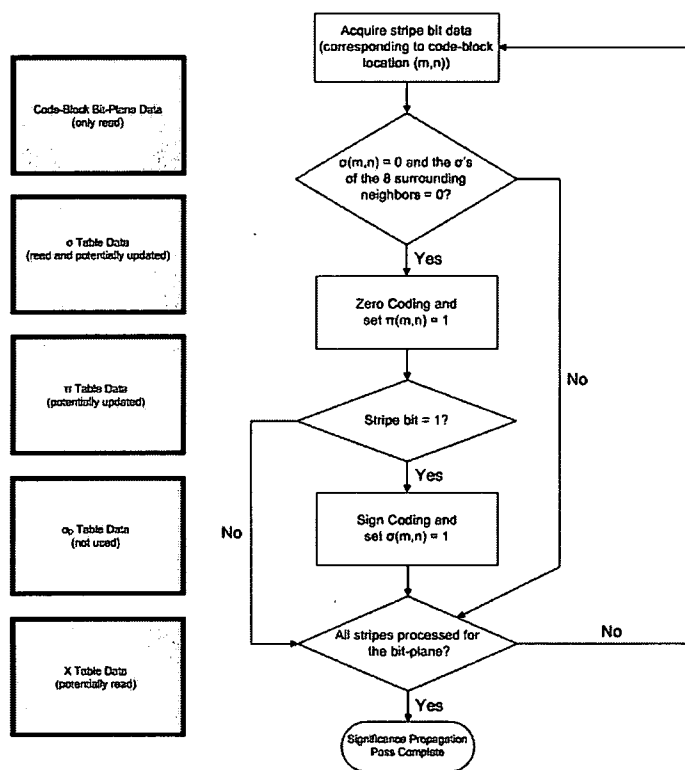


Figure 2.15: Block diagram for the Significance Propagation Pass.

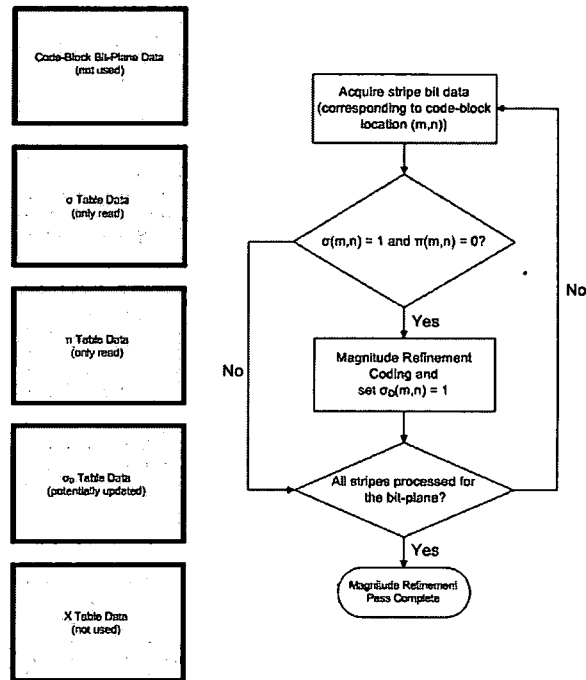


Figure 2.16: Block diagram for the Magnitude Refinement Pass.

### Cleanup Pass

The Cleanup Pass (CUP) operates on bit-planes  $K:1$ , where bit-plane 1 represents the least significant bit-plane. Figure 2.17 shows the block diagram for the CUP. If the current stripe bit is not significant ( $\sigma = 0$ ) and has not already been zero coded by the SPP ( $\pi = 0$ ), the bit is subject to zero coding or run-length coding; otherwise, the next bit of the stripe is examined. If the bit is the top bit in the stripe<sup>4</sup> and all of the significance values of the stripe and its neighbors are zero ( $\sigma$ 's = 0), the stripe is run-length coded; otherwise, the bit is subject to zero coding. If the bit has not already been included in run-length coding it will be zero coded; otherwise, the next bit of the stripe is examined. Following run-length coding and zero coding, if the

<sup>4</sup>This only true for full stripes, that is, those that are not cut off by subband boundaries.

coded bit is a one the coefficient's sign is coded and its significance is set to one ( $\sigma = 1$ ); otherwise the next bit of the stripe is examined.

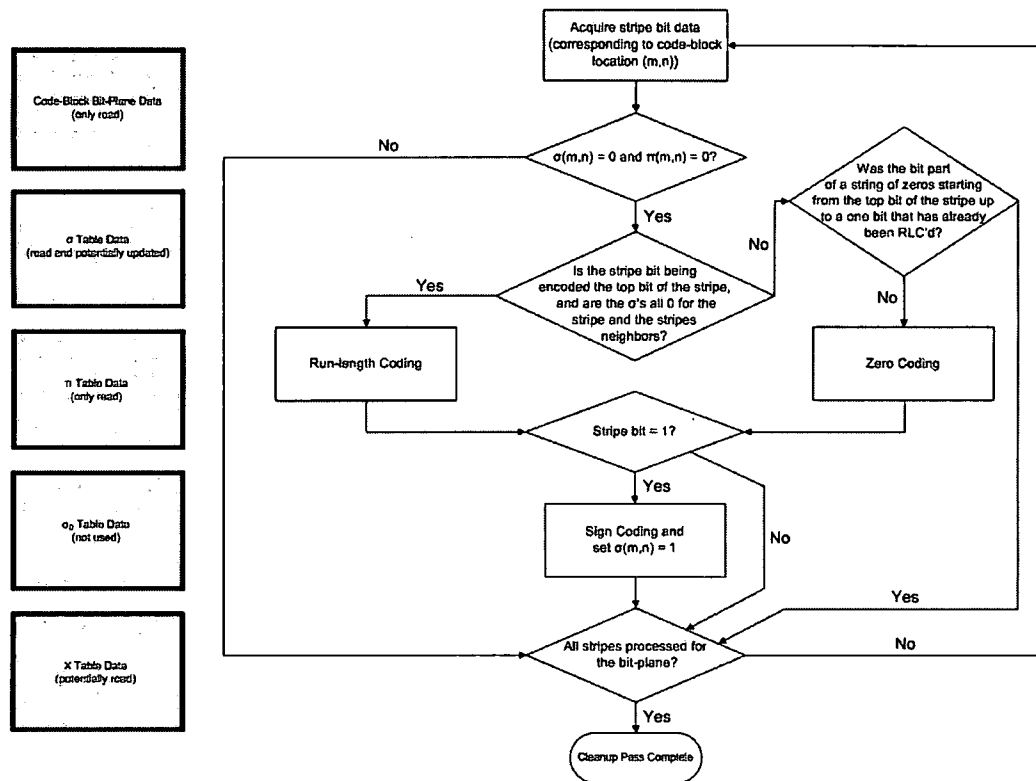


Figure 2.17: Block diagram for the Cleanup Pass.

## MQ Coder

The MQ coder constitutes the other major portion of Tier I and uses the output of the coding passes to generate bytes representing the compressed bit-stream. The MQ coder is a binary arithmetic entropy coder, meaning it losslessly encodes a set of symbols. For JPEG2000 compression the symbols are the coding pass decision bits—either a one or a zero. The contexts generated by the coding passes determine

whether a decision bit is a more probable symbol (MPS) or a less probable symbol (LPS). As outlined in [1, pg 40], in a binary image region of mostly white pixels the MPS is one. Conversely, in a binary image region of mostly black pixels the LPS is one. The contexts are used to determine whether ones or zeros are the MPS based on probabilities from a lookup table. The documentation for the hardware implementation of the MQ coder used in the proposed Tier I hardware design describes how the MQ coder uses MPS's and LPS's in generating compressed bytes. Refer to [5, 11] for more detailed documentation regarding the underlying workings of arithmetic coding and the JPEG2000 MQ coder.

## **Tier II**

Tier II is responsible for assembling the final JPEG2000 file. This process includes organizing the MQ coder output bytes according to the user specified progression method; for example, by resolution, quality, component, and spatial region (areas called precincts). The organization changes how the file is parsed and can have significance in image distribution applications. Further details of JPEG2000 file and header structures are beyond the scope of this documentation. Refer to [15, 22] for complete descriptions of JPEG2000 file organization.

## **CHAPTER 3**

### **JPEG2000 Tier I Hardware Architecture**

The objective of this thesis is to develop a JPEG2000 Tier I parallel architecture for FPGA implementation, with the target application of a wide area persistent surveillance system. Chapter 3 describes the hardware architecture from a functional perspective—No source code is included for proprietary reasons. Nevertheless, details are provided as to how Tier I is broken down modularly, as well as how the modules interact with one another. The descriptions and diagrams provide clear insight to the methodology used to build a functional Tier I compression engine. All of the JPEG2000 processing up to the Tier I encoding is completed in software, so the data sent to the FPGAs consists of quantized wavelet coefficients. The outputs of the encoders are the compressed code-blocks, which are sent back to the software for Tier II organization and construction of the final JPEG2000 file.

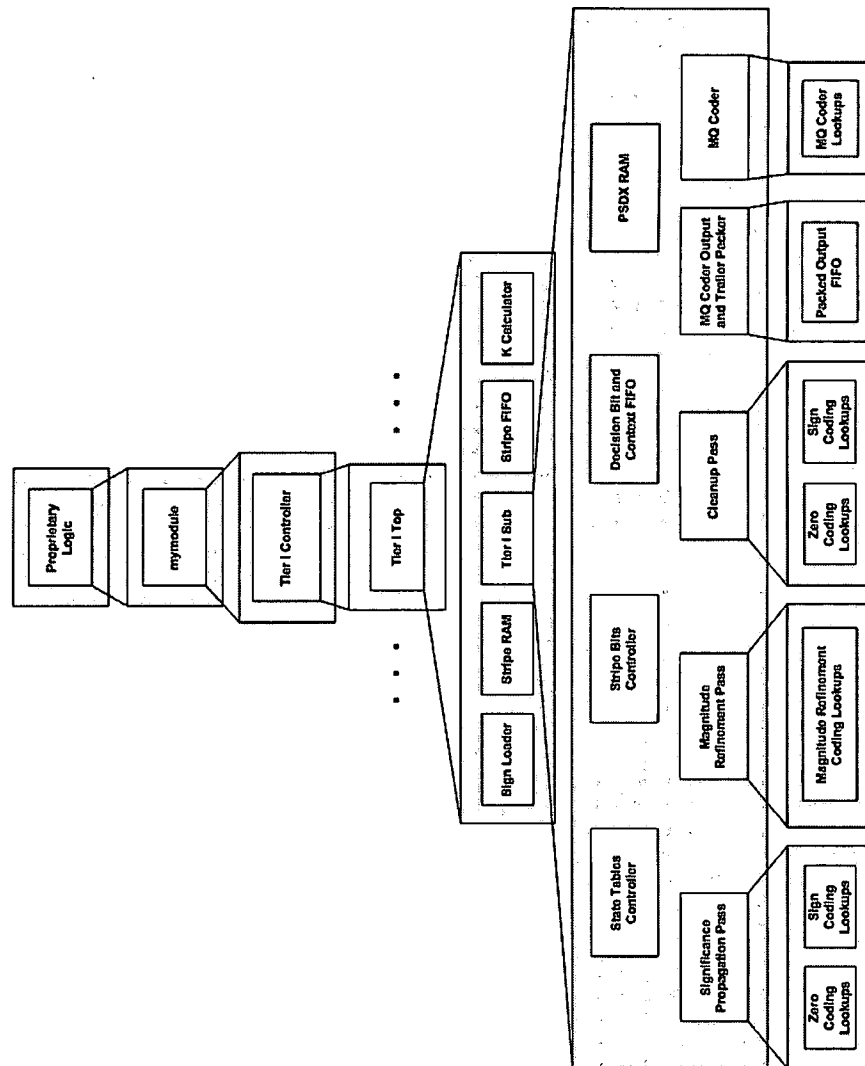


Figure 3.1: Hierarchy of the JPEG2000 Tier I encoder architecture.

### 3.0.7 Top Level

The hierarchy for the Tier I hardware is summarized in Figure 3.1, shown encompassed under the Hardware Description Language (HDL) module *mymodule*. The level of hierarchy above *mymodule* represents non-user logic and is dictated by the architecture of the platform with which the hardware compression system is integrated. The addition of a Peripheral Component Interconnect Express (PCIe) based FPGA board to the target wide area surveillance platform allows for minimal disturbance to the rest of the existing image capture/compression software system (e.g. no processors are replaced by FPGA add-on boards). To maximize utilization of the memory and PCIe bandwidths for host and FPGA board transactions, Direct Memory Access (DMA) transfers are invoked using an Application Programming Interface (API) provided by the FPGA board manufacturer. Because the image data is sent “in bulk” to the FPGA, it is buffered in a large bank of memory easily accessible to the FPGA. On-chip FPGA memory is consumed for Tier I cores and otherwise is not abundant enough to hold significant amounts of image data, so the data is buffered in external memory. As a result the Tier I cores begin processing new code-blocks immediately after encoding a set of data. The objective is to keep the Tier I cores from being starved for new data to minimize compression time. In the proposed design, the top-most level (shown as *Proprietary Logic*) is a memory controller provided by the FPGA board manufacturer, which provides an interface to use with the external FPGA memory. While interfacing to a vendor specific controller introduces FPGA board dependencies to the design, the Tier I encoders have been structured such that feeding or retrieving data is relatively generic.

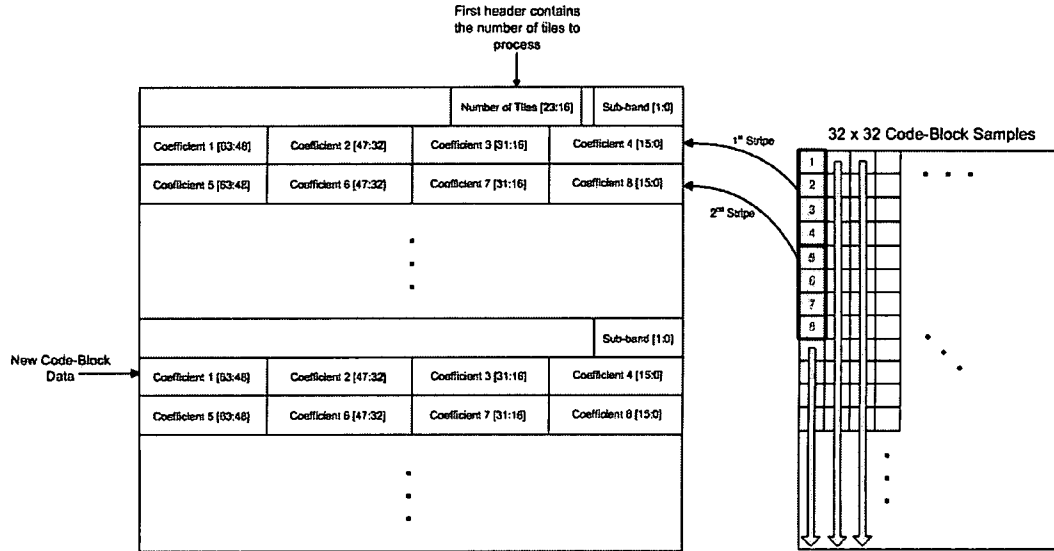


Figure 3.2: Depiction of how code-blocks are stored in the *Primary Input FIFO*.

The proprietary memory controller is configured to treat the external memory as very large FIFOs (First-In-First-Out modules). The proposed design uses one of these FIFOs for uncompressed tile data (*Primary Input FIFO*) and another for compressed output (*Primary Output FIFO*) for each FPGA. On the input side, because DMA transfers are used to send multiple tiles at once, all of the code-blocks are lumped together in the *Primary Input FIFO*. A method for determining where one code-block ends and another begins is required. This is currently accomplished by using a fixed code-block size of  $32 \times 32$  samples. The code-blocks are sent to the FPGA board such that a header precedes the wavelet coefficients, and the wavelet coefficients are stored in groups of four samples, called stripes. A  $32 \times 32$  sample code-block contains 1024 samples, so storing samples in groups of four necessitates 256 stripes being read from the *Primary Input FIFO* by each Tier I core, assuming each core operates on a code-block. Figure 3.2 shows how the header and wavelet coefficient data is stored in the

*Primary Input FIFO*. A counter is decremented as 256 stripes are sent to a Tier I core. When the stripe counter reaches zero it is known that the next element in the *Primary Input FIFO* holds the header for the following code-block. The code-block headers only contain the subband of the code-block, with the exception of the very first header which also contains the number of tiles to process. The proposed implementation follows the logic of the state machine shown in Figure 3.3 for reading in code-block data and resetting the system to permit more tiles to be sent. Control signals for writing compressed output to the *Primary Output FIFO* are modified in a stand-alone process so that data is written to the *Primary Output FIFO* independently, even if another Tier I core is loading a new code-block.

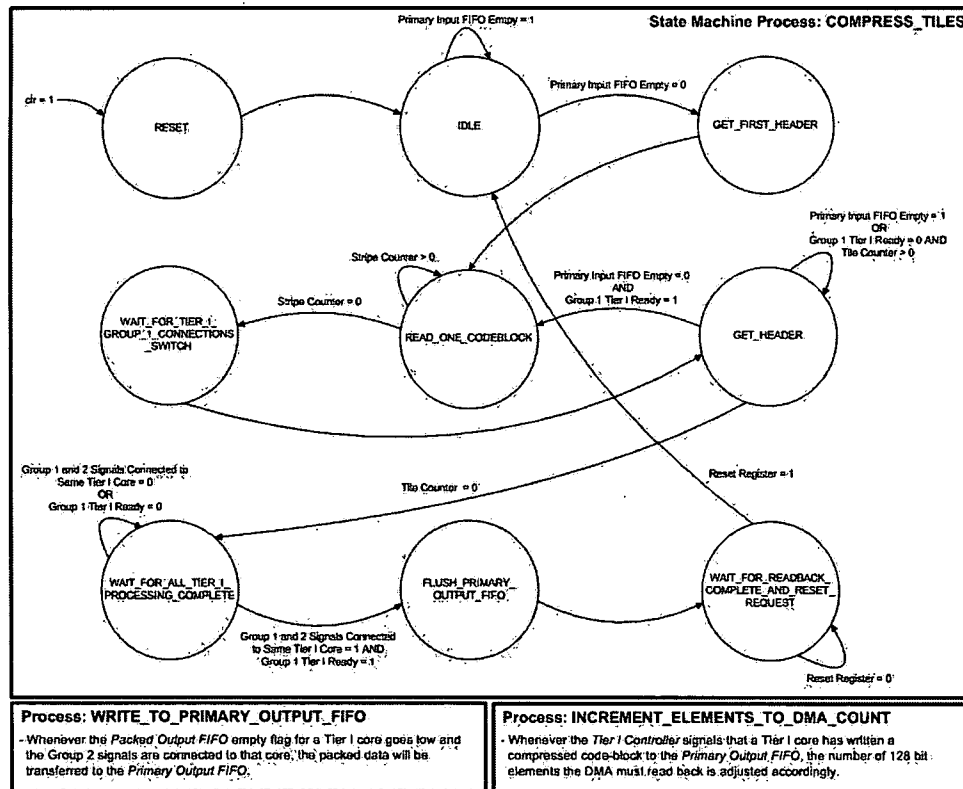


Figure 3.3: *mymodule* state machine and other processes.

### *mymodule* COMPRESS\_TILES State Machine

The following discussion provides an overview of the *mymodule* state machine.

The states are:

RESET,  
IDLE,  
GET\_FIRST\_HEADER,  
READ\_ONE\_CODE\_BLOCK,  
WAIT\_FOR\_TIER\_I\_GROUP\_1\_CONNECTIONS\_SWITCH,  
GET\_HEADER,  
WAIT\_FOR\_ALL\_TIER\_I\_PROCESSING\_COMPLETE,  
FLUSH\_PRIMARY\_OUTPUT\_FIFO,  
WAIT\_FOR\_READBACK\_COMPLETE\_AND\_RESET\_REQUEST.

#### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where all signals are initialized.

#### **IDLE**

This state waits for the software to send a status signal indicating all of the tiles have been transferred to the *Primary Input FIFO*. The eventual goal is to have data read from the *Primary Input FIFO* whenever it is not empty.

#### **GET\_FIRST\_HEADER**

This state reads one element from the *Primary Input FIFO* and extracts the number of tiles to process (bits 23:16 of the header). The subband (bits 1:0 of the header) is passed to the *Tier I Controller*, which passes it to the Tier I core the *Tier I Controller* is currently connected to. The position of the number of tiles information in the header is due to the 16 bit stripe coefficients sent by the DMA. Data exists in each DMA buffer at 16 bit boundaries because all of the quantized wavelet coefficients are allowed up to 16 bits of precision (signed). Because the headers are included in

the DMA transfer, putting the number of tiles at a 16 bit boundary trivializes the software that inserts the first header into the DMA buffer.

#### **READ\_ONE\_CODE\_BLOCK**

This state monitors a counter as 256 elements (one code-block) are read from the *Primary Input FIFO*. One element consists of four code-block samples (one stripe). All of the stripes are passed to the *Tier I Controller*, which loads a *Stripe FIFO* for a Tier I core.

#### **WAIT\_FOR\_TIER\_I\_GROUP\_1\_CONNECTIONS\_SWITCH**

This state allows the Group 1 signals within the *Tier I Controller* (Figure 3.4) to switch to another Tier I core once 256 stripes have been passed to the Tier I core currently connected to the *Primary Input FIFO*. Group 1 signals are those used when a code-block is loaded into a Tier I core. All signals in Group 1 switch concurrently as do the signals in Group 2. Group 1 signals include subband data, stripe data, a stripe ready-to-load flag, and a Tier I idle flag. The *Stripe FIFO* full flag is also included in the Group 1 signals, but it is only used internally to the *Tier I Controller* to determine when to switch the Group 1 signals to the next Tier I core. Group 2 signals are those used when a compressed code-block is sent to the *Primary Output FIFO* and the sending Tier I core is reset. Group 2 signals include the final compressed data packed into 128 bit elements, the *Packed Output FIFO* status flags and controls, the number of packed output elements written into the *Primary Output FIFO* for the code-block, and the reset register for the Tier I core. Figure 3.4 shows all of the multiplexors and demultiplexors combined into one for the signal groups to simplify the illustration.

## GET\_HEADER

This state reads a header from the *Primary Input FIFO*, which only contains the code-block's subband. This state always transitions to the READ\_ONE\_CODE\_BLOCK state until there are no more tiles to process, a case in which the state machine enters the WAIT\_FOR\_ALL\_TIER\_I\_PROCESSING\_COMPLETE state. Tile boundaries within the *Primary Input FIFO* are known by requiring a fixed tile size of  $1k \times 1k$  samples. A  $1k \times 1k$  tile consists of  $1024 \cdot 32 \times 32$  code-blocks, so a code-block counter is used to detect when an entire tile has been read from the *Primary Input FIFO*.

## WAIT\_FOR\_ALL\_TIER\_I\_PROCESSING\_COMPLETE

This state waits for all of the Tier I processing to complete after the last code-block has been loaded into a Tier I core. The Tier I cores are determined to all be idle when the *Tier I Controller's* Group 1 and Group 2 signals are all connected to the same Tier I core and that core is idle.

## FLUSH\_PRIMARY\_OUTPUT\_FIFO

This state asserts a flush signal for the *Primary Output FIFO*, which is required by the proprietary memory controller before the software read back.

## WAIT\_FOR\_READBACK\_COMPLETE\_AND\_RESET\_REQUEST

This state monitors a register set by the software after all of the data in the *Primary Output FIFO* has been read back. The software empties the *Primary Output FIFO* once the hardware updates a control register with the number of 128 bit elements the DMA must read from the *Primary Output FIFO*. Once this register is non-zero the software requests that the compressed data be read back. When the

Primary Output FIFO is empty the software signals that it has obtained all of the compressed data, and the state machine returns to the IDLE state.

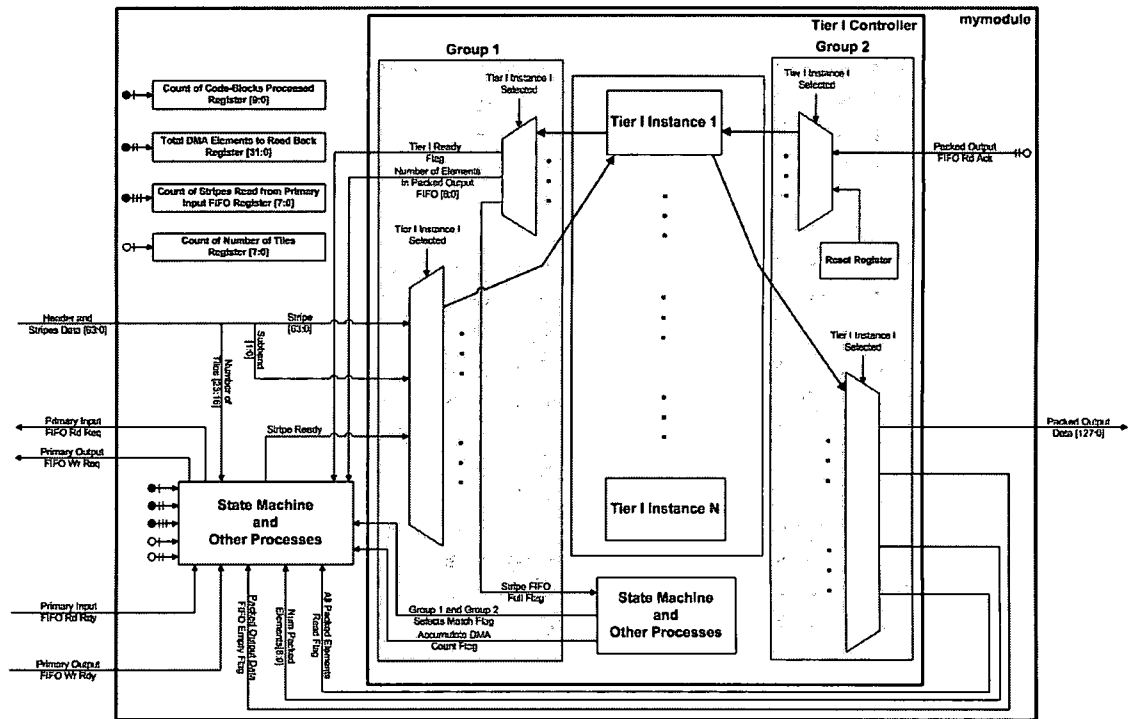


Figure 3.4: *mymodule* and *Tier I Controller* interface diagram with multiplexors and demultiplexors combined for Group 1 and 2 signals.

## Other Processes of *mymodule*

The following discussion provides an overview of the processes of *mymodule* other than the state machine.

The processes are:

WRITE\_TO\_PRIMARY\_OUTPUT\_FIFO,  
INCREMENT\_ELEMENTS\_TO\_DMA\_COUNT.

### WRITE\_TO\_PRIMARY\_OUTPUT\_FIFO

This process controls the *Primary Output FIFO* write request signal, so that data from the Tier I core currently connected to the Group 2 signals is written to the *Primary Output FIFO*. The process writes data to the *Primary Output FIFO* while the Tier I core's *Packed Output FIFO* is not empty. It also asserts the read acknowledge signal of the *Packed Output FIFO*.

As with the *Primary Input FIFO* for each FPGA, a situation of code-block data separation exists within the *Primary Output FIFO* since the compressed code-blocks are all loaded into the same FIFO. A trailer is inserted after each compressed code-block to make code-block data distinguishable. Specifics of the trailers will be discussed in the Section 3.0.21 for the *MQ Coder Output and Trailer Packer*.

### INCREMENT\_ELEMENTS\_TO\_DMA\_COUNT

This process keeps a running sum of the number of 128 bit elements written to the *Primary Output FIFO*. When a Tier I core finishes writing all of its compressed code-block data to the *Primary Output FIFO*, the number of elements it has written is passed to this process and added to the count.

### 3.0.8 Tier I Controller

The *Tier I Controller* is responsible for managing the input and output connections of the Tier I cores, primarily controlling which Tier I core is permitted to read from the *Primary Input FIFO* and which is able to write to the *Primary Output FIFO*. In addition, a user specified parameter is present in the source code to set the number of Tier I cores instantiated on an FPGA. This non-laborious replication of Tier I cores within an FPGA is accomplished through apt use of the *generate* statement in VHDL.

The *Tier I Controller* also performs the critical function of resetting the Tier I cores. A very simple state machine (Figure 3.5) is used for resetting the Tier I cores upon system initialization. The state machine also resets individual Tier I cores after their compressed code-block is sent to the *Primary Output FIFO*. In addition, the state machine adjusts which Tier I core is connected to the Group 2 signals.

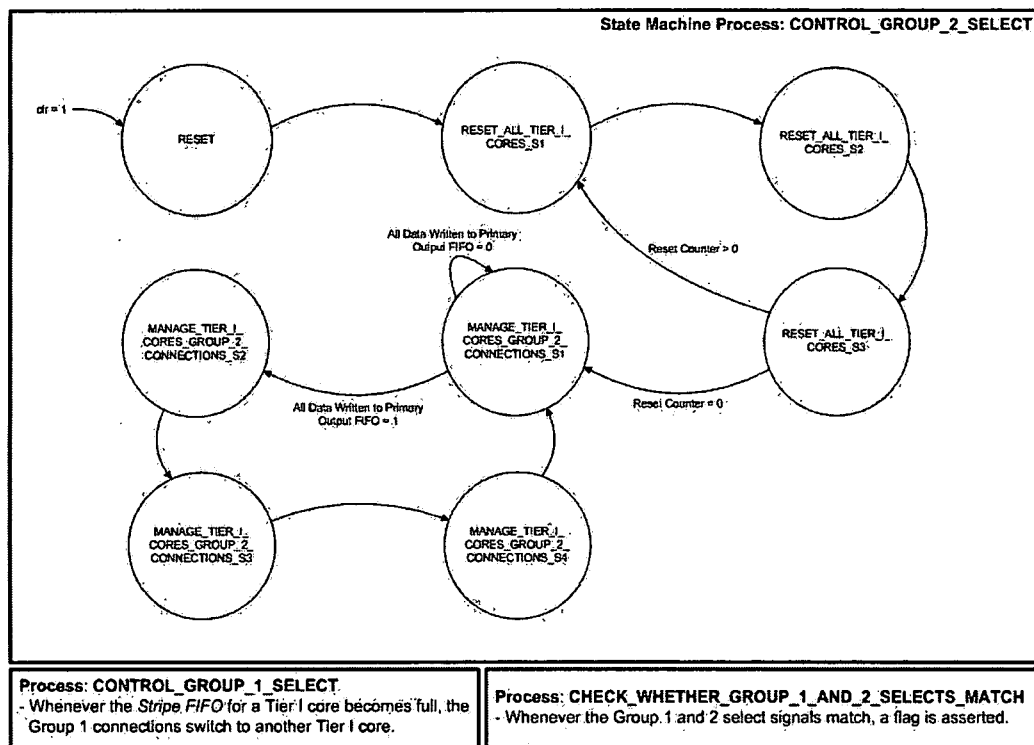


Figure 3.5: *Tier 1 Controller* state machine and other processes.

### ***Tier I Controller* CONTROL\_GROUP\_2\_SELECTS State Machine**

The following discussion provides an overview of the *Tier I Controller* state machine.

The states are:

RESET\_ALL\_TIER\_I\_CORES\_S1,  
RESET\_ALL\_TIER\_I\_CORES\_S2,  
RESET\_ALL\_TIER\_I\_CORES\_S3,  
MANAGE\_TIER\_I\_CORES\_GROUP\_2\_CONNECTIONS\_S1,  
MANAGE\_TIER\_I\_CORES\_GROUP\_2\_CONNECTIONS\_S2,  
MANAGE\_TIER\_I\_CORES\_GROUP\_2\_CONNECTIONS\_S3,  
MANAGE\_TIER\_I\_CORES\_GROUP\_2\_CONNECTIONS\_S4.

#### **RESET\_ALL\_TIER\_I\_CORES\_S1:3**

Upon system initialization, three reset states are cycled between as each Tier I core is independently initialized. Independent reset capability is crucial because it means all of the Tier I cores on the FPGA do not have to finish processing a code-block before idle cores can begin processing again. After a Tier I core sends its data to the *Primary Output FIFO*, it is reset and loads another code-block once the Group 1 signals are connected to that core. As stated in the previous section, Group 1 signals are those used when a code-block is loaded into a Tier I core. Group 2 signals are those used when a compressed code-block is sent to the *Primary Output FIFO* and the sending Tier I core is reset.

The RESET\_ALL\_TIER\_I\_CORES\_S1 state asserts the reset signal for the Tier I core currently connected to the Group 2 signals. The RESET\_ALL\_TIER\_I\_CORES\_S2 state de-asserts the reset signal. The RESET\_ALL\_TIER\_I\_CORES\_S3 state allows the Group 2 signals to switch connections to the next Tier I instance, so the reset

register will affect the newly connected Tier I core. The *Tier I Controller* state machine determines when it has reset all of the Tier I cores by using a down counter initialized to the total number of Tier I cores on the FPGA.

#### **MANAGE\_TIER\_I\_CORES\_GROUP\_2\_CONNECTIONS\_S1:4**

When the Tier I core currently connected to the Group 2 signals indicates that it has emptied all of its compressed data into the *Primary Output FIFO*, a flag is raised to indicate to *mymodule* that it should add the number of 128 bit elements the Tier I core just inserted into the *Primary Output FIFO* to its running count. The flag is asserted in the MANAGE\_TIER\_I\_CORES\_GROUP\_2\_CONNECTIONS\_S1 state. The MANAGE\_TIER\_I\_CORES\_GROUP\_2\_CONNECTIONS\_S2 state asserts the Tier I core's reset signal. The MANAGE\_TIER\_I\_CORES\_GROUP\_2\_CONNECTIONS\_S3 state de-asserts the reset signal. Finally, the MANAGE\_TIER\_I\_CORES\_GROUP\_2\_CONNECTIONS\_S4 state changes the select signal for the Group 2 connections.

#### **Other Processes of *Tier I Controller***

The following discussion provides an overview of the processes of *Tier I Controller* other than the state machine.

The processes are:

CONTROL\_GROUP\_1\_SELECT,  
CHECK\_WHETHER\_GROUP\_1\_AND\_2\_SELECTS\_MATCH.

#### **CONTROL\_GROUP\_1\_SELECT**

After initial reset the Group 1 and Group 2 signals of the *Tier I Controller* are all connected to the first Tier I core, and the first code-block is read into that Tier I core. The Group 2 signal connections do not change, but the Group 1 connections switch

to the second Tier I core so the code-block stripes can be loaded immediately. The Group 1 connections will switch again to the third Tier I core after loading another code-block, and so on. Loading Tier I cores with code-block stripes in rapid succession allows code-block processing to overlap, helping to achieve an improved overall compression throughput. The `CONTROL_GROUP_1_SELECT` process monitors the *Stripe FIFO* full flag of the Tier I core currently connected to the Group 1 signals. When the full flag is asserted the Group 1 select is changed so that another Tier I core can be loaded from the *Primary Input FIFO*.

#### **CHECK\_WHETHER\_GROUP\_1\_AND\_2\_SELECTS\_MATCH**

This process asserts a flag when the Group 1 and 2 signals are connected to the same Tier I core. The flag is used by *mymodule* to determine when all of the code-block processing is complete for the last tile. Tier I processing for a set of tiles is complete when the counter for the number of tiles to compress reaches zero, the Group 1 and Group 2 signals are all connected to the same Tier I core, and that Tier I core is idle.

### **3.0.9 Tier I Top**

*Tier I Top* constitutes a single instance of a Tier I encoder. The primary function of the *Tier I Top* module is to coordinate pre-processing operations: registering the current code-block's subband, loading stripes into the *Stripe FIFO*, loading the stripe coefficient sign bits into the *PSDX RAM*, determining the most significant bit-plane containing a one bit, and moving stripes into the *Stripe RAM*. Figure 3.6 shows



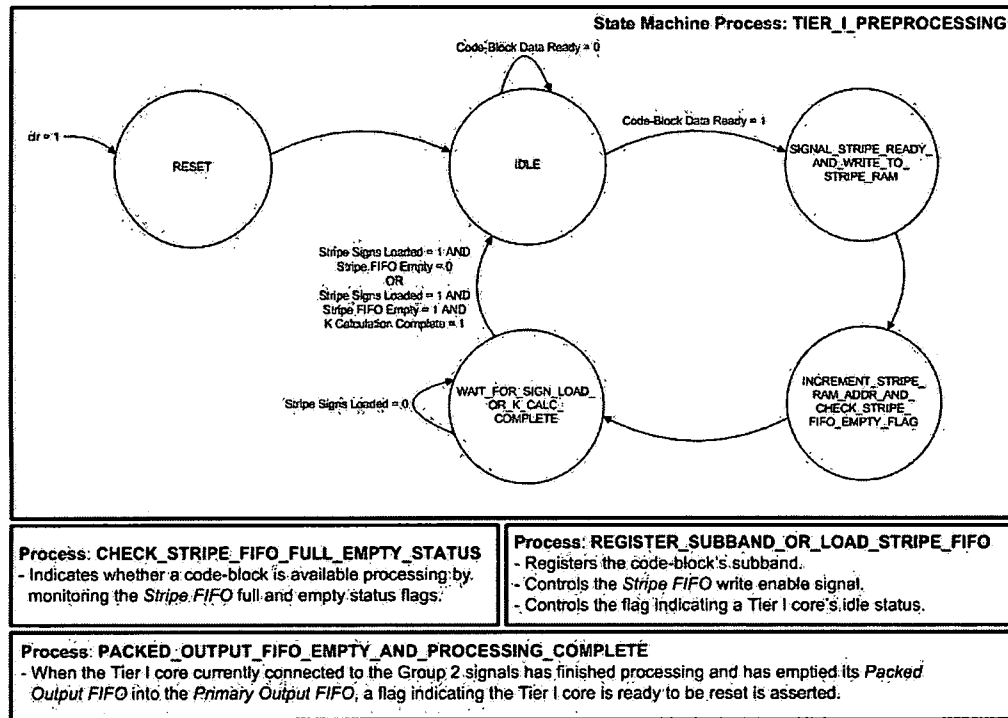


Figure 3.7: *Tier I Top* state machine and other processes.

### ***Tier I Top* TIER\_I\_PREPROCESSING State Machine**

The following discussion provides an overview of the *Tier I Top* state machine.

The states are:

RESET,  
 IDLE,  
 SIGNAL\_STRIPE\_READY\_AND\_WRITE\_TO\_STRIPE\_RAM,  
 INCREMENT\_STRIPE\_RAM\_ADDR\_AND\_CHECK\_STRIPE\_FIFO\_EMPTY\_FLAG,  
 WAIT\_FOR\_SIGN\_LOAD\_OR\_K\_CALC\_COMPLETE.

#### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where the *Stripe RAM* address and all status signals are initialized.

## **IDLE**

This state monitors a flag set by the CHECK\_STRIPE\_FIFO\_FULL\_EMPTY-STATUS process, which determines whether code-block stripes are available in the *Stripe FIFO*. Once an entire code-block is present, it is read from the *Stripe FIFO*. This state asserts the read request signal for the *Stripe FIFO* if code-block pre-processing has begun.

## **SIGNAL\_STRIPE\_READY\_AND\_WRITE\_TO\_STRIPE\_RAM**

This state asserts a status signal for the *Sign Loader* and *K Calculator* to indicate that a new stripe has just been read from the *Stripe FIFO*. In addition, the write request signal for the *Stripe RAM* is asserted.

## **INCREMENT\_STRIPE\_RAM\_ADDR\_AND\_CHECK\_STRIPE\_FIFO\_EMPTY\_FLAG**

This state increments the *Stripe RAM* address as well as checks whether the *Stripe FIFO* is empty. If the *Stripe FIFO* is empty, a status signal is asserted for the *K Calculator* to indicate that bit-plane K should be output.

## **WAIT\_FOR\_SIGN\_LOAD\_OR\_K\_CALC\_COMPLETE**

This state waits for the *Sign Loader* to signal that it has loaded the sign bits for the current stripe into the *PSDX RAM*. If the *Stripe FIFO* is not empty the state machine returns to the IDLE state and continues pre-processing until the *Stripe FIFO* is empty. Once the *Stripe FIFO* is empty, the sign bits are loaded for last stripe, and K is returned in the range 15:1, the *Tier I Sub* module is signalled to begin processing the code-block in the *Stripe RAM*. If the *K Calculator* returns a K value of 0, it means the code-block contains all zeros. The *MQ Coder Output and Trailer Packer* is then signalled to insert a trailer into the *Packed Output Data FIFO* indicating an all-zero code-block is present.

### Other Processes of *Tier I Top*

The following discussion provides an overview of the processes of *Tier I Top* other than the state machine.

The processes are:

CHECK\_STRIPE\_FIFO\_FULL\_EMPTY\_STATUS,  
REGISTER\_SUB\_BAND\_OR\_LOAD\_STRIPE\_FIFO,  
PACKED\_OUTPUT\_FIFO\_EMPTY\_AND\_PROCESSING\_COMPLETE.

#### CHECK\_STRIPE\_FIFO\_FULL\_EMPTY\_STATUS

This process asserts a flag to indicate that an entire code-block has been written to the *Stripe FIFO*. The flag remains high until the *Stripe FIFO* is empty, which means the entire code-block has undergone pre-processing operations.

#### REGISTER\_SUBBAND\_OR\_LOAD\_STRIPE\_FIFO

This process registers the subband for a code-block, since it has been predetermined to be the first piece of data a Tier I core receives. Subsequent data arrivals to *Tier I Top* are written to the *Stripe FIFO*. This process controls the write enable signal for the *Stripe FIFO* as well as the idle signal for the Tier I core. Once a subband is received the Tier I core is no longer idle and does not become idle again until the core is reset.

#### PACKED\_OUTPUT\_FIFO\_EMPTY\_AND\_PROCESSING\_COMPLETE

This process controls the flag indicating whether the Tier I core is ready to be reset. If the code-block encoding is complete and the *Packed Output FIFO* has been emptied into the *Primary Output FIFO*, the Tier I core is ready for reset. In addition, when *mymodule* sees the flag asserted it updates the number of 128 bit elements being tracked in the *Primary Output FIFO*.

### 3.0.10 Stripe FIFO

Each Tier I core has a *Stripe FIFO* for buffering a code-block. Because all of the Tier I cores for an FPGA receive data from the same external memory, the cores can only be supplied with code-blocks one at a time. Therefore, providing individual cores with data at as high a rate as possible is important for maximizing parallel processing benefits. The use of FIFOs allows the proprietary memory controller interfaced with the external memory to provide Tier I cores with code-block stripes at its maximum rate, independent of how fast the Tier I cores can process the data. Once a *Stripe FIFO* is full the sign bits of all the stripe coefficients are loaded into the *PSDX RAM* and bit-plane K is found. The stripes are also loaded into a *Stripe RAM* so that they can be accessed in non-successive order. Ideally, the read clock of a *Stripe FIFO* is the same clock as the rest of the Tier I core but is different than whatever bus clock fills the *Stripe FIFO*, since the write clock should be much faster than a Tier I core can operate at. In the current implementation of the Tier I architecture, the *Stripe FIFO* read and write clocks are the same. Figure 3.8 shows how a code-block is stored in the *Stripe FIFO*. The organization of coefficients is identical to that shown in Figure 3.2 minus the headers.

### 3.0.11 Stripe RAM

Figure 3.8 shows that code-block stripes are stored in the *Stripe FIFO* column by column. However, Tier I processes stripes from adjacent columns as shown in Figure 2.12. The *Stripe FIFO* does not receive stripes in the proper order because the software avoids memory reorganization before the DMA transfer to reduce software processing time. Using on-chip FPGA RAM allows access to any code-block stripe,

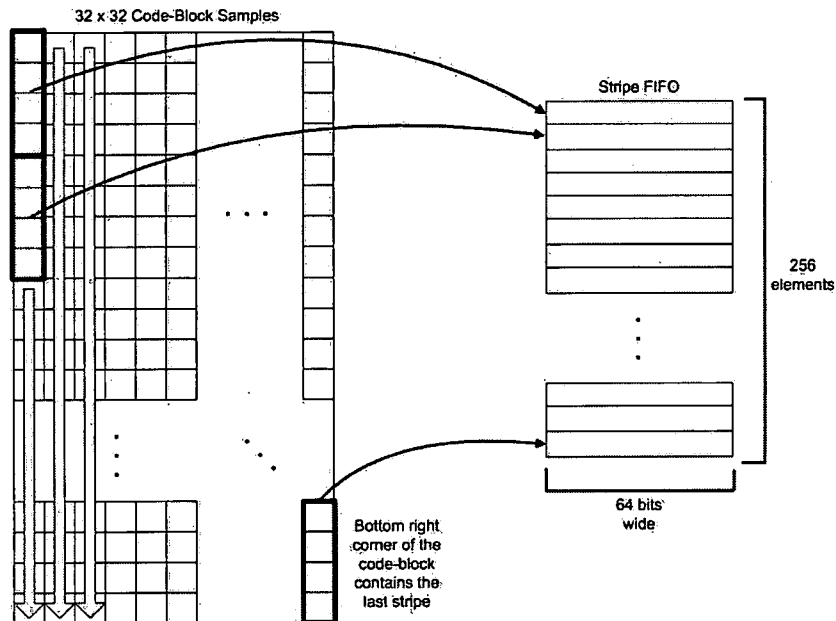


Figure 3.8: Depiction of how code-block stripes are stored in the Tier I *Stripe FIFOs*.

so the *Stripe FIFO* is essentially copied into the *Stripe RAM*. Like the *Stripe FIFO*, the *Stripe RAM* is configured to be 64 bits wide and hold 256 elements.

### 3.0.12 K Calculator

The *K Calculator* finds the first bit-plane of a code-block containing a one bit, beginning from the most significant bit-plane. The methodology for finding this bit-plane, denoted as bit-plane K, is to bit-wise OR all of the incoming stripe coefficients together and locate the most significant one bit in the final register. A K value of 15 indicates a one bit in the most significant bit-plane, while a value of 0 indicates the presence of an all-zero code-block. Figure 3.9 depicts how K is determined, and Figure 3.10 shows the *K Calculator* state machine.

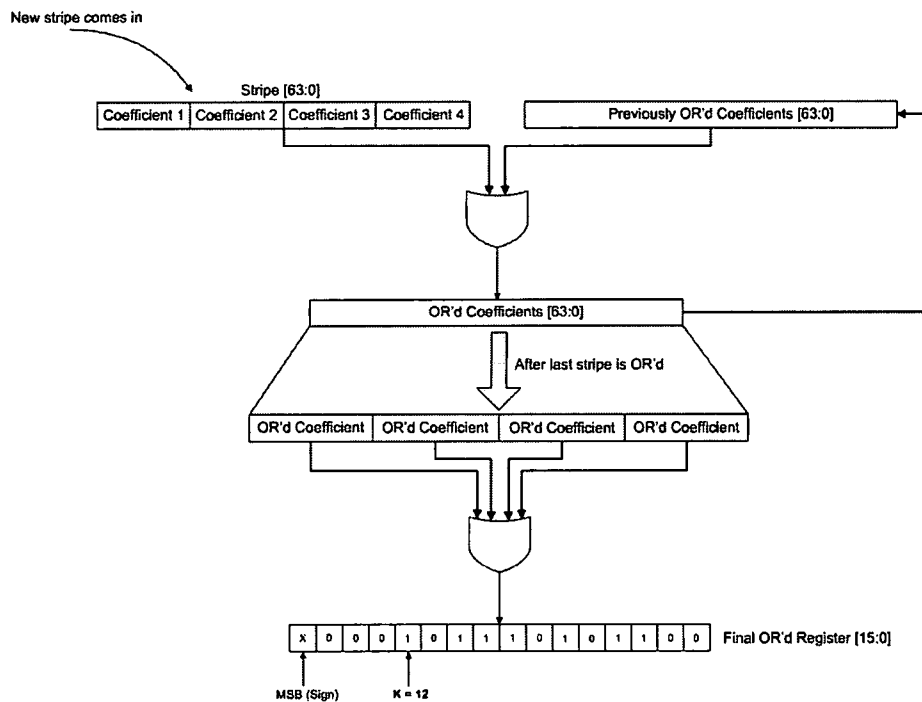


Figure 3.9: Diagram showing how bit-plane K is found by the *K Calculator*.

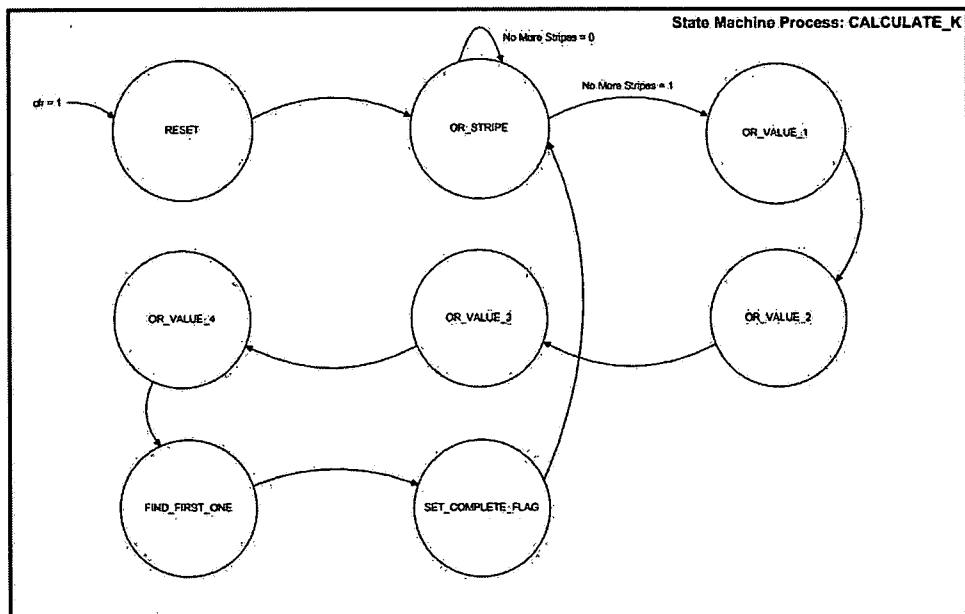


Figure 3.10: *K Calculator* state machine.

### ***K Calculator* CALCULATE\_K State Machine**

The following discussion provides an overview of the *K Calculator* state machine.

The states are:

RESET,  
OR\_STRIPE,  
OR\_VALUE\_1,  
OR\_VALUE\_2,  
OR\_VALUE\_3,  
OR\_VALUE\_4,  
FIND\_FIRST\_ONE,  
SET\_COMPLETE\_FLAG.

#### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where the ORing registers and all status signals are initialized.

#### **OR\_STRIPE**

This state bit-wise ORs a stripe and an ORing register of the same size when a flag indicates a new stripe is present. The ORing register is updated with the result of the operation.

#### **OR\_VALUE\_1:4**

After the last stripe has been OR'd with the ORing register, the K value for the code-block can be determined. The OR\_VALUE\_1:4 states bit-wise OR the four 16 bit values left in the ORing register.

#### **FIND\_FIRST\_ONE**

This state determines the K value by scanning the final OR'd register for the first one bit, starting from the most significant bit. The position of the first one bit is the K value provided to the *Tier I Sub* module.

## SET\_COMPLETE\_FLAG

This state asserts a flag indicating the value for bit-plane K is valid.

### 3.0.13 PSDX RAM

Although the *PSDX RAM* is instantiated within the *Tier I Sub* module, it makes sense to discuss the *PSDX RAM*'s data organization before examining the final pre-processing module, the *Sign Loader*. The *PSDX RAM* is used to store the coding pass membership (*P*), significance (*S*), delayed significance (*D*), and sign (*X*) values for a code-block. The three state variables and the sign bits each form their own table the size of a code-block. A location in the actual code-block associates directly with a location in the tables; e.g.  $\sigma[m,n]$  corresponds directly to the code-block coefficient at location  $[m,n]$ .

When a code-block's sign bits are loaded into the sign table within the *PSDX RAM*, the bits representing the state variable tables are all initialized to zeros. Figure 3.11 shows that each table has zero border columns surrounding the state variable and sign data; For ease of discussion, state table data will refer collectively to both the state variables and sign values hereafter. The zero borders are used when stripes along the state table edges are examined. The left side double zero border shown in Figure 3.11 is implementation specific, and its purpose will be described in Section 3.0.17 regarding the *State Tables Controller*.

As Figure 3.11 demonstrates, the data in the *PSDX RAM* is stored in groups of two at each RAM address. Storing the data in pairs allows a stripe and its vertical neighbors to be accessed in three reads (Figure 3.12). The true benefit is that three consecutive addresses can provide a valid stripe and its vertical neighbors. Therefore,

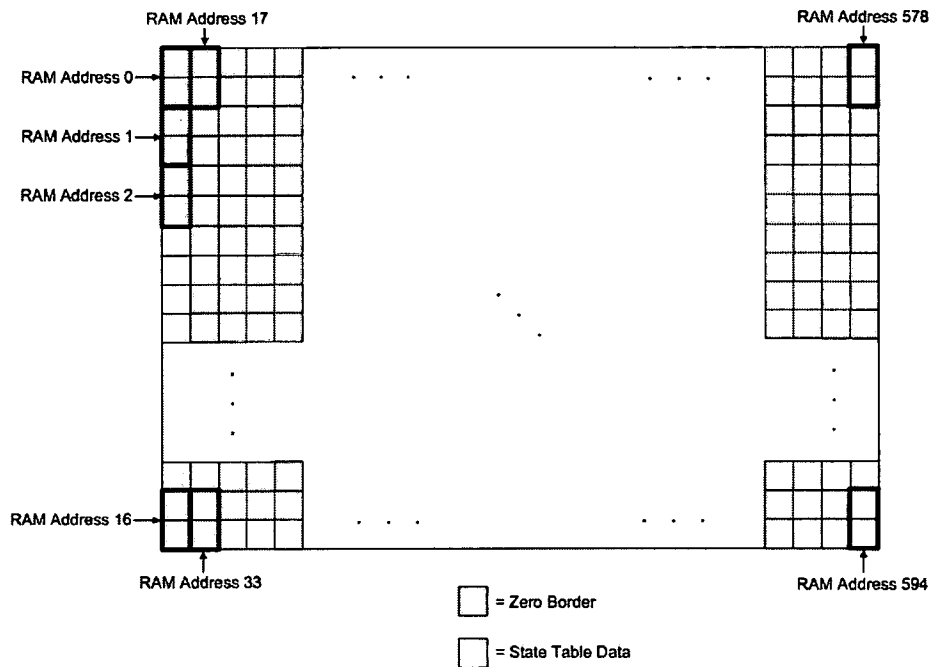


Figure 3.11: *PSDX RAM* memory organization for one state table.

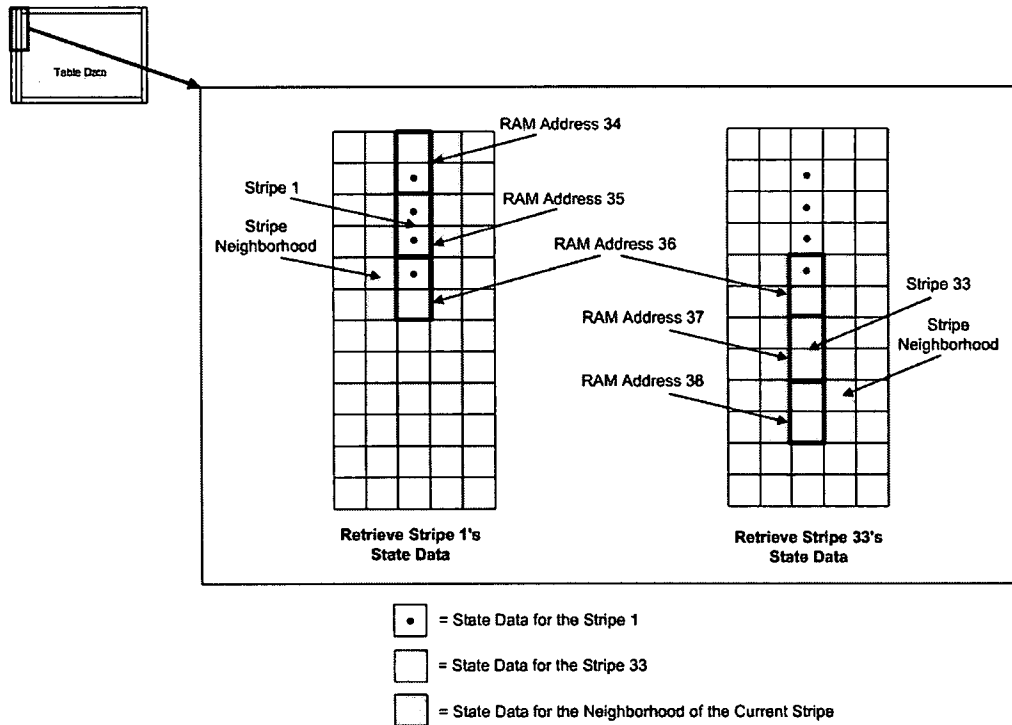


Figure 3.12: *PSDX RAM* example access of stripes 1 and 33.

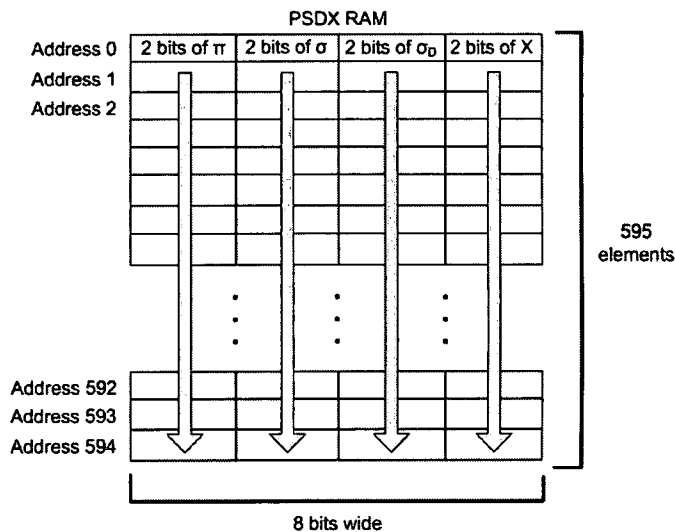


Figure 3.13: *PSDX RAM* memory organization.

the only logic required to access valid stripes is simple address control, which is conducted by the *State Tables Controller*. It is important to visualize the relationship between Figures 3.11 and 3.13. Figure 3.11 shows one state table and how its bits are grouped at RAM addresses. The organization is the same for all four tables, so each *PSDX RAM* location is made large enough to hold four tables worth of data. Since table data is stored in groups of two, each RAM location is therefore 8 bits wide. Figure 3.13 shows the organization of the state table data in the *PSDX RAM*.

### 3.0.14 Sign Loader

The *Sign Loader* extracts the four sign bits from a stripe and loads them into the *PSDX RAM* before Tier I encoding begins. The sign bits are loaded into the two least significant bit locations at each *PSDX RAM* address, starting at address 34 as indicated by Figure 3.12. Because sign bits are loaded into a RAM it takes four clock

cycles to load all four sign bits of a stripe. The state machine for the *Sign Loader* is shown in Figure 3.14.

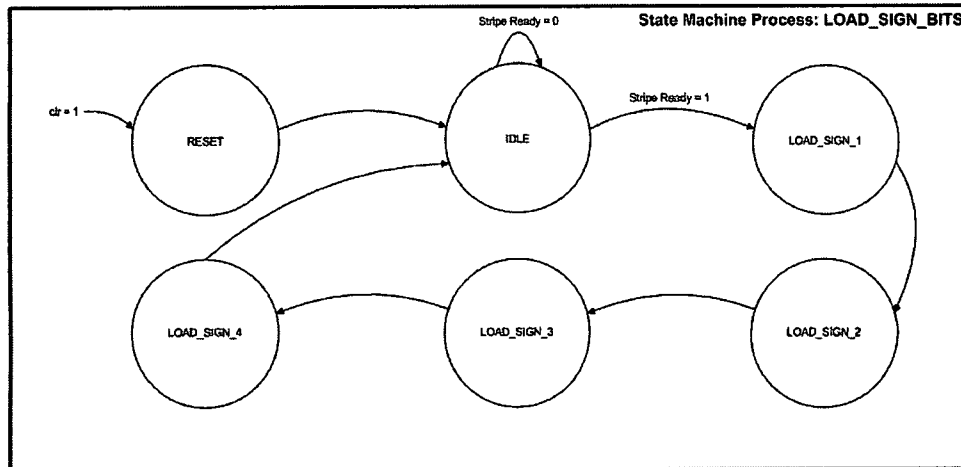


Figure 3.14: *Sign Loader* state machine.

### ***Sign Loader* LOAD\_SIGN\_BITS State Machine**

The following discussion provides an overview of the *Sign Loader* state machine.

The states are:

RESET,  
 IDLE,  
 LOAD\_SIGN\_1,  
 LOAD\_SIGN\_2,  
 LOAD\_SIGN\_3,  
 LOAD\_SIGN\_4.

#### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where the *PSDX RAM* address and write enable signals are initialized along with all other status signals.

## IDLE

This state registers the sign bits of a stripe once a flag is asserted to indicate a valid stripe is present.

## LOAD\_SIGN\_1:4

The LOAD\_SIGN\_1:4 states place the four sign bits in *PSDX RAM* locations representing valid state table data. The states control the addressing and keep track of when the top and bottom zero borders must be accounted for when loading sign bits. In addition, bits 7:2 of each RAM location are zeroed for proper initialization of the three state variables.

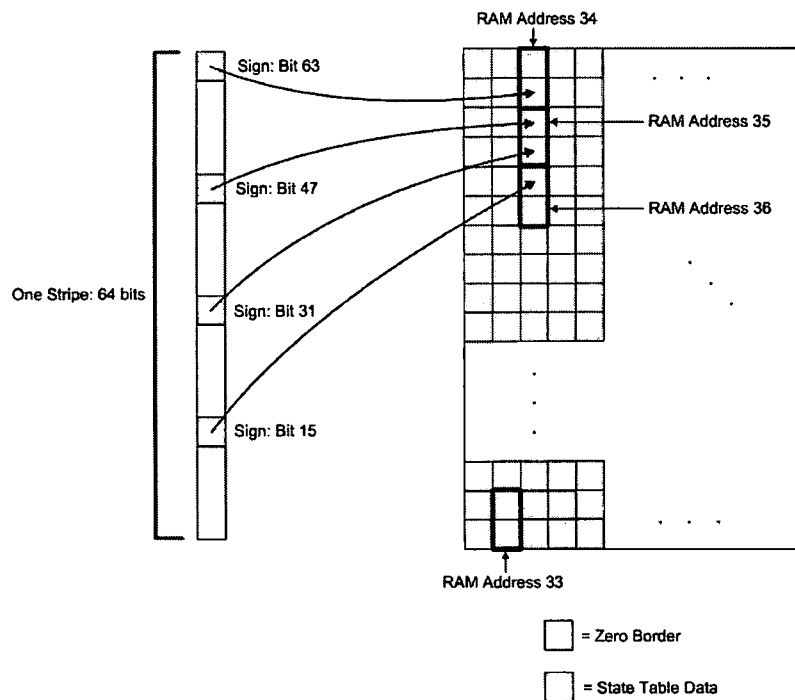


Figure 3.15: Example of where the first stripe's sign bits are loaded in terms of *PSDX RAM* addresses for a code-block.

Figure 3.15 shows how the sign bits of the first stripe of a code-block are loaded into the sign table. For the *PSDX RAM* address groupings shown, data bit 1 is on top and bit 0 is on the bottom. Bit 1 of the *PSDX RAM* data at address 34 has a zero written to it to form the top zero border. Bit 0 receives the first sign bit of the stripe. *PSDX RAM* address 35 receives the next two sign bits for the stripe. Address 36 receives the sign bit for the last coefficient of the first stripe. The first sign bit of the next stripe is written to bit 0 of the *PSDX RAM* data at address 36. The address is written to twice, but only bit 0 is changed during the second write.

### 3.0.15 Tier I Sub

*Tier I Sub* contains two state machines responsible for controlling the interactions between the *State Tables Controller* and the three coding passes as well as managing the *MQ Coder*. An overall view of all the components within *Tier I Sub* is shown in Figure 3.16. Figure 3.17 shows a more detailed diagram of the shaded subsection of Figure 3.16.

The `PROCESS_BIT_PLANES` state machine (Figure 3.18) coordinates tasks such as keeping track of the current bit-plane being processed, determining when each coding pass needs to operate, and signalling when all of the coding pass processing is complete. The `MQ_CODER_INTERFACE` state machine (Figure 3.19) requests data from the *Decision Bit and Context FIFO* when the *MQ Coder* is idle in addition to initiating a flush operation when all of the code-block data has been processed.

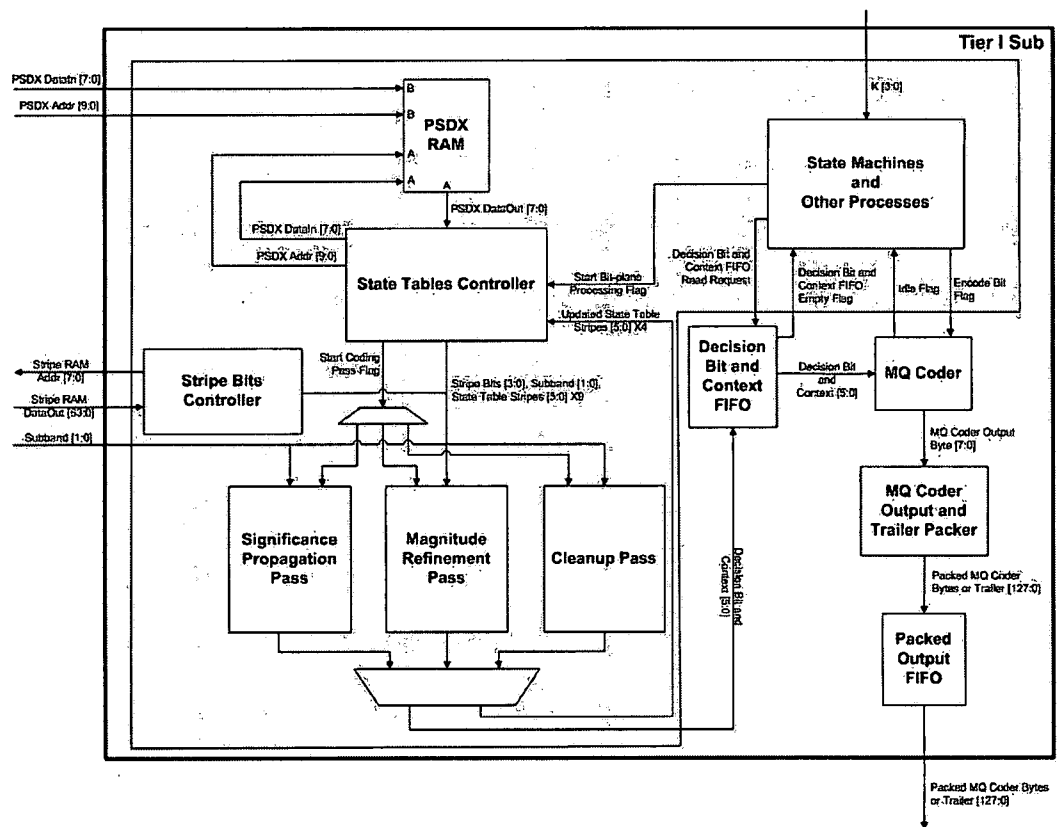


Figure 3.16: *Tier I Sub* components and major interconnect signals.

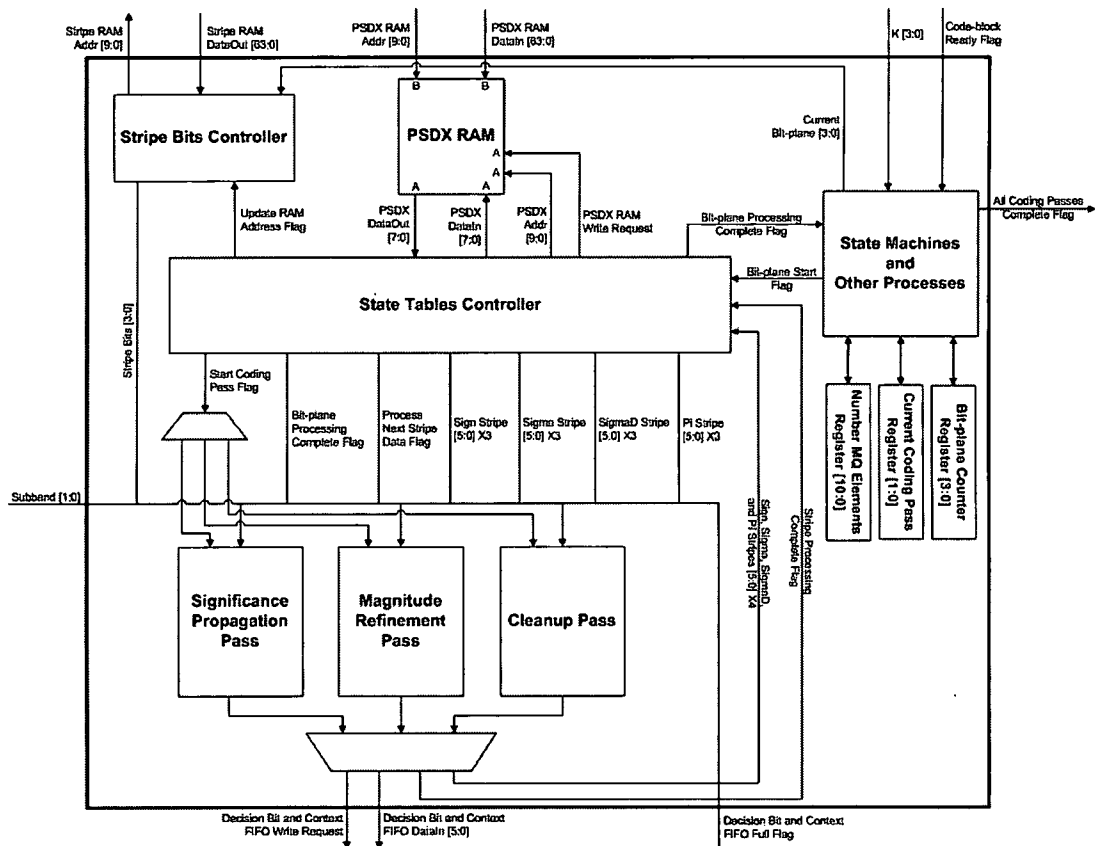


Figure 3.17: Detailed diagram for module connections between several *Tier 1 Sub* components (shaded region of Figure 3.16).

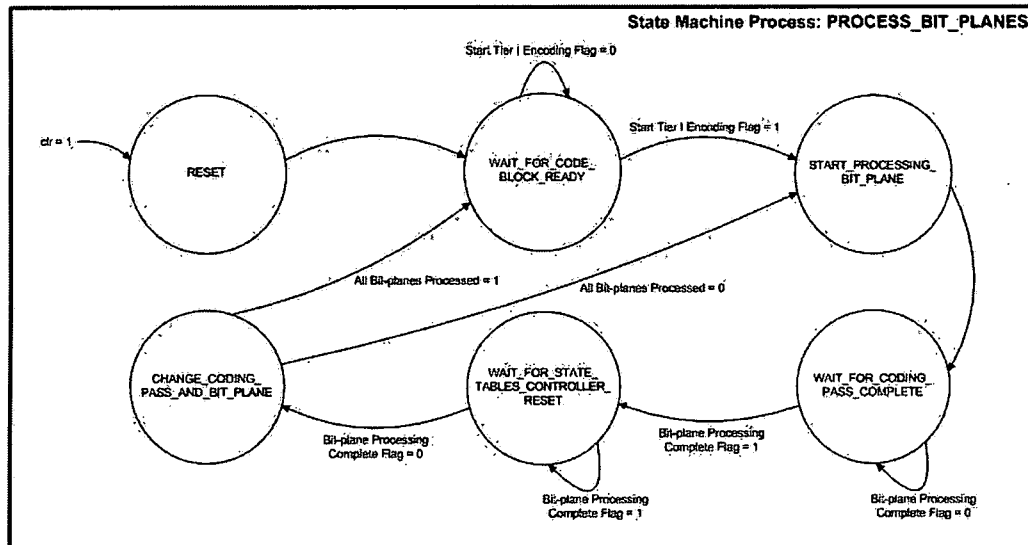


Figure 3.18: *Tier I Sub* state machine for the *State Tables Controller* and coding pass interface management.

### ***Tier I Sub* PROCESS\_BIT\_PLANES State Machine**

The following discussion provides an overview of the first *Tier I Sub* state machine.

The states are:

RESET,  
 WAIT\_FOR\_CODE\_BLOCK\_READY,  
 START\_PROCESSING\_BIT\_PLANE,  
 WAIT\_FOR\_CODING\_PASS\_COMPLETE,  
 WAIT\_FOR\_STATE\_TABLES\_CONTROLLER\_RESET,  
 CHANGE\_CODING\_PASS\_AND\_BIT\_PLANE.

#### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where all counters and status signals are initialized.

#### **WAIT\_FOR\_CODE\_BLOCK\_READY**

This state waits for *Tier I Top* to signal that all pre-processing operations are complete, meaning a code-block is available in the *Stripe RAM*, the K value for the code-block is known, and all the sign bits have been loaded into the *PSDX RAM*.

#### **START\_PROCESSING\_BIT\_PLANE**

This state asserts a flag to transition the state machine of the targeted coding pass out of its IDLE state.

#### **WAIT\_FOR\_CODING\_PASS\_COMPLETE**

This state waits for the *State Tables Controller* to signal that the bit-plane has been completely processed by a coding pass. Once the bit-plane is processed a counter is incremented to track the number of coding passes, which is later inserted into the trailer by the *MQ Coder Output and Trailer Packer*.

#### **WAIT\_FOR\_STATE\_TABLES\_CONTROLLER\_RESET**

This state waits for the *State Tables Controller* to reset so it can begin processing another bit-plane.

#### **CHANGE\_CODING\_PASS\_AND\_BIT\_PLANE**

This state determines the next coding pass to run, whether a new bit-plane needs examined, and whether to signal that all of the coding pass processing is complete.

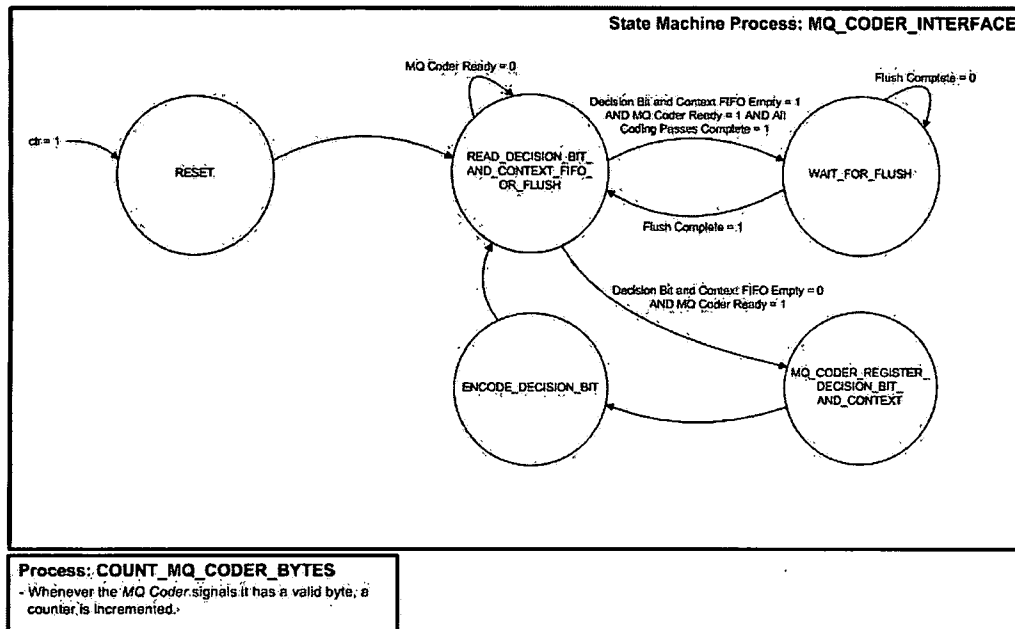


Figure 3.19: *Tier I Sub* state machine for *MQ\_Coder* interface management and byte counting process.

### ***Tier I Sub* MQ\_CODER\_INTERFACE State Machine**

The following discussion provides an overview of the second *Tier I Sub* state machine.

The states are:

RESET,  
 READ\_DECISION\_BIT\_AND\_CONTEXT\_FIFO\_OR\_FLUSH,  
 WAIT\_FOR\_FLUSH,  
 MQ\_CODER\_REGISTER\_DECISION\_BIT\_AND\_CONTEXT,  
 ENCODE\_DECISION\_BIT.

#### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where all signals are initialized.

## **READ\_DECISION\_BIT\_AND\_CONTEXT\_FIFO\_OR\_FLUSH**

This state either initiates a read from the *Decision Bit and Context FIFO* or an *MQ Coder* flush. If the *MQ Coder* is idle and there is data available to process, the read request is asserted for the *Decision Bit and Context FIFO*. If the flag indicating all coding pass processing is complete has been asserted, once all of the data in the *Decision Bit and Context FIFO* is processed the *MQ Coder* flush operations are initiated.

## **WAIT\_FOR\_FLUSH**

This state waits for the *MQ Coder* to complete its flush operations and then signals that all processing for the code-block is complete.

## **MQ\_CODER\_REGISTER\_DECISION\_BIT\_AND\_CONTEXT**

This state allows the *MQ Coder* to register a decision bit and context read from the *Decision Bit and Context FIFO*.

## **ENCODE\_DECISION\_BIT**

This state signals the *MQ Coder* to encode the decision bit it has previously registered.

## **Other Process of *Tier I Sub***

The following discussion provides an overview of the process of *Tier I Sub* other than the state machines.

## **COUNT\_MQ\_CODER\_BYTES**

This process increments a counter each time the *MQ Coder* outputs a valid byte. The counter value is passed to the *MQ Coder Output and Trailer Packer* for insertion into the code-block's trailer.

### 3.0.16 Stripe Bits Controller

The *Stripe Bits Controller* pulls stripes from the *Stripe RAM* in the proper processing order; i.e. stripes are processed in raster scan order as opposed to column by column, which is how they are stored. The *Stripe Bits Controller* uses the value of the bit-plane counter from *Tier I Sub* to determine which bits of a stripe to extract for processing. Because the code-blocks are  $32 \times 32$  samples and four coefficients are stored at each *Stripe RAM* location, the address needs to increment by 8 to access a stripe directly adjacent to the current stripe in the next column. In Figure 3.20 stripes at addresses 0 and 8 would be the first and second stripes processed, respectively. After the stripe at *Stripe RAM* address 248 is processed, the next stripe to process resides at address 1. The *Stripe Bits Controller* therefore subtracts 247 from the current address and does so whenever the current address is referencing a stripe on the right border of the code-block. For example, if the stripe at *PSDX RAM* address 254 has just been processed, 247 would be subtracted from 254 to obtain address 7, the location of the next stripe. Figure 3.21 presents another depiction of how the *Stripe Bits Controller* accesses code-block stripes, as well as the majority of the information sent to a coding pass.

To extract the bits of stripes corresponding to the current bit-plane, the bit-plane counter from *Tier I Sub* is used as an index of the bits for the stripe read from the *Stripe RAM*. The bit-plane counter is initialized to  $K-1$  so that a counter value of zero represents the least significant bits in a stripe. For example, if the value of bit-plane counter is 7, then bit 7 for each of the stripe coefficients is presented to the coding passes. The *Stripe Bits Controller* obtains a new set of stripe bits whenever signalled by the *State Tables Controller*. Because the *Stripe Bits Controller* only increments

or decrements an address and masks stripe bits, no state machine is required for the module.

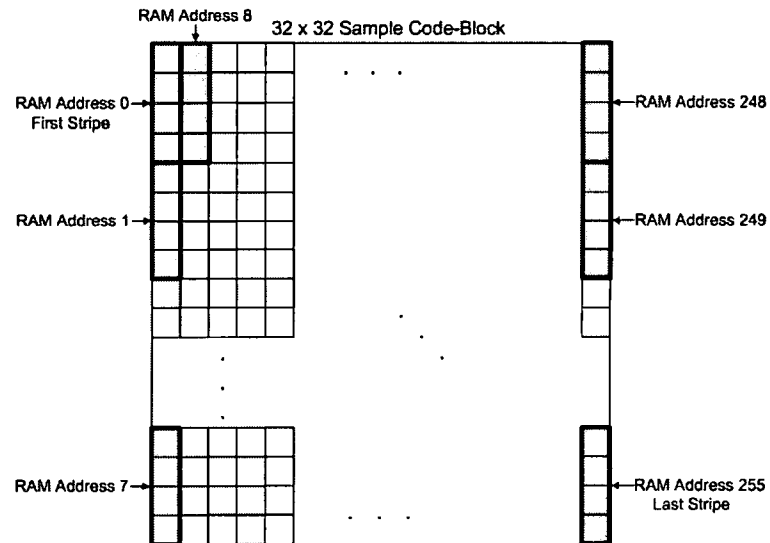


Figure 3.20: Depiction of how *Stripe RAM* locations correspond to code-block stripes.

### 3.0.17 State Tables Controller

The *State Tables Controller* is responsible for providing the coding passes with state variable and sign information<sup>5</sup> for the current stripe and its 14 surrounding coefficients. The *State Tables Controller* also updates the state tables after a coding pass operates on the stripe. The state machine for the *State Tables Controller* is provided in Figure 3.22.

The principle concept behind the *State Tables Controller's* manipulation of state table data is summarized in three stages: reading, shifting, and writing. The current state table registers shown in Figure 3.23 maintain the state table information for

<sup>5</sup>As previously denoted, the state variable and sign information will be referred to collectively as state table data.

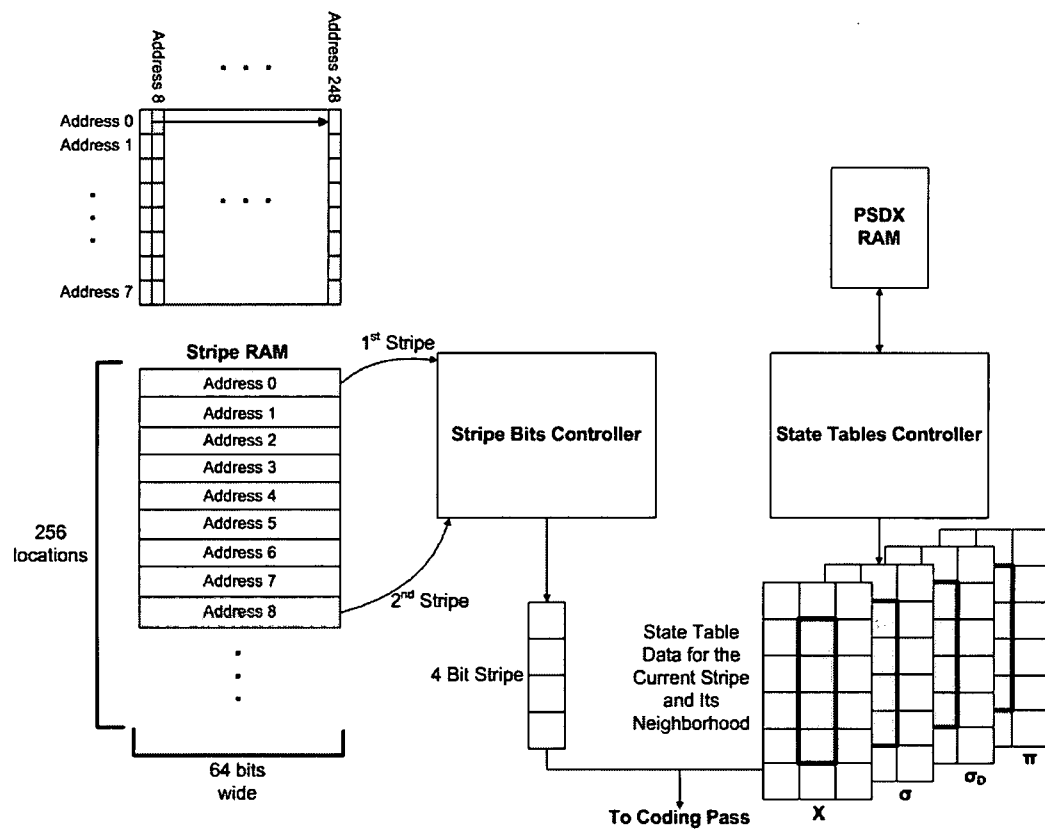


Figure 3.21: Depiction of how the *Stripe Bits Controller* accesses code-block stripes and the state table information provided to a coding pass.

the current stripe and its neighborhood, and exist separately for the  $\sigma$ ,  $\sigma_D$ ,  $\pi$ , and  $X$  variables. The state table registers are referred to in terms of columns 2, 1, and 0. Figure 3.21 shows that the state table registers for all of the state table variables are given to the coding pass currently operating. The shaded regions show the state table data associated with the current stripe being coded. Column 1 for each set of state registers is updated with new state information from the coding pass once all the stripe bits are coded. After column 1 of the state table registers is updated, previously updated data is written back to the *PSDX RAM*. The state table register columns are shifted left by one as the next state table data is read from the *PSDX RAM* and inserted into column 0 of the state table registers. Column 2 is moved to a write register to be written back to the *PSDX RAM* after the next stripe is coded. This cycle of reading, shifting, and writing will be explored further in the description of the state machine.

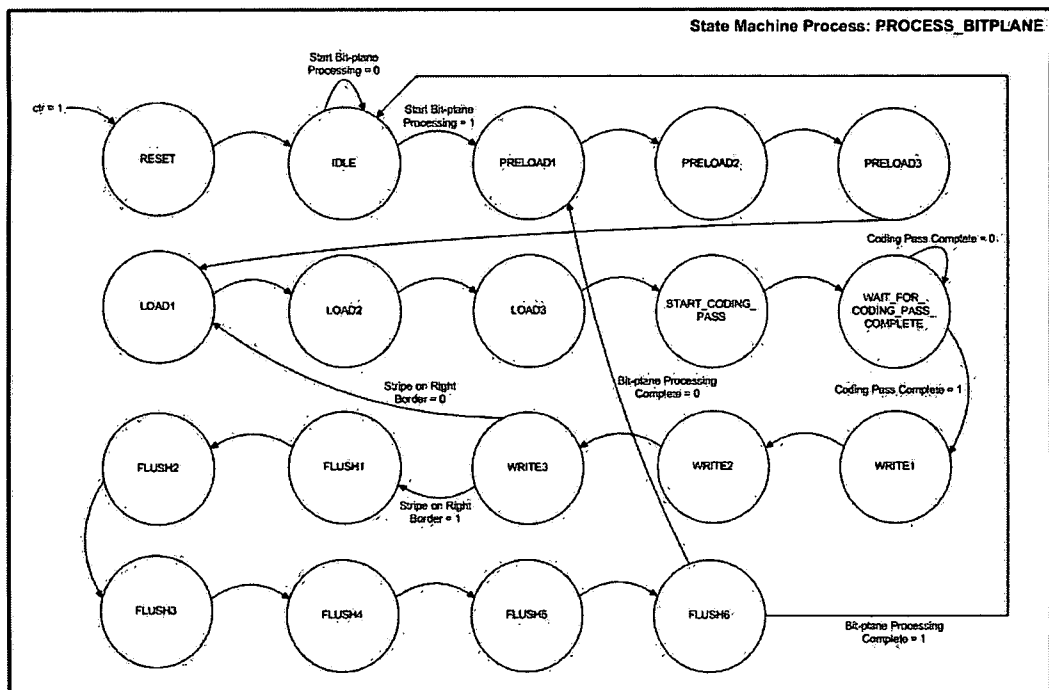


Figure 3.22: *State Tables Controller* state machine.

## ***State Tables Controller* PROCESS\_BITPLANE State Machine**

The following discussion provides an overview of the *State Tables Controller* state machine.

The states are:

RESET,  
IDLE,  
PRELOAD1:3,  
LOAD1:3,  
START\_CODING\_PASS,  
WAIT\_FOR\_CODING\_PASS\_COMPLETE,  
WRITE1:3,  
FLUSH1:6.

### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where all signals are initialized. The write register and the registers for the current state table data are initialized to zeros.

### **IDLE**

This state waits for *Tier I Sub* to initiate bit-plane processing. As shown in Figure 3.23, the *State Tables Controller* requires that the state variable tables be initialized to zeros within the *PSDX RAM* at the start of code-block processing. The sign table is loaded during Tier I pre-processing by the *Sign Loader*.

### **PRELOAD1:3**

An example of the PRELOAD operations is shown in Figure 3.24. Whenever a stripe in the left border of a code-block is processed, the associated state table stripes are read by the PRELOAD states and inserted into column 0 of the state table registers (Figure 3.23 labels columns). As shown in Figure 3.21, each state table variable has its own set of state table registers. Three PRELOAD states exist because



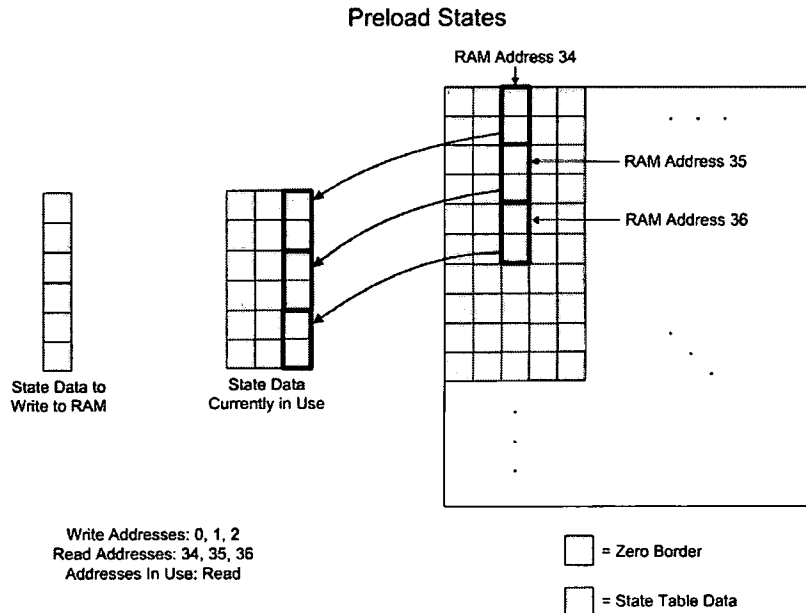


Figure 3.24: Depiction of the *State Tables Controller* PRELOAD operations.

the current state table register columns left by one. These operations continue for processing successive stripes until the right border of the state tables is reached.

### START\_CODING\_PASS

This state asserts the signal to indicate a set of load operations has completed and the state table registers are valid for use by the current coding pass.

### WAIT\_FOR\_CODING\_PASS\_COMPLETE

This state waits for the current coding pass to code the current stripe. Upon receiving status that the current coding pass has finished processing the stripe, the column 1 state table registers are updated (Figure 3.26).

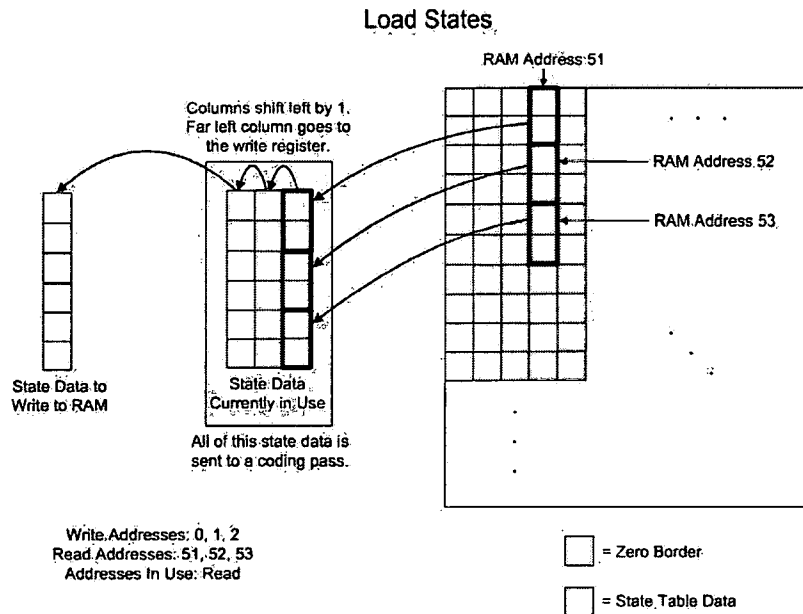


Figure 3.25: Depiction of the *State Tables Controller* first loading operations for a bit-plane.

### WRITE1:3

The WRITE states insert the write register into the *PSDX RAM*. Like the PRELOAD and LOAD states, there are three WRITE states because the write register is written to three separate *PSDX RAM* locations. The purpose of the write operations is to update the state tables as shown in Figure 3.26. The figure depicts the first write operation for a bit-plane, and it is obvious that no actual state table information is updated. The reason for the non-necessary write is for overall state machine simplicity—the state machine can always follow the pattern of reading, shifting, and writing. The two exceptions are when a left border stripe is processed (PRELOAD states will load a column of state table data first) and when the right border of the state tables is reached (FLUSH states shift and write the remaining data in the state table registers). Figures 3.27 and 3.28 show the second sets of loads, shifts, and

writes. Extending the pattern it can be seen that the third set of writes will update actual state table data.

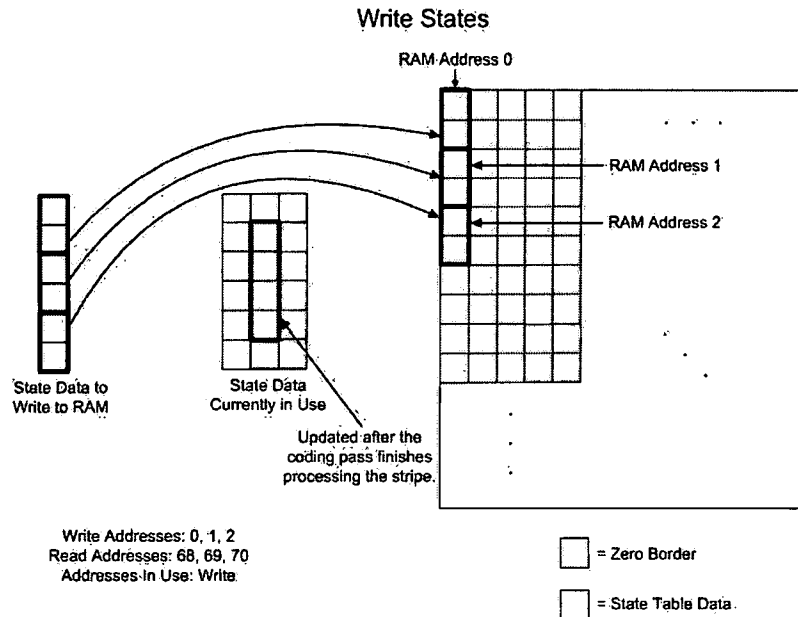


Figure 3.26: Depiction of the *State Tables Controller* first writing operations for a bit-plane.

## FLUSH1:6

The FLUSH states are entered after a stripe on the right border of a code-block has been processed, meaning the corresponding stripe in the state tables is also on the right border. Because the next stripe to code is back on the left border, the contents of the state registers must be written to the *PSDX RAM* before being filled with the left border state table data. Figures 3.29 and 3.30 demonstrate the actions of states FLUSH1, FLUSH2, and FLUSH3. Within the WRITE3 state, a check is made about whether the state machine should return to the LOAD1 state or the FLUSH1 state based on the addresses used for reading state table data. If the read addresses are

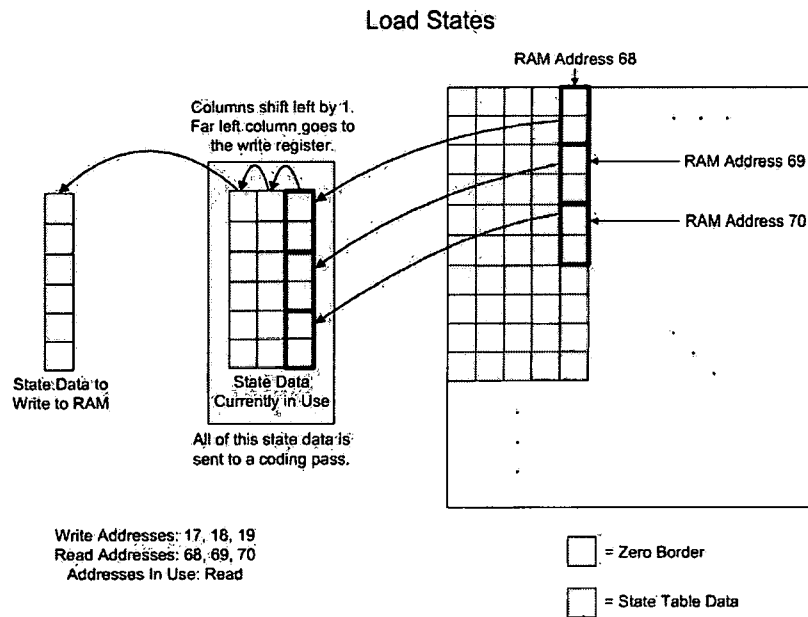


Figure 3.27: Depiction of the *State Tables Controller* second loading operations for a bit-plane.

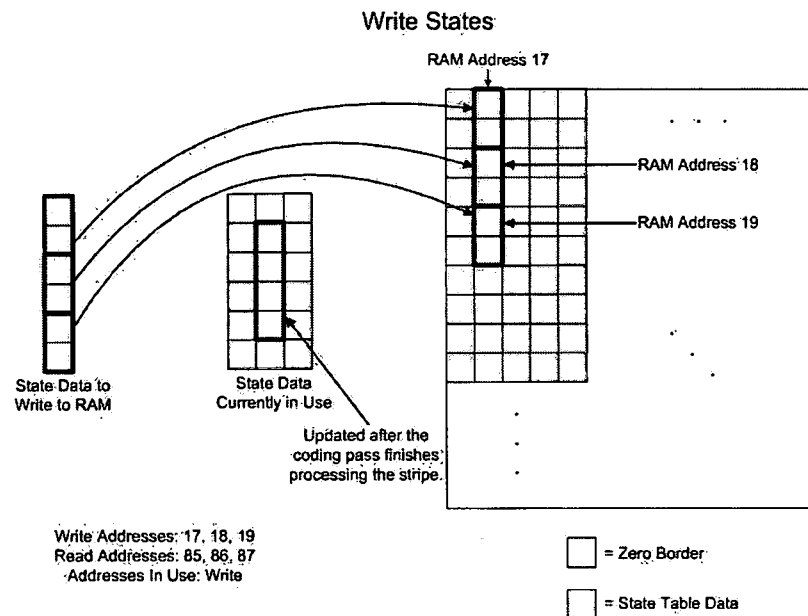


Figure 3.28: Depiction of the *State Tables Controller* second writing operations for a bit-plane.

greater than 594, the flush operations are assumed. In addition, the read and shift process is followed (same process used by the LOAD states) so that valid state table data is shifted into the write register. The FLUSH1 state writes the shifted portion of data into the *PSDX RAM*, increments the writing address, and initiates the read and shift process again. The address of the *PSDX RAM* used for reads in the FLUSH states is the same used for writes. Since the objective is to only write the valid state table data within the state table registers up to the last processed stripe, filling the state table registers with data from the write address locations does not matter. For example, in Figure 3.29 the read and shift process loads the two most significant bits of column 0 of the state table registers with data from address 544 of the *PSDX RAM*. This loading is irrelevant, since the purpose of using the process is to shift part the state table register data into the write register. As Figure 3.30 shows, the state table data in the two most significant bits of write register is then written to address 544 of the *PSDX RAM*. The same events occur for address 545 and 546. Figures 3.31 and 3.32 depict the actions of states FLUSH4, FLUSH5, and FLUSH6. Once the FLUSH6 state completes, the state table data for the right border has been written the the *PSDX RAM* and the left border stripe can be processed. If the state table write addresses align with the bottom right corner of the zero border, it means the bit-plane has been completely processed (Figure 3.33). The starting read and write addresses are re-initialized and the state machine is sent to the IDLE state, where it waits to begin processing the next bit-plane.

### Flush States: Reads

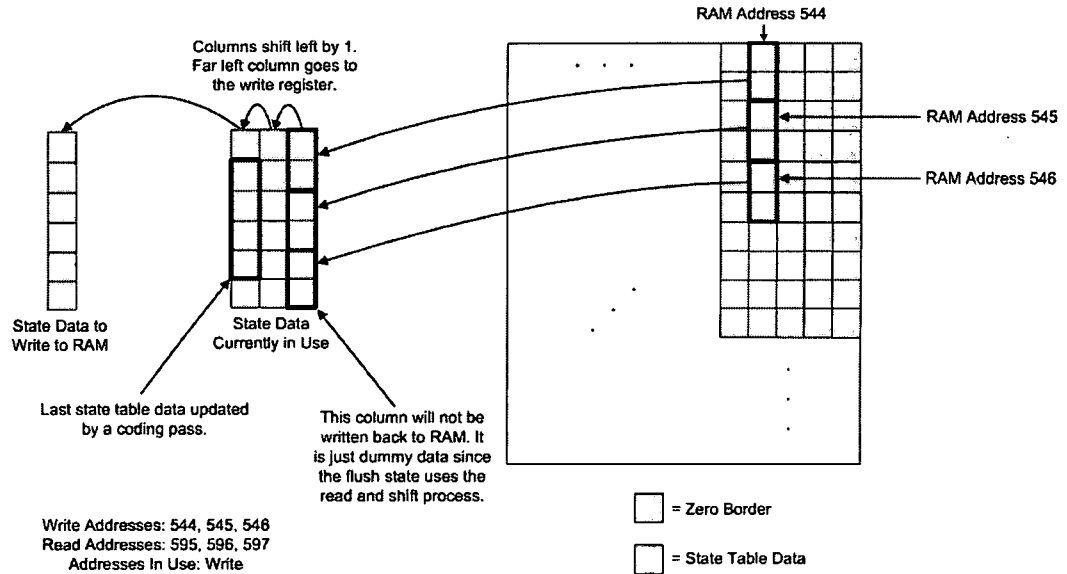


Figure 3.29: Depiction of the *State Tables Controller* first read operations for the first flush for a bit-plane.

### Flush States: Writes

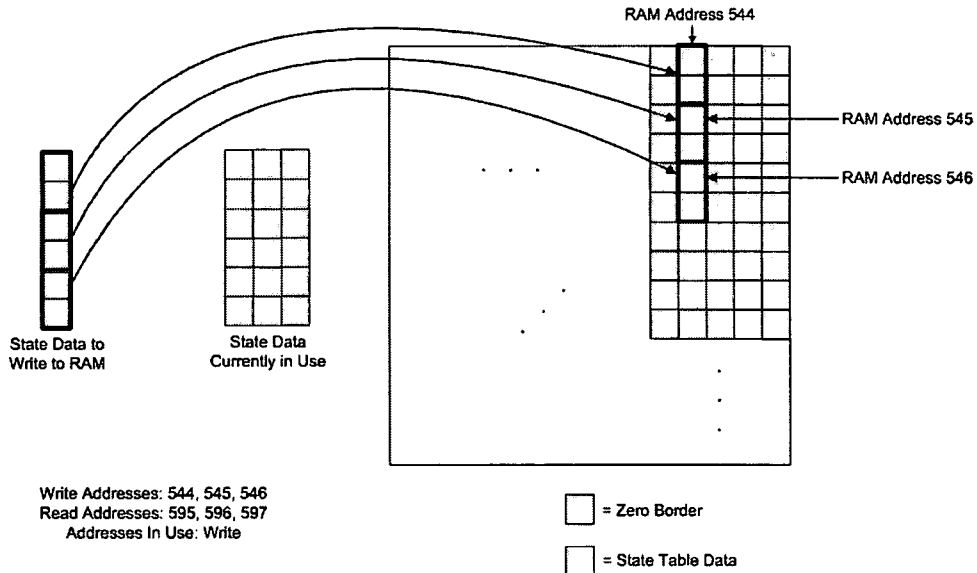


Figure 3.30: Depiction of the *State Tables Controller* first write operations for the first flush for a bit-plane.

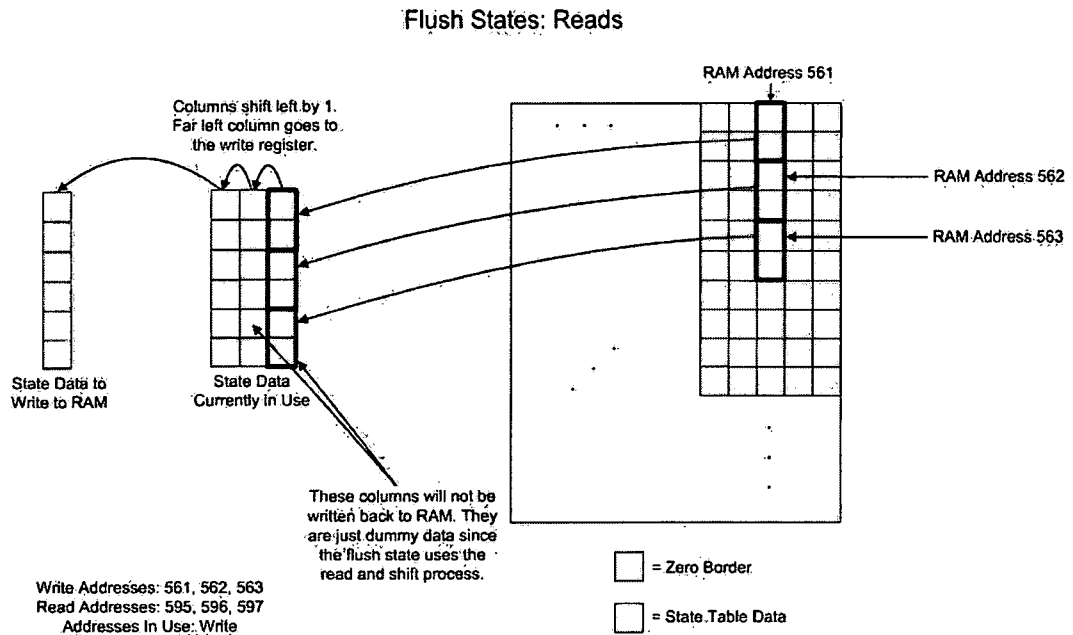


Figure 3.31: Depiction of the *State Tables Controller* second read operations for the first flush for a bit-plane.

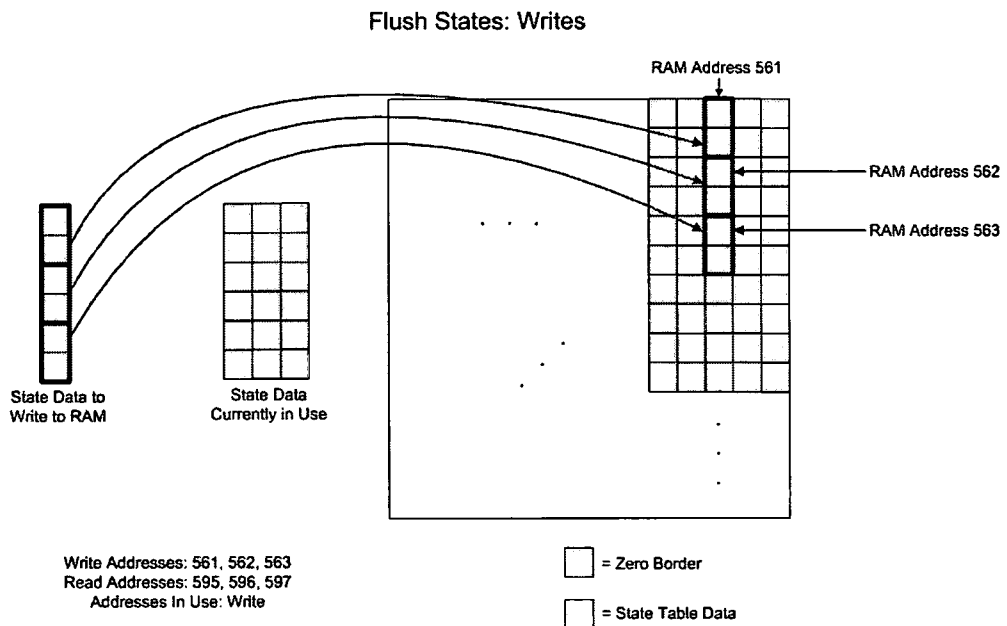


Figure 3.32: Depiction of the *State Tables Controller* second write operations for the first flush for a bit-plane.

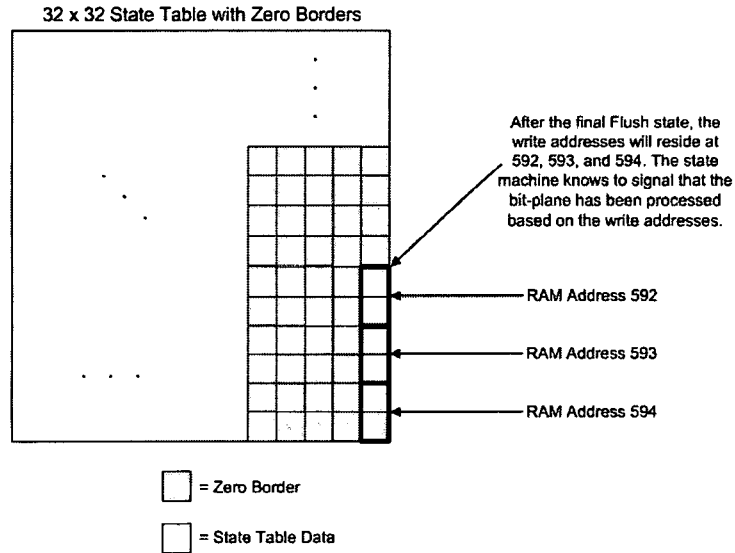


Figure 3.33: Depiction of the how the *State Tables Controller* determines a bit-plane has been processed.

### 3.0.18 The Coding Passes

The coding passes—Significance Propagation Pass (SPP), Magnitude Refinement Pass (MRP), and Cleanup Pass (CUP)—receive bit-plane stripes from the *Stripe Bits Controller* and state table information from the *State Tables Controller*. The coding passes are “blind” in the sense that none cares which bit-plane is currently being processed. Two of the coding passes are instead subband oriented, meaning the code-block’s subband determines the context used when zero coding. Although functionally different, the structures of the coding passes are all very similar. The coding passes use several lookup tables to provide contexts for bits being coded. The *Zero Coding Lookups*, *Sign Coding Lookups*, and *Magnitude Refinement Coding Lookups* modules perform three of the four types of Tier I bit-plane coding. Run-length coding is performed within the *Cleanup Pass*.

## Zero Coding Lookups

*Zero Coding Lookups* is instantiated by the *Significance Propagation Pass* and *Cleanup Pass* for each bit of the stripe under examination. The inputs to a *Zero Coding Lookups* module are the significance values of the eight neighbors surrounding a stripe bit. Based on these significance values, the *Zero Coding Lookups* outputs the context for each subband in accordance with the tables shown in Section 2.2.6. The coding pass state machine determines which context to use based on the subband. The decision bit for zero coding is the value of the stripe bit being coded, so the *Zero Coding Lookups* does not provide any information regarding the decision bit.

## Sign Lookups

*Sign Lookups* is used by the *Significance Propagation Pass* and *Cleanup Pass* for sign coding. Like the *Zero Coding Lookups*, the *Sign Lookups* module is instantiated for each bit of a stripe. The inputs to the *Sign Lookups* are the horizontal and vertical significance and sign bits for a stripe bit. Sign coding is independent of the code-block's subband, so the *Sign Lookups* returns the context for the stripe bit in accordance with the table in Section 2.2.6. The decision bit is not determined exactly. Instead  $\hat{X}$  is provided, from which the decision bit is easily determined in the coding pass; by exclusively ORing  $\hat{X}$  with  $X$ .

## Magnitude Refinement Lookups

*Magnitude Refinement Lookups* is used by the *Magnitude Refinement Pass* for determining the context for magnitude refinement coding. The *Magnitude Refinement Lookups* module is instantiated for each bit of a stripe. In addition to the delayed

significance state variable, magnitude refinement coding is dependent on the significance values of the eight neighbors of the stripe bit being coded. The *Magnitude Refinement Lookups* returns a context for the stripe bit being coded as determined by the table in Section 2.2.6, but does not return a decision bit since the decision bit is equal to the value of the bit being coded.

### Significance Propagation Pass

As described in Section 3.0.18, the SPP processes the bit-plane following the first non-zero bit-plane, all the way through the least significant bit-plane. After bit-plane K is processed by the CUP, the SPP is the first coding pass to operate on all subsequent bit-planes for the code-block. The SPP performs zero coding and sign coding. The SPP state machine in Figure 3.34 very closely follows the block diagram shown in Figure 2.15.

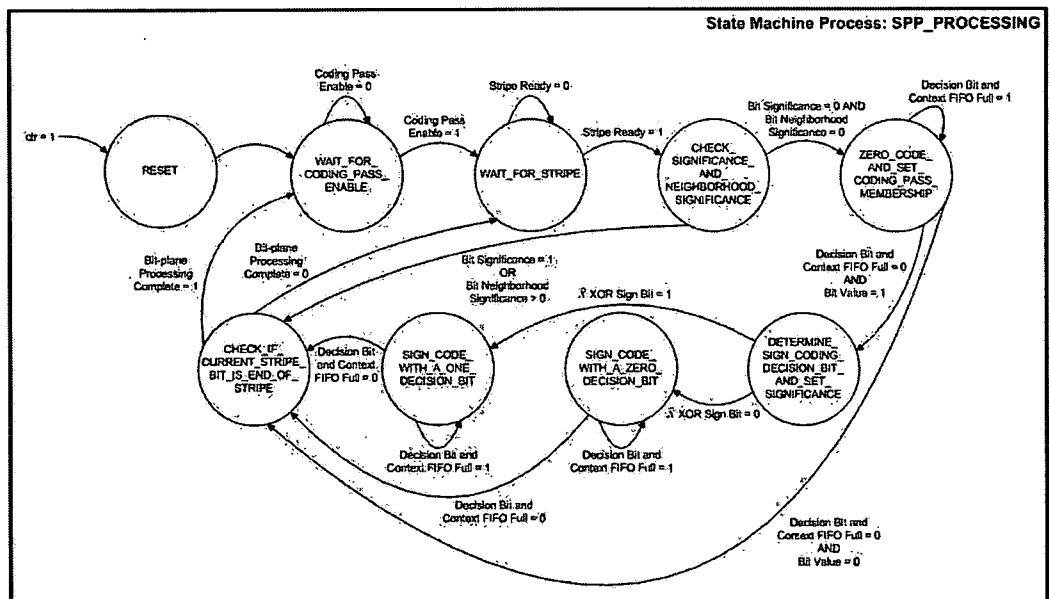


Figure 3.34: *Significance Propagation Pass* state machine.

### ***Significance Propagation Pass* SPP\_PROCESSING State Machine**

The following discussion provides an overview of the *Significance Propagation Pass* state machine.

The states are:

RESET,  
WAIT\_FOR\_CODING\_PASS\_ENABLE,  
WAIT\_FOR\_STRIPE,  
CHECK\_SIGNIFICANCE\_AND\_NEIGHBORHOOD\_SIGNIFICANCE,  
ZERO\_CODE\_AND\_SET\_CODING\_PASS\_MEMBERSHIP,  
DETERMINE\_SIGN\_CODING\_DECISION\_BIT\_AND\_SET\_SIGNIFICANCE,  
SIGN\_CODE\_WITH\_A\_ZERO\_DECISION\_BIT,  
SIGN\_CODE\_WITH\_A\_ONE\_DECISION\_BIT,  
CHECK\_IF\_CURRENT\_STRIPE\_BIT\_IS\_END\_OF\_STRIPE.

#### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where all signals are initialized.

#### **WAIT\_FOR\_CODING\_PASS\_ENABLE**

This state waits for the *State Tables Controller* to enable the coding pass.

#### **WAIT\_FOR\_STRIPE**

This state waits for the *State Tables Controller* to indicate that a valid stripe and its state table data are ready for coding.

#### **CHECK\_SIGNIFICANCE\_AND\_NEIGHBORHOOD\_SIGNIFICANCE**

This state checks whether or not the stripe bit under examination is significant or has significant neighbors. If the bit is not significant but has significant neighbors, it is zero coded using the *Zero Coding Lookups*. Otherwise, the coding pass membership state variable is set to zero for the bit, meaning the SPP did not perform zero coding on the bit in the current bit-plane.

## **ZERO\_CODE\_AND\_SET\_CODING\_PASS\_MEMBERSHIP**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. Depending on the subband of the code-block, the appropriate zero coding context is selected from the *Zero Coding Lookups* module and written to the *Decision Bit and Context FIFO*. The decision bit is equal to the value of the current stripe bit. In addition, the coding pass membership state variable is set to one for the bit to indicate zero coding has been performed.

## **DETERMINE\_SIGN\_CODING\_DECISION\_BIT\_AND\_SET\_SIGNIFICANCE**

This state sets the significance to one for the bit being coded, while also determining the decision bit for sign coding. The *Sign Coding Lookups* module returns an  $\hat{X}$  parameter, which is exclusively OR'd with the bit's sign value. If the result of the exclusive OR is one, the sign coding decision bit is one. Otherwise, it is zero.

## **SIGN\_CODE\_WITH\_A\_ZERO\_DECISION\_BIT**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit of zero and the context returned by the *Sign Coding Lookups* module are written to the *Decision Bit and Context FIFO*.

## **SIGN\_CODE\_WITH\_A\_ONE\_DECISION\_BIT**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit of one and the context returned by the *Sign Coding Lookups* module are written to the *Decision Bit and Context FIFO*.

## **CHECK\_IF\_CURRENT\_STRIPE\_BIT\_IS\_END\_OF\_STRIPE**

This state checks whether the last bit of the stripe has been processed. If so, a flag indicating the stripe has been coded is asserted. Otherwise, the next bit of the current stripe is examined.

## Magnitude Refinement Pass

As described in Section 3.0.18, the MRP processes the bit-plane following the first non-zero bit-plane, all the way through the least significant bit-plane. The MRP always follows the SPP during processing. The MRP instantiates four *Magnitude Refinement Lookups* modules; one for each bit of the stripe passed to the MRP. The MRP state machine in Figure 3.35 very closely follows the block diagram shown in Figure 2.16.

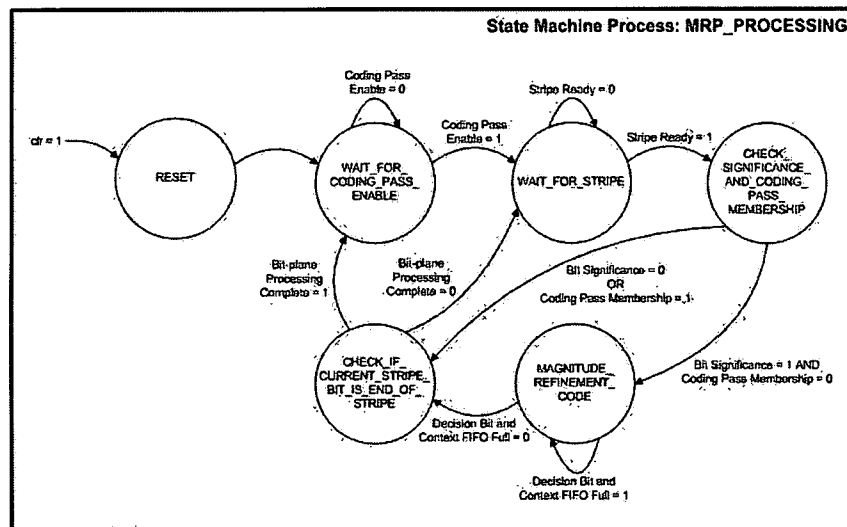


Figure 3.35: *Magnitude Refinement Pass* state machine.

## ***Magnitude Refinement Pass* MRP\_PROCESSING State Machine**

The following discussion provides an overview of the *Magnitude Refinement Pass* state machine.

The states are:

RESET,  
WAIT\_FOR\_CODING\_PASS\_ENABLE,  
WAIT\_FOR\_STRIPE,  
CHECK\_SIGNIFICANCE\_AND\_CODING\_PASS\_MEMBERSHIP,  
MAGNITUDE\_REFINEMENT\_CODE,  
CHECK\_IF\_CURRENT\_STRIPE\_BIT\_IS\_END\_OF\_STRIPE.

### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where all signals are initialized.

### **WAIT\_FOR\_CODING\_PASS\_ENABLE**

This state waits for the *State Tables Controller* to enable the coding pass.

### **WAIT\_FOR\_STRIPE**

This state waits for the *State Tables Controller* to indicate that a valid stripe and its state table data are ready for coding.

### **CHECK\_SIGNIFICANCE\_AND\_CODING\_PASS\_MEMBERSHIP**

This state checks whether or not the stripe bit under examination is significant or has already been zero coded. If the bit is significant and has not already been zero coded during the SPP, it is magnitude refinement coded using the *Magnitude Refinement Coding Lookups*.

### **MAGNITUDE\_REFINEMENT\_CODE**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit equal to the bit being coded and the context returned

by the *Magnitude Refinement Coding Lookups* module are written to the *Decision Bit and Context FIFO*. In addition, the delayed significance state variable is set to one.

#### **CHECK\_IF\_CURRENT\_STRIPE\_BIT\_IS\_END\_OF\_STRIPE**

This state checks whether the last bit of the stripe has been processed. If so, a flag indicating the stripe has been coded is asserted. Otherwise, the next bit of the current stripe is examined.

#### **Cleanup Pass**

As described in Section 3.0.18, the CUP processes the first non-zero bit-plane, K, all the way through the least significant bit-plane. After bit-plane K is processed, the CUP always follows the MRP on all subsequent bit-planes for the code-block. The CUP instantiates four *Zero Coding Lookups* and four *Sign Coding Lookups* modules; one for each bit of the stripe passed to the CUP. The CUP state machine in Figure 3.36 very closely follows the block diagram shown in Figure 2.17.

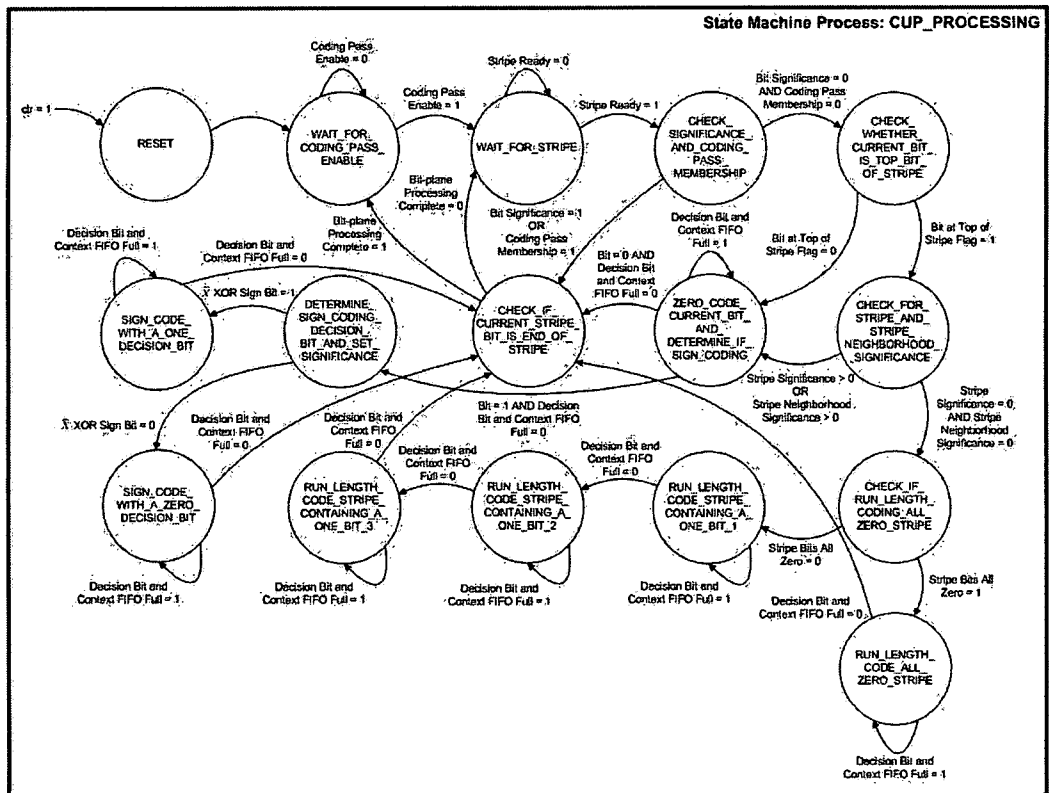


Figure 3.36: *Cleanup Pass* state machine.

### ***Cleanup Pass* CUP\_PROCESSING State Machine**

The following discussion provides an overview of the *Cleanup Pass* state machine.

The states are:

RESET,  
WAIT\_FOR\_CODING\_PASS\_ENABLE,  
WAIT\_FOR\_STRIPE,  
CHECK\_SIGNIFICANCE\_AND\_CODING\_PASS\_MEMBERSHIP,  
CHECK\_WHETHER\_CURRENT\_BIT\_IS\_TOP\_BIT\_OF\_STRIPE,  
CHECK\_FOR\_STRIPE\_AND\_STRIPE\_NEIGHBORHOOD\_SIGNIFICANCE,  
CHECK\_IF\_RUN\_LENGTH\_CODING\_ALL\_ZERO\_STRIPE,  
RUN\_LENGTH\_CODE\_ALL\_ZERO\_STRIPE,  
RUN\_LENGTH\_CODE\_STRIPE\_CONTAINING\_A\_ONE\_BIT\_1,  
RUN\_LENGTH\_CODE\_STRIPE\_CONTAINING\_A\_ONE\_BIT\_2,  
RUN\_LENGTH\_CODE\_STRIPE\_CONTAINING\_A\_ONE\_BIT\_3,  
ZERO\_CODE\_CURRENT\_BIT\_AND\_DETERMINE\_IF\_SIGN\_CODING,  
DETERMINE\_SIGN\_CODING\_DECISION\_BIT\_AND\_SET\_SIGNIFICANCE,  
SIGN\_CODE\_WITH\_A\_ZERO\_DECISION\_BIT,  
SIGN\_CODE\_WITH\_A\_ONE\_DECISION\_BIT,  
CHECK\_IF\_CURRENT\_STRIPE\_BIT\_IS\_END\_OF\_STRIPE.

#### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where all signals are initialized.

#### **WAIT\_FOR\_CODING\_PASS\_ENABLE**

This state waits for the *State Tables Controller* to enable the coding pass.

#### **WAIT\_FOR\_STRIPE**

This state waits for the *State Tables Controller* to indicate that a valid stripe and its state table data are ready for coding.

## **CHECK\_SIGNIFICANCE\_AND\_CODING\_PASS\_MEMBERSHIP**

This state checks whether or not the stripe bit under examination is significant or has already been zero coded by the SPP. If either are true the next bit of the stripe is examined.

## **CHECK\_WHETHER\_CURRENT\_BIT\_IS\_TOP\_BIT\_OF\_STRIPE**

If the current bit is not significant and has not been zero coded by the SPP, a check is made to determine whether the bit is the top bit of a complete four bit stripe. The proposed design currently only supports  $32 \times 32$  code-blocks, so all stripes are complete. If the bit is not the top bit of the stripe and has not already been included in a run-length code, it is zero coded.

## **CHECK\_FOR\_STRIPE\_AND\_STRIPE\_NEIGHBORHOOD\_SIGNIFICANCE**

This state examines the significance of the current stripe and its neighborhood. If all significance values are zero and the bit is the top bit of the stripe, run-length coding commences. Otherwise, the bit is zero coded if it has not already been included in a run-length code. If the bit has already been run-length coded the next bit of the stripe is examined.

## **CHECK\_IF\_RUN\_LENGTH\_CODING\_ALL\_ZERO\_STRIPE**

This state examines the stripe bits to determine if an all zero stripe is to be run-length coded.

## **RUN\_LENGTH\_CODE\_ALL\_ZERO\_STRIPE**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit of zero and a context of 17 are written to the *Decision Bit and Context FIFO* to indicate an all-zero stripe has been run-length coded. The next stripe is then examined.

#### **RUN\_LENGTH\_CODE\_STRIPE\_CONTAINING\_A\_ONE\_BIT\_1**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit of one and a context of 17 are written to the *Decision Bit and Context FIFO* to indicate that a stripe containing at least a single one bit has been run-length coded. The index of the first one bit is then coded.

#### **RUN\_LENGTH\_CODE\_STRIPE\_CONTAINING\_A\_ONE\_BIT\_2**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit equal to the most significant bit of the index of the first one bit of the stripe and a context of 18 are written to the *Decision Bit and Context FIFO*.

#### **RUN\_LENGTH\_CODE\_STRIPE\_CONTAINING\_A\_ONE\_BIT\_3**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit equal to the least significant bit of the index of the first one bit of the stripe and a context of 18 are written to the *Decision Bit and Context FIFO*. The sign of the first one bit is then coded.

#### **ZERO\_CODE\_CURRENT\_BIT\_AND\_DETERMINE\_IF\_SIGN\_CODING**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit equal to the stripe bit and a context returned by the *Zero Coding Lookups* module are written to the *Decision Bit and Context FIFO*. If the bit coded is a one, sign coding will commence.

#### **DETERMINE\_SIGN\_CODING\_DECISION\_BIT\_AND\_SET\_SIGNIFICANCE**

This state sets the significance to one for the bit being coded, while also determining the decision bit for sign coding. The *Sign Coding Lookups* module returns an

$\hat{X}$  parameter, which is exclusively OR'd with the bit's sign value. If the result of the exclusive OR is one, the sign coding decision bit is one. Otherwise, it is zero.

#### **SIGN\_CODE\_WITH\_A\_ZERO\_DECISION\_BIT**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit of zero and the context returned by the *Sign Coding Lookups* module are written to the *Decision Bit and Context FIFO*.

#### **SIGN\_CODE\_WITH\_A\_ONE\_DECISION\_BIT**

This state asserts the write enable for the *Decision Bit and Context FIFO*, as long as it is not full. A decision bit of one and the context returned by the *Sign Coding Lookups* module are written to the *Decision Bit and Context FIFO*.

#### **CHECK\_IF\_CURRENT\_STRIPE\_BIT\_IS\_END\_OF\_STRIPE**

This state checks whether the last bit of the stripe has been processed. If so, a flag indicating the stripe has been coded is asserted. Otherwise, the next bit of the current stripe is examined.

### **3.0.19 Decision Bit and Context FIFO**

The *Decision Bit and Context FIFO* buffers the output of the coding passes, since the coding passes provide contexts and decision bits at a faster rate than the *MQ Coder* processes them. The *Decision Bit and Context FIFO* is 6 bits wide and 1024 elements deep. The most significant bit at each location is the decision bit, and bits 4:0 represent the context as shown in Figure 3.37.

### **3.0.20 MQ Coder**

The *MQ Coder* uses the contexts and decision bits produced by the coding passes to generate a compressed bit-stream for each code-block. The *MQ Coder* is a fairly

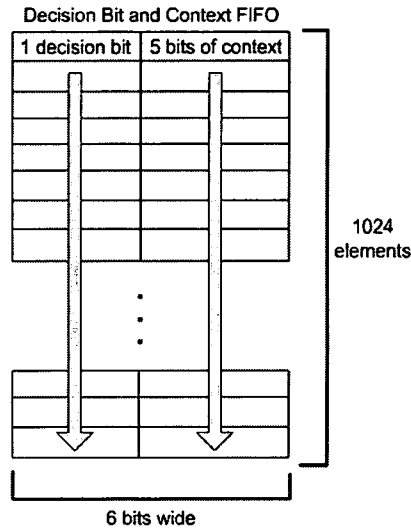


Figure 3.37: Depiction of how data is stored in the *Decision Bit and Context FIFO*.

large and complicated portion of Tier I, and is documented separately from this thesis. For detailed information regarding the *MQ Coder* design used in the proposed Tier I hardware architecture, refer to [11].

### 3.0.21 MQ Coder Output and Trailer Packer

The *MQ Coder Output and Trailer Packer* collects bytes produced by the *MQ Coder* and packs them into 128 bit words for insertion into the *Packed Output FIFO*. Packing the *MQ Coder* data consists of inserting bytes back to back in a 128 bit word. The *MQ Coder Output and Trailer Packer* coordinates the order in which bytes are packed and when a packed register is written to the *Packed Output FIFO*. Once all of the *MQ Coder* bytes are received for a code-block, a trailer is inserted into the *Packed Output FIFO* to serve as a code-block separator after the data is moved to the *Primary Output FIFO*. The trailer contains the number of *MQ Coder* bytes generated for the code-block, the value of  $K$ , and the number of coding passes that

operated on the code-block. After the *Primary Output FIFO* has been read back to software, the code-block data is parsed through based on the fact that each trailer specifies the number of *MQ Coder* bytes separating the next trailer from itself. The state machine for the *MQ Coder Output and Trailer Packer* is shown in Figure 3.38.

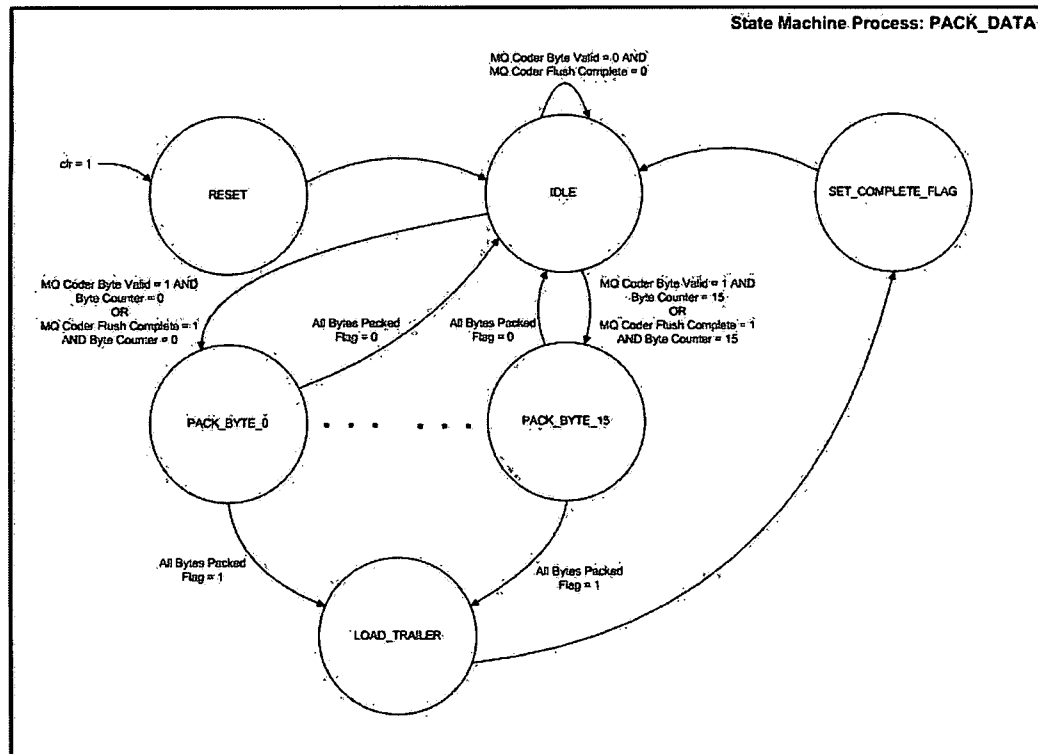


Figure 3.38: *MQ Coder Output and Trailer Packer* state machine. PACK\_BYTE\_14:1 states are inferred.

### ***MQ Coder Output and Trailer Packer PACK\_DATA State Machine***

The following discussion provides an overview of the *MQ Coder Output and Trailer Packer* state machine.

The states are:

RESET,  
IDLE,  
PACK\_BYTE\_15:0,  
LOAD\_TRAILER,  
SET\_PACKING\_COMPLETE\_FLAG.

#### **RESET**

When the *clr* signal is asserted the state machine enters the RESET state where all status and counter signals are initialized.

#### **IDLE**

This state waits for either the flag indicating a valid *MQ Coder* byte is present to be asserted or the flag indicating the *MQ Coder* has completed flushing operations to be asserted.

#### **PACK\_BYTE\_15:0**

Because the *MQ Coder* outputs bytes, to fill a 128 bit register requires 16 *MQ coder* outputs. The PACK\_BYTE\_15:0 states insert the current *MQ Coder* byte into a 128 bit register. Each state assigns the byte to a different location within the 128 bit register. State PACK\_BYTE\_0 loads the first byte into bits 7:0 of the 128 bit register. State PACK\_BYTE\_15 loads the 16<sup>th</sup> byte into bits 127:120. This ordering is dictated by the software reading the compressed data back. State PACK\_BYTE\_15 asserts the *Packed Output FIFO* write enable to write the packed register to the *Packed Output FIFO*. In the event that the *MQ Coder* has flushed and the last byte

inserted did not fill the 128 bit register completely, the most significant bits up to the last byte are zeroed and the state machine moves to the `LOAD_TRAILER` state.

### **LOAD\_TRAILER**

This state inserts a 128 bit trailer into the *Packed Output FIFO*. The trailers are used by the software to distinguish code-blocks after reading the *Primary Output FIFO* (via DMA) from an FPGA. Each trailer consists of the *K* value, number of *MQ Coder* bytes, and the total number of 128 bit elements written into the *Packed Output FIFO* for the code-block. The count for the total number of 128 bit elements written to the *Packed Output FIFO* is also passed up to *mymodule*, which keeps a running sum of the number of 128 bit elements in the *Primary Output FIFO* to be used by the DMA read back. Once all of the compressed code-block data for all of the tiles has been written to the *Primary Output FIFO*, the running sum is placed into the control register that initiates the software read back.

### **SET\_PACKING\_COMPLETE\_FLAG**

This state asserts the flag indicating that packing is complete for the current code-block. The Tier I core then waits for the Group 2 signals to connect to it so the compressed data and trailer can be loaded into the *Primary Output FIFO*.

## **3.0.22 Packed Output FIFO**

The *Packed Output FIFO* is used to temporarily store packed data for a code-block while the Tier I core waits for the Group 2 connections to provide the core access to the *Primary Output FIFO*. As shown in Figure 3.39, the *Packed Output FIFO* is 256 elements deep and 128 bits wide.

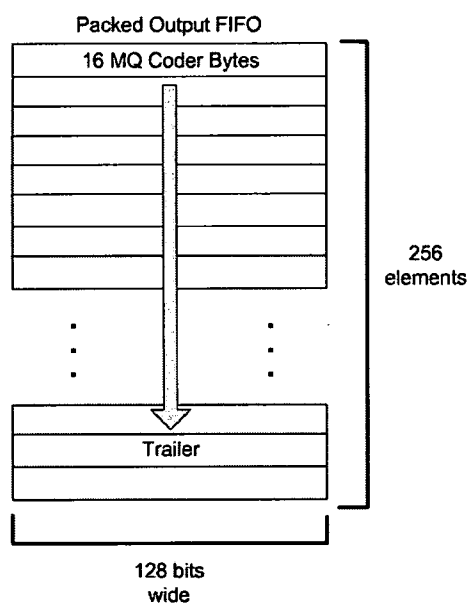


Figure 3.39: Depiction of how data is stored in the *Packed Output FIFO*.

## CHAPTER 4

### Hardware Performance

The success merit of the proposed Tier I hardware compression system is measured primarily by the compression time reduction over an all-software implementation. The software base used in the Tier I timing comparisons is a commercial implementation of a JPEG2000 compressor produced by Intel and optimized for the x86 processor architecture. All timing analysis is performed on a Dell PowerEdge 2950 server with 2 Intel Quad-Core Xeon X5460 processors running at 3158.76 MHz, 16 GB of RAM, and Linux RedHat OS. The timing results presented in this section are for a single process/single thread implementation for both the all-software and software-plus-hardware implementations. The FPGA add-on card uses an 8 lane PCIe expansion slot, with a theoretical 250 MB/sec throughput per lane. However, observed PCIe speeds for the FPGA board are closer to 80 MB/sec per lane for data sent to the board and 166.25 MB/sec per lane for data read back. The FPGA board provides four Altera Stratix III EP3SL150 FPGAs for user logic (Tier I core references are therefore x4). All hardware designs are configured and built using Altera's Quartus II 8.1 design suite.

As with most projects, the initial goal of the Tier I hardware acceleration effort has been to obtain a functional FPGA design. This task has been accomplished and verified by passing hardware encoded images through commercial decoders, such as

IrfanView, OPJViewer, and ImageMagick, and observing properly decoded images. Optimizing the design is a future objective, and potential improvements are discussed in Chapter 5.

The Altera Stratix III EP3SL150 FPGAs configured with the Tier I cores proved to be adequate in terms of on-chip memory and logic resources. For 2, 16, and 31 Tier I cores, the FPGA resource statistics are presented in Table 4.1. These numbers include the logic for the proprietary memory controller used to access the external memory. A small, medium, and large number of cores is used to provide a wide range of performance results. 31 Tier I cores is the maximum number that a Stratix III EP3SL150 FPGA currently holds.

Table 4.1: Altera Stratix III EP3SL150 FPGA Resource Utilization for 2, 16, and 31 Tier I Cores

<b>2 Tier I Cores</b>	
Logic Utilization	22%
Block Memory Utilization	8%
<b>16 Tier I Cores</b>	
Logic Utilization	60%
Block Memory Utilization	28%
<b>31 Tier I Cores</b>	
Logic Utilization	91%
Block Memory Utilization	49%

When evaluating a hardware design one of chief concerns is the clock speed the design operates reliably at. Currently, the development tools provide a maximum clock frequency of 106 MHz for a configuration of 31 Tier I cores. The highest clock frequency on the FPGA board under 106 MHz is 87.5 MHz, so the Tier I cores currently operate at 87.5 MHz. Timing analysis provided by the development tools

shows that the *MQ Coder* is currently responsible for the relatively low maximum operational frequency.

Even with the current unoptimized Tier I design, the timing results provided in Tables 4.2 through 4.5 (plotted in Figures 4.3 through 4.12) reveal success for compressing various  $1k \times 1k$  tile combinations of *peppers* and *pentagon* [17]. Figure 4.1 shows an example of four  $1k \times 1k$  *pentagon* images merged together to form a  $2k \times 2k$  image, and likewise is done to obtain the other image sizes. The images are compressed at a 10:1 ratio through quantization across subbands at multiple resolutions. For hardware processing, the images are broken into  $1k \times 1k$  tiles and are distributed evenly to the four FPGAs. Considering the  $2k \times 2k$  image, for example, each FPGA receives one  $1k \times 1k$  tile, resulting in not only code-block concurrent processing but tile processing concurrency as well. The *write to board* and *read from board* timings are included in Tables 4.2 and 4.4, since they introduce a time penalty for using the hardware. Therefore, the true total hardware Tier I time is the sum of the transfers and the Tier I processing time. It is very significant to note that transfers to the board currently finish before any processing begins. Therefore, the hardware waits idle while the external memory fills with tiles for all of the FPGAs, which is why the *write to board* time must be added to the Tier I processing time in its entirety. Fixing this inefficiency (i.e. overlapping transfers to the board and Tier I processing) is dependent on the memory controller provided by the board manufacturer. The manner in which to properly request data from the memory controller while it is loading data from a DMA is still under investigation.

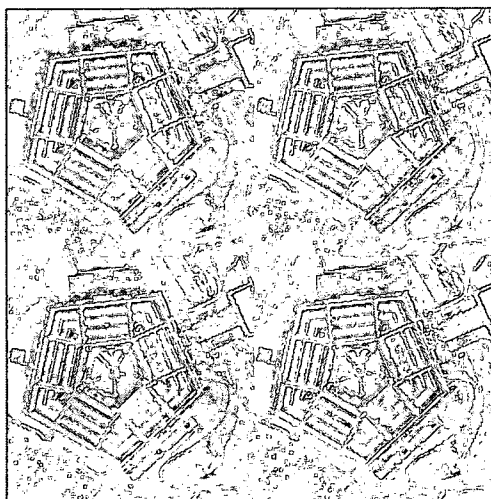


Figure 4.1:  $2k \times 2k$  *pentagon*.



Figure 4.2:  $6k \times 6k$  *peppers*.

Table 4.2: Hardware Timing for *peppers*.

	Peppers								
	2k x 2k			4k x 4k			6k x 6k		
	2 x4 Cores	16 x4 Cores	31 x4 Cores	2 x4 Cores	16 x4 Cores	31 x4 Cores	2 x4 Cores	16 x4 Cores	31 x4 Cores
Write to HW Time (ms)	14.44	13.76	14.33	51.85	51.35	51.58	114.45	114.5	114.23
HW Tier I Time (ms)	110.52	20.97	11.67	442.06	83.76	46.55	993.88	155.65	104.66
Read from HW Time (ms)	2.43	2.37	2.32	3.38	3.32	3.33	5.41	5.34	4.87
HW Tier I Time w/ Transfers (ms)	127.39	37.1	28.32	497.29	138.43	101.46	1113.74	275.49	223.76
Total Compression Time Using HW (ms)	221.84	131.52	123.98	870.28	511.18	474.33	1951.74	1140.81	1060.95

	Peppers					
	8k x 8k			10k x 10k		
	2 x4 Cores	16 x4 Cores	31 x4 Cores	2 x4 Cores	16 x4 Cores	31 x4 Cores
Write to HW Time (ms)	202.36	202.01	202.49	313.33	313.21	313.79
HW Tier I Time (ms)	1936.95	401.18	227.75	2762.79	523.31	290.75
Read from HW Time (ms)	7.09	7.52	7.26	9.81	10.03	10.08
HW Tier I Time w/ Transfers (ms)	2146.4	610.71	437.5	3085.93	846.55	614.62
Total Compression Time Using HW (ms)	3646.45	2114.25	1945.21	5413.81	3176.9	2939.93

Table 4.3: Software Timing for *peppers*.

	Peppers				
	2k x 2k	4k x 4k	6k x 6k	8k x 8k	10k x 10k
SW Tier I Time (ms)	143.02	575.84	1313.17	2367.36	3537.3
Total SW Only Compression Time (ms)	211.87	847.93	1961.28	3460.19	5222.78

Table 4.4: Hardware Timing for *pentagon*.

	Pentagon								
	2k x 2k			4k x 4k			6k x 6k		
	2 x4 Cores	16 x4 Cores	31 x4 Cores	2 x4 Cores	16 x4 Cores	31 x4 Cores	2 x4 Cores	16 x4 Cores	31 x4 Cores
Write to HW Time (ms)	14.33	13.7	13.86	51.71	51.97	52.01	114.91	114.44	114.1
HW Tier I Time (ms)	117.12	18.54	10.57	468.35	74.04	42.07	1053.73	166.53	94.56
Read from HW Time (ms)	2.45	2.4	2.39	3.37	3.48	3.32	4.94	5.08	5.03
HW Tier I Time w/ Transfers (ms)	133.9	34.64	26.82	523.43	129.49	97.4	1173.58	286.05	213.69
Total Compression Time Using HW (ms)	230.8	131.14	123.99	906.54	511.61	481.53	2031.04	1145.9	1072.2

	Pentagon					
	8k x 8k			10k x 10k		
	2 x4 Cores	16 x4 Cores	31 x4 Cores	2 x4 Cores	16 x4 Cores	31 x4 Cores
Write to HW Time (ms)	202.16	202.67	202.29	313.95	313.42	313.98
HW Tier I Time (ms)	1873.27	296.01	168.06	2926.96	462.5	262.56
Read from HW Time (ms)	7.48	7.15	7.33	10.3	10.47	10.68
HW Tier I Time w/ Transfers (ms)	2082.91	505.83	377.68	3251.21	786.39	587.22
Total Compression Time Using HW (ms)	3611.81	2035.75	1906.75	5628.01	3166.27	2954.88

Table 4.5: Software Timing for *pentagon*.

	Pentagon				
	2k x 2k	4k x 4k	6k x 6k	8k x 8k	10k x 10k
SW Tier I Time (ms)	153.11	600.29	1344.78	2443.21	3787.2
Total SW Only Compression Time (ms)	223.78	880.8	1969.22	3564.38	5519.88

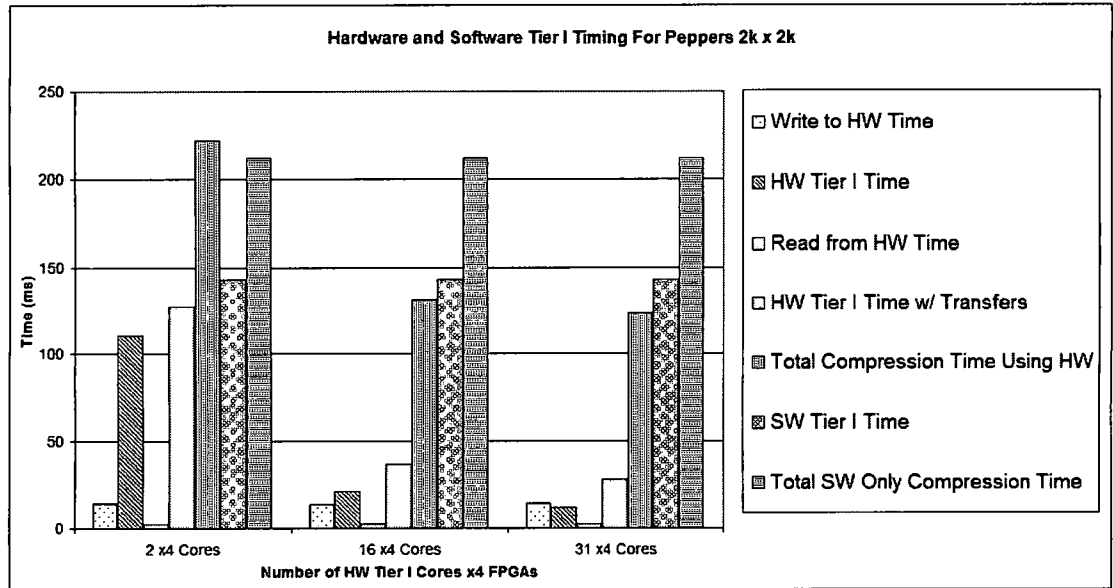


Figure 4.3: Hardware and software timing comparisons for *peppers* 2k × 2k.

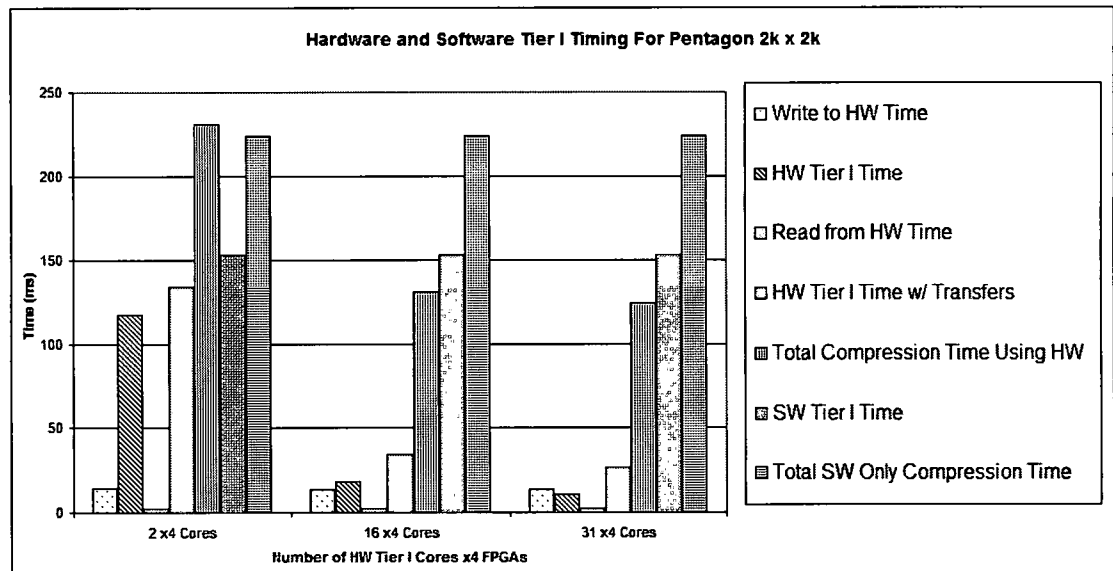


Figure 4.4: Hardware and software timing comparisons for *pentagon* 2k × 2k.

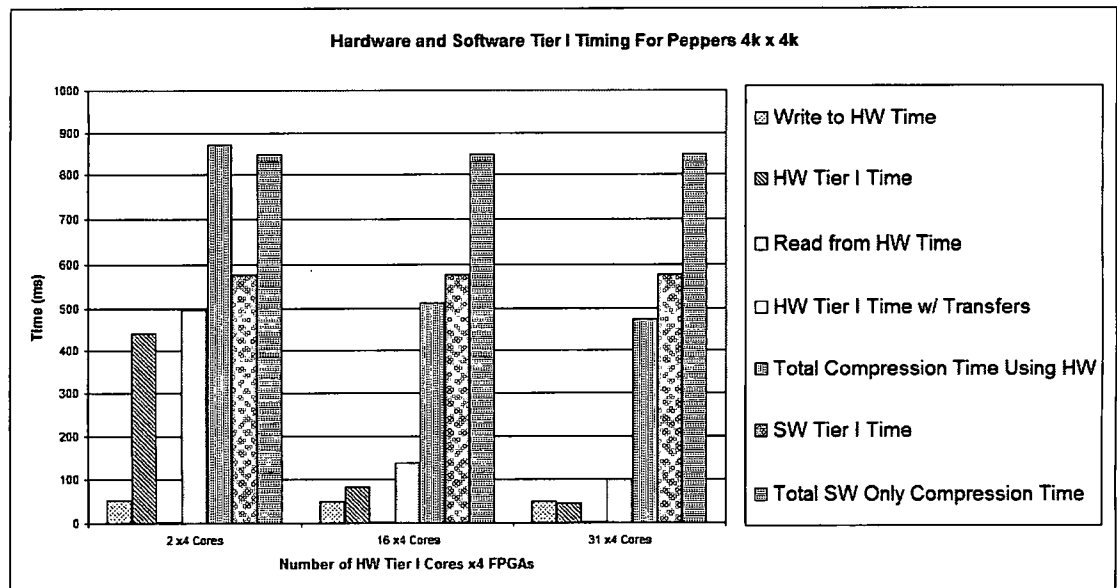


Figure 4.5: Hardware and software timing comparisons for *peppers* 4k × 4k.

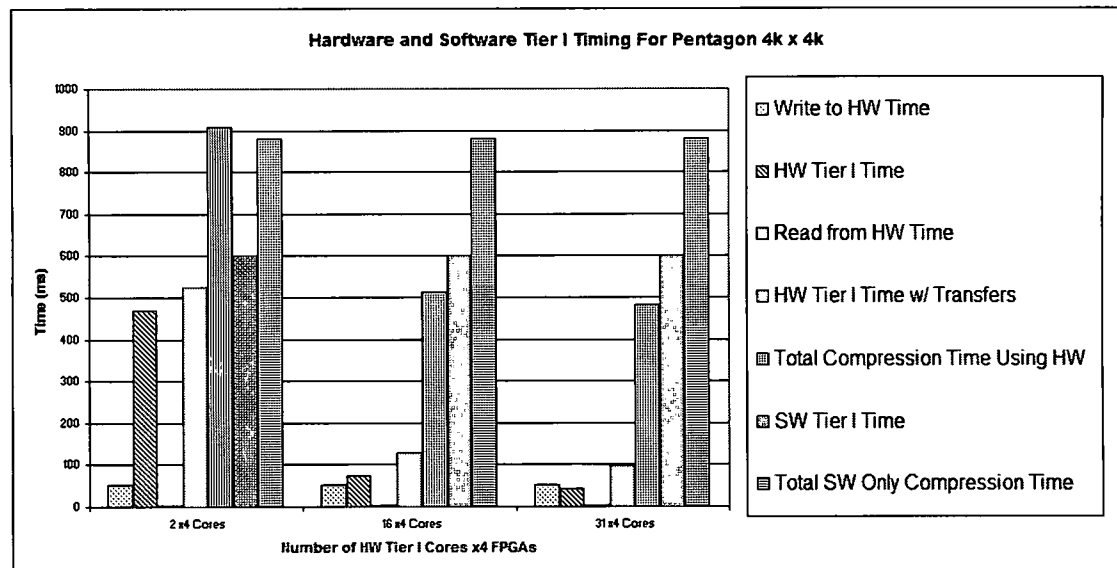


Figure 4.6: Hardware and software timing comparisons for *pentagon* 4k × 4k.

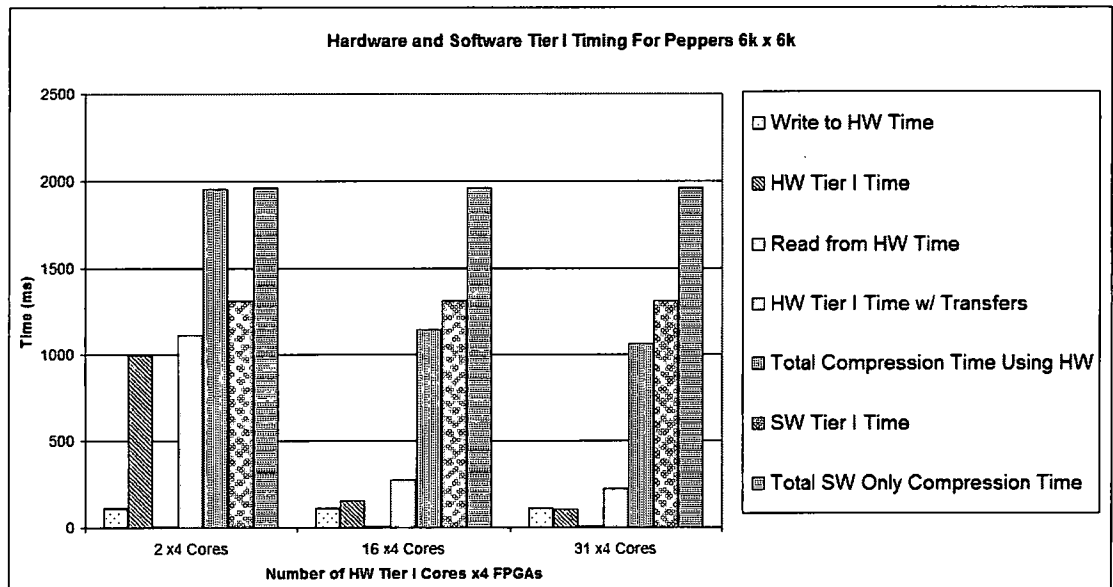


Figure 4.7: Hardware and software timing comparisons for *peppers* 6k x 6k.

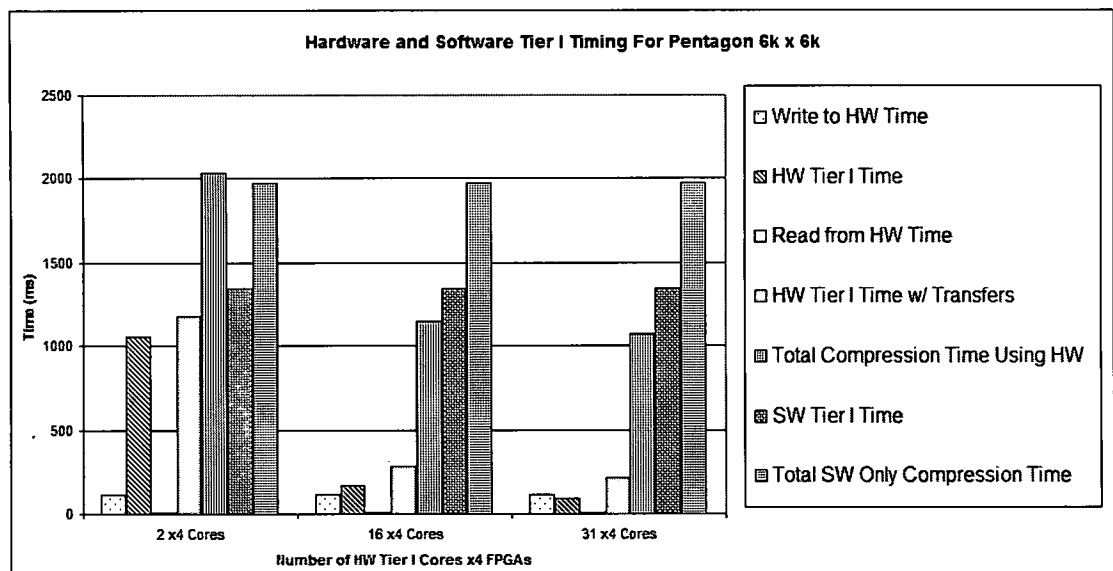


Figure 4.8: Hardware and software timing comparisons for *pentagon* 6k x 6k.

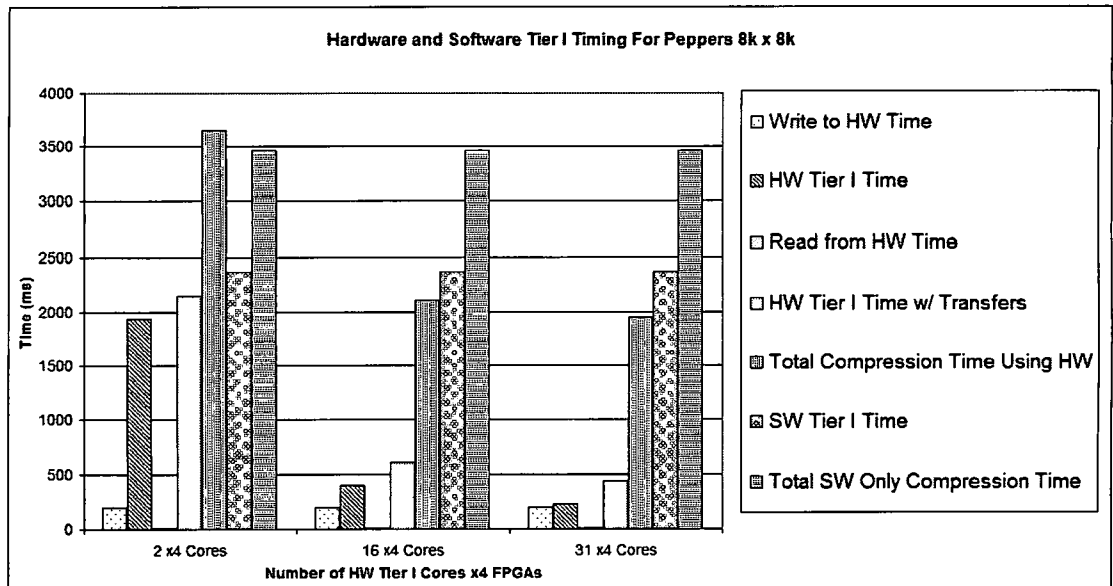


Figure 4.9: Hardware and software timing comparisons for *peppers* 8k × 8k.

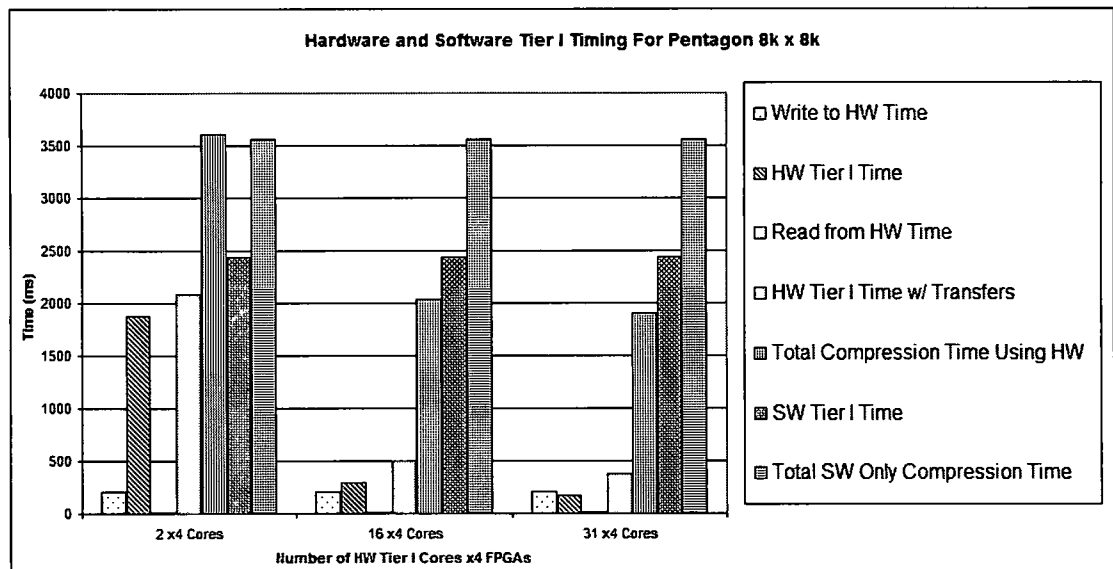


Figure 4.10: Hardware and software timing comparisons for *pentagon* 8k × 8k.

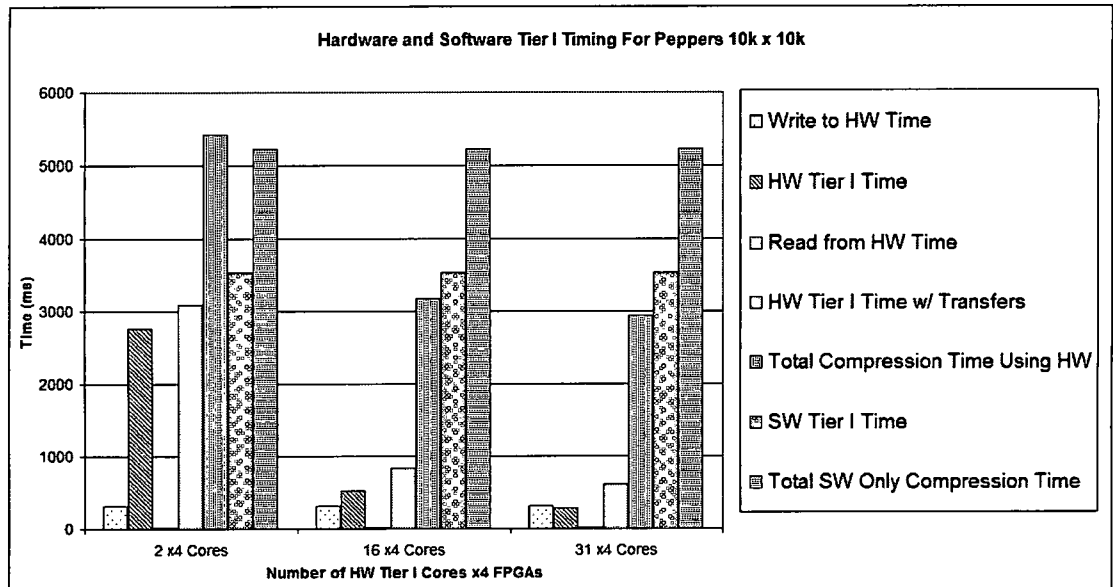


Figure 4.11: Hardware and software timing comparisons for *peppers* 10k × 10k.

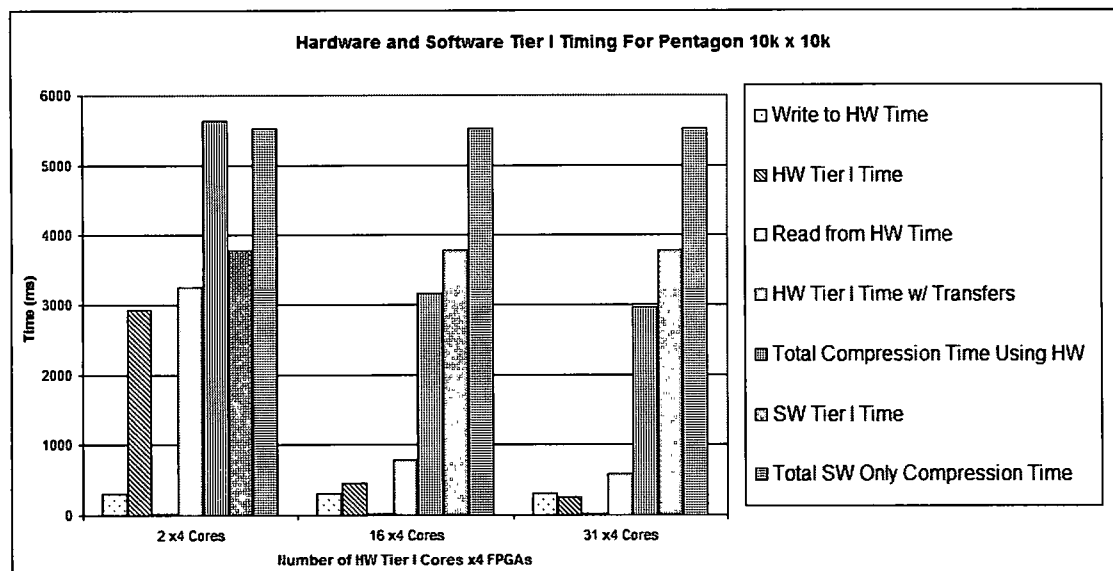


Figure 4.12: Hardware and software timing comparisons for *pentagon* 10k × 10k.

The plots in Figures 4.3 through 4.12 clearly show that for Tier I processing the FPGA concurrent processing outperforms the software in all cases. For the 2 Tier I cores per FPGA (8 cores total), while the Tier I processing including transfers is faster than the software, the overall compression time for the images in all but one case falls short of the software. The reason for this seeming inconsistency is that the software must reorganize the tiles in preparation for the DMA transfer as well as parse through compressed tiles that are read back, which adds overhead the all-software version does not have. However, as the number of Tier I cores increases, the hardware Tier I processing is so much faster than the software that the reorganization and parsing overhead is consumed. In all cases for the 16 x4 and 31 x4 Tier I cores, the entire image is compressed in a shorter time than it takes the software to perform only Tier I processing. Figures 4.13 and 4.14 show the trend of Tier I processing times for the software and hardware (including transfers) versus image resolution.

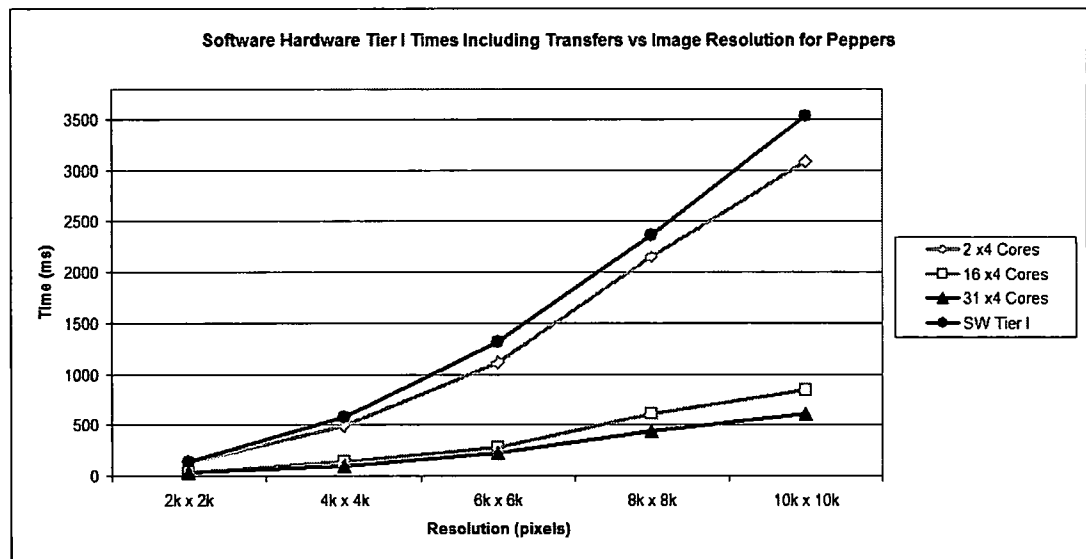


Figure 4.13: Hardware and software Tier I timing comparisons for *peppers*.

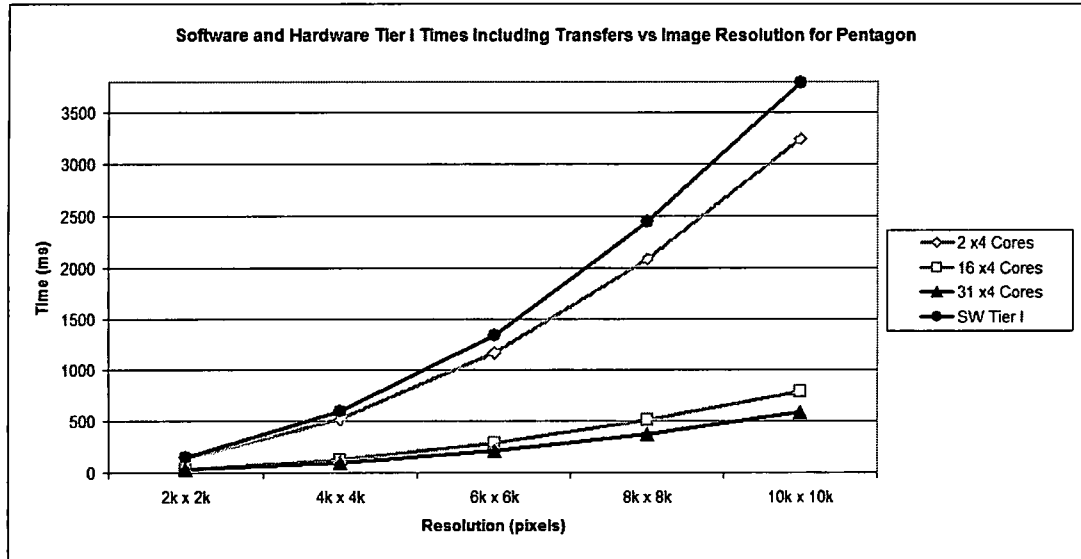


Figure 4.14: Hardware and software Tier I timing comparisons for *pentagon*.

Figures 4.15 and 4.16 show that 31 x4 Tier I cores nearly cuts the overall image compression time in half (45% speed improvement range). The 16 x4 Tier I cores are slightly less effective in the scope of entire image compression time, introducing a speed improvement in the 40% range. Tables 4.6 and 4.7 summarize the timing results as percentage improvements for the hardware compared to the Intel software base. For Tier I times, whether transfers are included or not, the hardware outperforms the software in every case. For the 31 x4 cores, the increase in Tier I performance without including transfers is as high as 93%. The overhead of waiting for writing all of the tiles to the FPGA external memory before beginning processing lowers this improvement to around 82%, which makes the overlapping of the processing with the writes to the board all the more desirable.

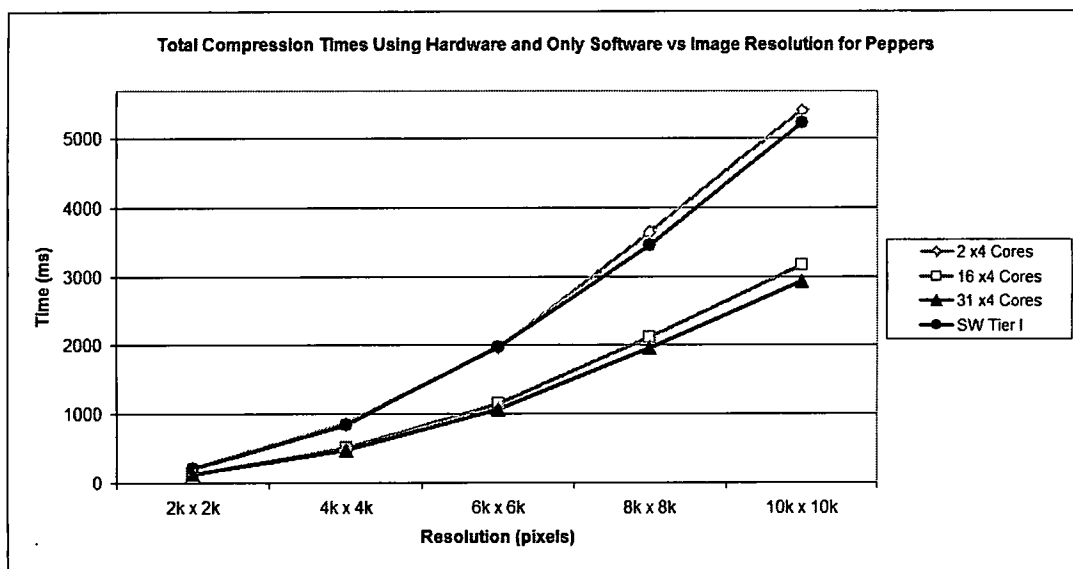


Figure 4.15: Hardware and software timing comparisons for beginning-to-end compression for *peppers*.

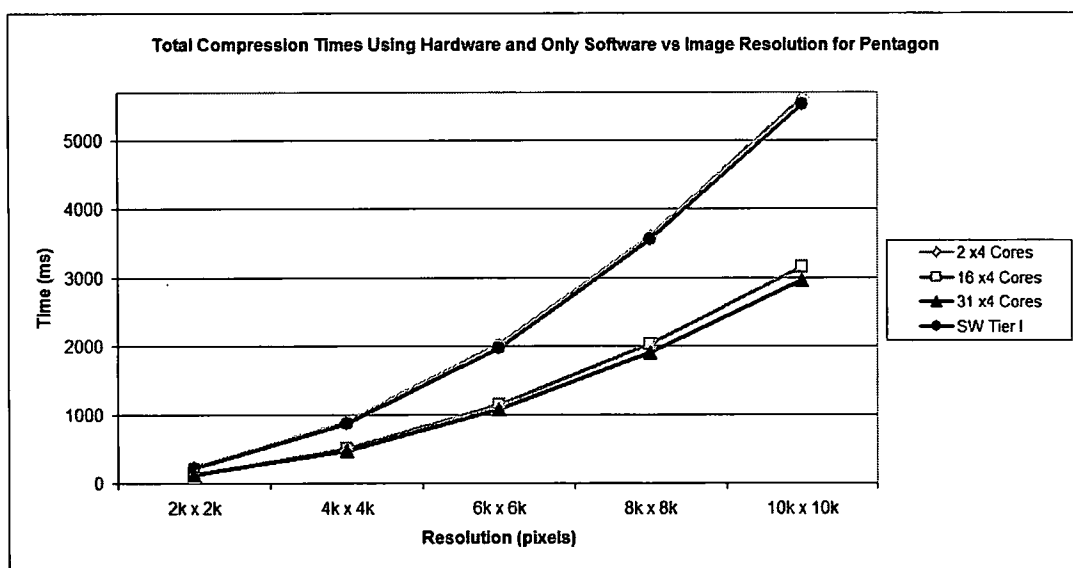


Figure 4.16: Hardware and software timing comparisons for beginning-to-end compression for *pentagon*.

Table 4.6: Percentage Time Improvements for the Hardware Tier I Implementation vs Software for *peppers*.

Resolution	Peppers								
	Tier I Time Improvement (%)			Tier I Time Improvement w/ Transfers (%)			Overall Time Improvement (%)		
	Number of Tier I Cores			Number of Tier I Cores			Number of Tier I Cores		
	2 x4	16 x4	31 x4	2 x4	16 x4	31 x4	2 x4	16 x4	31 x4
2k x 2k	22.72	85.34	91.84	10.93	74.06	80.20	-4.71	37.92	41.48
4k x 4k	23.23	85.45	91.92	13.64	75.96	82.38	-2.64	39.71	44.06
6k x 6k	24.31	88.15	92.03	15.19	79.02	82.96	0.49	41.83	45.91
8k x 8k	18.18	83.05	90.38	9.33	74.20	81.52	-5.38	38.90	43.78
10k x 10k	21.90	85.21	91.78	12.76	76.07	82.62	-3.66	39.17	43.71

Table 4.7: Percentage Time Improvements for the Hardware Tier I Implementation vs Software for *pentagon*.

Resolution	Pentagon								
	Tier I Time Improvement (%)			Tier I Time Improvement w/ Transfers (%)			Overall Time Improvement (%)		
	Number of Tier I Cores			Number of Tier I Cores			Number of Tier I Cores		
	2 x4	16 x4	31 x4	2 x4	16 x4	31 x4	2 x4	16 x4	31 x4
2k x 2k	23.51	87.89	93.10	12.55	77.38	82.48	-3.14	41.40	44.59
4k x 4k	21.98	87.67	92.99	12.80	78.43	83.77	-2.92	41.92	45.33
6k x 6k	21.64	87.62	92.97	12.73	78.73	84.11	-3.14	41.81	45.55
8k x 8k	23.33	87.88	93.12	14.75	79.30	84.54	-1.33	42.89	46.51
10k x 10k	22.71	87.79	93.07	14.15	79.24	84.49	-1.96	42.64	46.29

## CHAPTER 5

### Future Work and Conclusions

#### 5.1 Future Work

As indicated in Chapter 4, the proposed hardware architecture for Tier I encoding is completely functional, compliant with the JPEG2000 standard, and much faster than an all-software implementation running on a single CPU core. However, due to time constraints the design presented in this documentation has not yet been optimized for clock count and clock rate, or compared to a multi-CPU implementation. Several specific areas are known to possess the capability of being improved upon. The first area concerns the transfer of data from the host software to the FPGA board. Currently, all of the uncompressed tiles are sent to the FPGAs' external memory banks before the FPGAs begin to process them. Overlapping the transfers and processing is critical for significant compression performance gains. For processing a 100 megapixel image, 53% of the 31 x4 Tier I encoding time is spent waiting for data to be transferred to the board. The solution to this problem resides in investigating the interface to the *Primary Input FIFO* and conducting further testing.

Another inefficiency lies in the way empty code-blocks are found. The current design assumes the sign bits of every code-block must be loaded into the *PSDX RAM*. Therefore, for every stripe four clock cycles are consumed while loading the

sign bits. At the same time, the stripe bits are examined for the ongoing  $K$  calculation, which determines whether an all zero code-block is present. Quantization can produce substantial numbers of empty code-blocks in the LH, HL, and HH subbands, so clock cycles are wasted loading sign bits for those code-blocks. An alternative method would be to modify the *K Calculator* to be able to take in new data on consecutive clocks and determine if a zero code-block is present in one-fourth the current time. However, for non-zero code-blocks, a 1024 clock cycles would be added to each code-block's processing time since the *K Calculator* and *Sign Loader* would no longer be operating in parallel. The trade-off analysis has not yet been investigated.

In order to reduce processing the software must complete before transferring data to the FPGA board, a hardware module could be added to perform the two's complement to sign-magnitude conversion on code-block stripes immediately after they are read from the *Primary Input FIFO*. Other options for expansion include adding the DWT and rate-distortion tracking to hardware. Quantization by powers of 2 could easily be implemented using shift registers. To improve the maximum clock rate beyond 106 MHz, a new version of the *MQ Coder* is being developed. Investigation of other reasons for the limited clock rate is ongoing.

## 5.2 Conclusions

In conclusion, the JPEG2000 Tier I FPGA architecture proposed in this thesis has proven to be fully functional and significantly reduce the amount of time required to encode tiles compared to the likely fastest software implementation available for a single CPU core—one built upon Intel's Integrated Performance Primitives running on an Intel processor. While all inefficiencies have not yet been eliminated, the abundant

parallelism of the design operating on four FPGAs still introduces greatly improved performance. Using 16 Tier I cores on four FPGAs, raw Tier I improvement is on the order of 87%, not including transfers to the board. 31 x4 Tier I cores are 93% faster than the software Tier I, not including transfers to the board. Incorporating the DMA transfer times and extra software overhead for organizing data sent to and read from the FPGA board, overall compression time improvements are on the order of 40% for 16 x4 Tier I cores and 45% for 31 x4 Tier I cores. In other words, using 31 x4 Tier I cores nearly cuts the amount of time required to compress an image in half. Although a testing environment incorporating multiple CPU cores has not yet been developed, the results obtained by moving Tier I from software to FPGAs have shown hardware accelerated compression to be a worthwhile endeavor for the target wide area persistent surveillance application.

## BIBLIOGRAPHY

- [1] Acharya, T. and Tsai, P. *JPEG2000 Standard for Image Compression: Concepts, Algorithms, and VLSI Architectures*. John Wiley & Sons, Inc., Hoboken, NJ, 2005.
- [2] Chai, D. and Bouzerdoun, A. JPEG2000 Image Compression: An Overview. In *Australian and New Zealand Intelligent Information Systems Conference (ANZISCI2001)*, pages 237–241, November 2001.
- [3] Christopolous, C. and Skodras, A. and Ebrahimi, Touradj. The JPEG2000 Still Image Coding System: An Overview. In *IEEE Transactions on Consumer Electronics*, volume 46, pages 1103–1127, November 2000.
- [4] Cruz, D. and Ebrahimi, T. An Analytical Study of JPEG 2000 Functionalities. In *Proceedings of ICIP 2000*, 2000.
- [5] Duttweiler, D. and Christodoulos, C. Probability Estimation in Arithmetic and Adaptive-Huffman Entropy Coders. In *IEEE Transactions on Image Processing*, volume 4, pages 237–246, March 1995.
- [6] Flaherty, M. The 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, May 2006.
- [7] Gonzalez, R. and Woods, R. *Digital Image Processing: Second Edition*. Prentice Hall, Upper Saddle River, NJ, 2002.
- [8] Gormish, M. Lee, D. and Marcellin, M. JPEG2000: Overview, Architecture, and Applications. In *Proceedings of the IEEE*, 2000.
- [9] Khademi, A. and Krishnan, S. Comparison of JPEG2000 and Other Lossless Compression Schemes for Digital Mammograms. In *Proceedings of the IEEE Engineering in Medicine and Biology 27th Annual Conference*, pages 3771–3774, September 2005.
- [10] Lee, D. JPEG2000: Retrospective and New Developments. In *Proceedings of the IEEE*, volume 93, January 2005.

R002594554

- [11] Mundy, D. The Study and HDL Implementation of the JPEG2000 MQ Coder. Master's thesis, University of Dayton, 300 College Park, Dayton, OH 45440, May 2007.
- [12] Roa, K. and Huh, Y. JPEG2000. In *4th EURASIP - IEEE Region 8 International Symposium on Video/Image Processing and Multimedia Communications*, June 2002.
- [13] Sweldens, W. The Lifting Scheme: A Construction Of Second Generation Wavelets, 1997.
- [14] Symes, P. *Digital Video Compression*. McGraw-Hill Companies, 2003.
- [15] Taubman, D. and Marcellin, M. *JPEG2000: Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, Norwell, MA, 2002.
- [16] Taubman, D. and Marcellin, M. JPEG2000: Standard for Interactive Imaging. In *Proceedings of the IEEE*, volume 90, pages 1336–1357, August 2002.
- [17] University of Southern California—Signal & Image Processing Institute. The USC-SIPI Image Database. <http://sipi.usc.edu/database/>, May 2009.
- [18] Varma, K. and Bell, A. JPEG2000—Choices and Tradeoffs for Encoders. *IEEE Signal Processing Magazine*, pages 70–75, November 2004.
- [19] Walker, J. *A Primer on Wavelets and Their Scientific Applications*. Chapman & Hall/CRC, Boca Raton, FL, 1999.
- [20] Yang, G. and Zheng, N. and Li, C. and Guo, S. Extensible JPEG2000 Image Compression Systems. In *IEEE National Conference on Industrial Technology (ICIT2005)*, pages 1376–1380, 2005.
- [21] Information Technology—Digital Compression and Coding of Continuous-Tone Still Images—Requirements and Guidelines. Technical report, CCITT Recommendation | ISO/IEC International Standard, Septmeber 1992.
- [22] Information Technology—JPEG2000 Image Coding System: Core Coding System. Technical report, ISO/IEC 15444-1 International Standard, Septmeber 2004.