University of Stuttgart

Faculty of Computer Science

Master Thesis Nr. 3195

Parallel architectural design space exploration for real-time image compression

Mahesh Krishnappa

Course of Study:	Information Technology
Examiner:	Prof. Dr. Sven Simon
Supervisor:	Dipl. Inf. Simeon Wahl
Commenced:	May 01, 2011
Completed:	December 15, 2011
CR-Classification:	E.4, F.1.2, I.3.1, I.4.2

Institut für Parallele und
Verteilte Systeme
Abteilung Parallele Systeme
Universitätsstraße 38
D-70569 Stuttgart

Acknowlegement

I would like to express special thanks to my supervisor, Mr. Dipl.-Inf. Simeon Wahl. His valuable inputs and support were vital for successful completion of my master thesis. I would also like to thank the members of "Institute for Parallel and Distributed Systems" for their kind co-operation and support which helped me in completion of my thesis.

I would like to express my gratitude to Prof. Dr.-Ing. Sven Simon for providing me a opportunity to work on this topic.

I would like to extend my gratitude to my parents and friends for their support and encouragement which helped me greatly in successful completion of this thesis.

Mahesh Krishnappa

Abstract

Embedded block coding with optimized truncation (EBCOT) is a coding algorithm used in JPEG2000. EBCOT operates on the wavelet transformed data to generate highly scalable compressed bit stream. Sub-band samples obtained from wavelet transform are partitioned into smaller blocks called code-blocks. EBCOT encoding is done on blocks to avoid error propagation through the bands and to increase robustness. Block wise encoding provides flexibility for parallel hardware implementation of EBCOT. The encoding process in JPEG2000 is divided into two phases: Tier 1 coding (Entropy encoding) and Tier 2 coding (Tag tree coding).
This thesis deals with design space exploration and implementation of parallel hardware architecture of Tier 1 encoder used in JPEG2000. Parallel capabilities of Tier-1 encoder is the motivation for exploration of high performance real time image compression architecture in hardware. The design space covers the following investigations:

- The effect of block-size in terms of resources, speed, and compression performance,
- Computational performance.

The key computational performance parameters targeted by the architecture are

- significant speedup compared to a sequential implementation,
- minimum processing latency and,
- minimum logic resource utilization.

The proposed architecture is developed for an embedded application system, coded in VHDL and synthesized for implementation on Xilinx FPGA system.

Contents

List of Tables

List of Figures

1. INTRODUCTION

Joint Photographic Experts Group (JPEG) is a still image compression standard developed in collaboration by the International Organization for Standardization (ISO), and the International Electrotechnical Commission (IEC) and recommended by International Telecommunication Union (ITU). JPEG supports compression of still images with most image sizes in any color space. It aims at achieving compression performance with user-adjustable compression ratio, lossless compression mode and excellent reconstruction quality. Although JPEG compression standard is very successful for more than a decade, there has been significant innovation towards development of technology related to multimedia, internet and communication. JPEG in spite of its excellent performance lacks some of the desired features related to multimedia, internet and communication platforms.

JPEG2000 [1, 4] was developed to overcome the limitation of JPEG, it offers high performance in the application domain such as Internet, medical imaging, mobile multimedia communication, satellite imagery, digital photography, digital library, image archival, 3g cellular telephony, client-server networking, graphics etc. Some of the important features of JPEG2000 are:

- Lossless and lossy compression: Unified compression architecture which supports lossless or lossy modes of compression and decompression.
- Low bit rate compression performance: It can achieve better image quality for a particular compression ratio as compared to JPEG. It offers high Peak signal-to-noise ratio (PSNR)and image quality as compared to JPEG.
- Large image size and components: It can support image size up to $(2^{32}-1) \times (2^{32}-1)$ and image components up to 2^{14}. This feature supports efficient processing of satellite and astronomical images.
- Progressive transmission by resolution and image quality: It offers flexibility in organizing bit-stream in progressive mode of image quality. This feature allows efficiency and flexibility in real time browsing of images on the internet.
- Random access and compressed domain processing: It offers flexibility in compressed-domain processing such as cropping, rotation, scaling, feature extraction, etc. This can be achieved by randomly accessing and modifying code-block from the compressed bit-stream.
- Region of interest (ROI) coding: It offers flexibility in selectively encoding specific region of image with higher quality as compared to rest of the image.
- Robustness to bit-errors: JPEG2000 offers error detection and correction features within the code-block. Robustness is achieved by coding independent small-size code-blocks and using markers for resynchronization in the bitstream

Some of the limitations and drawbacks of JPEG2000 are:

- JPEG2000 requires huge computational resource.

- JPEG2000 requires very high computational time.
- JPEG2000 is very complex standard, hence requires huge development time.
- JPEG2000 introduces artifacts such as blocking and blurring in the compressed images for higher compression ratio as compared to other compression standards such as H.264/ Advanced Video Coding (AVC) and Portable Network Graphics (PNG)[5, 6]
- JPEG2000 has memory bandwidth constraints[6] for high quality and high bandwidth applications, since wavelet transforms is global transform phenomenon the memory bandwidth requirement is high as compared to AVC.
- For applications with communication bandwidth constrains, JPEG2000 with lossy compression (20:1) is used, the quality of the image degrades as compared to H.264 at the same compression ratio [7].

H.264/AVC compression standard is known for excellent image quality at low bit rates. It is based on a block-based integer Discrete Cosine Transform (DCT) analogous to DWT transform used in JPEG2000. Improvement in coding performance is mainly achieved by the prediction block in the image processing chain of H.264/AVC. Inter and intra block prediction phenomena are the key for achieving high quality compressed image with low bit-rate.

JPEG-LS [8] is simple lossless compression algorithm as compared to JPEG2000. JPEG-LS offers lossless compression at less computation time as compared to JPEG2000, however JPEG-LS does not support rich set of features such as scalability, error resilience, progressiveness, etc supported by JPEG2000. The significant enhancement in compression time is achieved by low-complexity processing blocks based on adaptive prediction, context modeling and Golomb coding.

This thesis aims at optimizing the computational resource and enhancing compression time of JPEG2000 by proposing parallel hardware architecture for EBCOT encoder used in entropy encoding phase. Profiling of software implementation of JPEG2000 in subsection 3.3.3 shows that Tier-1 coding phase in JPEG2000 is responsible for considerable portion of overall compression time. Synthesis results discussed under subsection 4.3 shows that EBCOT encoder in Tier-1 coding phase is computational expensive. Section 5 discusses several parallel architectures to improve image compression performance of JPEG2000 encoder.

2. EBCOT ALGORITHM

2.1. Introduction

JPEG2000 aims at compression algorithm that compresses images once and offers flexibility in decompression for different application with multiple image quality and size. It also offers lossless and lossy image compression modes. However JPEG2000 is computationally exhaustive compared JPEG algorithm, this is one of the major drawback of JPEG2000.

The block diagram of JPEG2000 encoder algorithm is as shown in Figure 1. It shows different phases of JPEG2000 compression algorithm, the compression phases can be mainly divided into three phases

- Discrete wavelet transform (DWT)
- Quantization and
- Entropy encoding.

Forward multicomponent transformation is done as a preprocessing transformation before the actual compression algorithm starts, in MultiComponent transformation the correlation between multiple components of the image is reduced. This increases the compression performance by reducing the redundant components in the image.

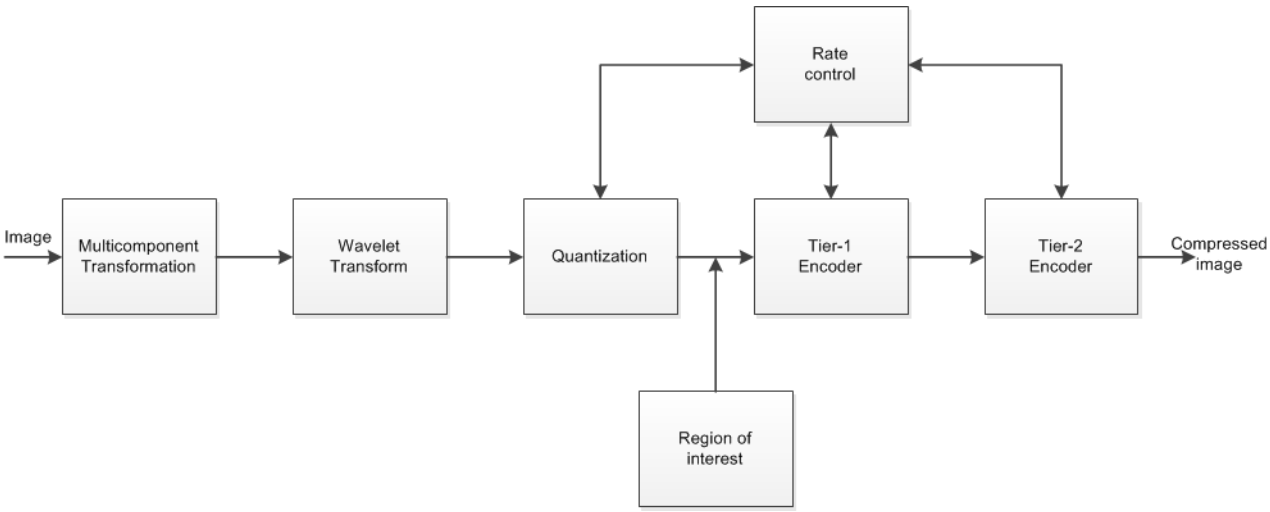

Figure 1: JPEG2000 encoder block diagram[2]

The DWT is done on the image components after preprocessing stage; DWT decomposes each component into number of sub-bands of different resolution levels. Figure 2 is an example of decomposition of image by DWT. DWT decomposes images into low frequency and high frequency sub-bands. Row-wise decomposition of the image results in low frequency (L) and high frequency (H) sub-bands as shown in the Figure 2. Further decomposition of the image

in column-wise results in 4 sub-bands LL1, LH1, HL1 and HH1. Decomposition of the image column-wise first and then row-wise results in same 4 sub-bands. LL1 sub-band represents coarser approximate of the original image and LH1, HL1, HH1 represents detailed high frequency approximate of the original image. LL1 sub-band can be further decomposed into 4 sub-bands recursively as shown in the Figure 3 a) and b).

Sub-bands from DWT are quantized independently and divided into number of code-blocks of fixed size as shown in Figure 4. Entropy encoding is applied on each of these sub-bands independently to generate compressed bit stream. Entropy encoding phase consists of Tier-1 and Tier-2 coding steps. Tier-1 coding decomposes the code-block into bit-planes as shown in Figure 4 and coding is done per bit-plane starting from most significant bit-plane to least significant bit-plane. Tier-1 coding consists of two steps, Fractional Bit-Plane Coding (BPC) and Binary Arithmetic Coding (BAC).

Embedded block coding with optimized truncation (EBCOT) algorithm is used for BPC coding and MQ-coder is used for BAC coding. Tier-2 coding engine is responsible for efficeint representation of encoding information associated with code-blocks such as block summary information, bitstream layers, truncation points between bitstream layers and most significant bit-plane. Tag Tree data structure is used by Tier-2 coding phase for efficient representation of information.

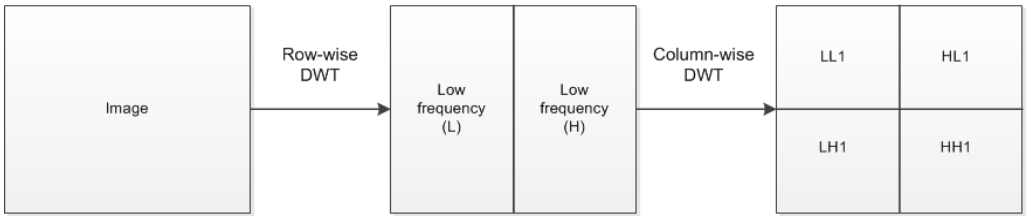

Figure 2: Image decompositon by DWT[2]

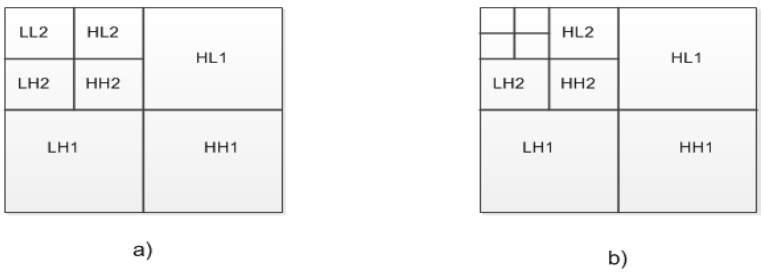

Figure 3: Multi level image decompositon by DWT[2]

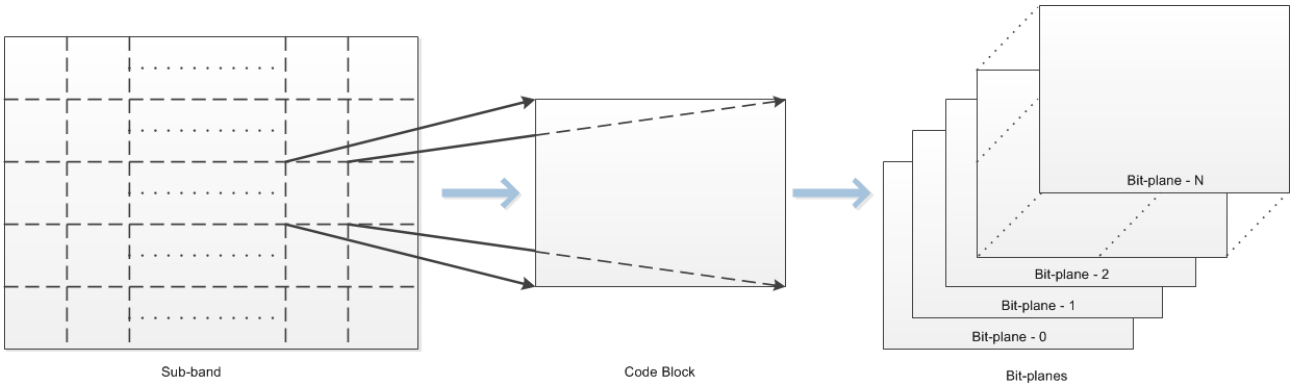

Figure 4: Sub-band decompositon into code-blocks and bit-planes

This chapter discusses EBCOT coding and MQ-coder algorithm in detail. The chapter introduces various terminologies used in the algorithm, basic encoding algorithms and coding stages. Coding stages uses these encoding algorithms to produce Context(CX) and Decision bit(D) outputs of EBCOT coding phase. CX and D bit information are encoded by MQ-coder to generate compressed bit-stream.

Entropy encoding in Tier-1 is done on wavelet sub-bands generated by DWT. The sub-bands are further divided into code-blocks of fixed size as shown in Figure 4. Pixel components of the sub-bands are represented in sign-magnitude representation of integers. Code-block dimensions are restricted by the standard, dimensions has to be power of 2 with minimum height and width being restricted to 2 and 4 and maximum height and width being restricted to 1024. Maximum elements of the code block is further restricted to 4096. 32×32 or 64×64 code-block size is recommended by standard for good compression performance.

Tier-1 coding in JPEG2000 consists of two processing stages, Bit Plane Coding (BPC) and Binary Arithmetic Coding (BAC). BPC coding is applied on each bit-planes of the code blocks, so the code-blocks are further decomposed into bit-planes as shown in Figure 4. If the resolution of each component is N bits then the code-blocks of each sub-bands are decomposed into N bit-planes and BPC is applied on each of these bit-planes of the code-block. BPC generates context and binary decision value as intermediate output. This intermediate data is input to BAC coding stage.

Embedded Block Coding with Optimized Truncation (EBCOT) algorithm [4] is adapted for implementation of BPC in JPEG2000.This algorithm aims at minimizing bitstream generated and minimizes the statistics to be maintained by exploiting the redundancies within and across bit planes of the code-block. EBCOT encoding is also called as fractional bit-plane coding because encoding is done in three different phases with no overlapping between the phases. Encoding phases of BPC in sequences of their processing are

- Significant Propagation Pass (SPP),
- Magnitude Refinement Pass (MRP) and

- Cleanup Pass (CUP).

These encoding phases will be discussed in detail in subsection 2.4.

2.2. Terminology

This subsection introduces some of the terms used for describing the algorithm.

Code-Block(y): Code-block is a two dimensional array of predetermined dimension. The array elements are the components of the image after DWT and quantization. The samples are represented in sign-magnitude form. Every sample of the code block are associated with σ, σ' and η(discussed in detail later) to indicate their status of encoding.

Sign Array (χ): Sign array is two dimensional array representing the signs of the sample of code-block. The dimension of sign array is same as that of code-block. Sample $\chi[m,n]$ represents the sign information of sample $y[m,n]$ of the code-block. Value of $\chi[m,n]$ is given as below

$$\chi[m,n] \begin{cases} 1 & \text{if y[m, n] } <0 \\ 0 & \text{otherwise} \end{cases}$$

$\chi[m,n]$ is assigned zero for the cases for which m and n are out of range of code-block size.

Magnitude Array (υ): υ is a two dimensional array of unsigned integer, the dimension of υ array is same as that of the code-block. The array element $\upsilon[m,n]$ represents the magnitude of the element at $[m,n]$ location of the code-block. The notation $\upsilon^p[m,n]$ is used to denote the P^{th} bit of $\upsilon[m,n]$.

Bit-Plane: The magnitude of the elements of the code-blocks are decomposed bitwise to form a plane called bit-plane. If the magnitude array (υ) is represented using P bits then the elements are decomposed into P different bit-planes.

Scan Pattern: Scan pattern defines the sequence in which the elements of the bit-plane are encoded or decoded. Bit-plane of a code-block is conceptually divided into sections: Each section consists of four rows and number of column is same as that of columns of the code-block. If the row dimension of code-block is not a multiple of 4 then all sections has 4 rows except the last section.

Encoding of elements of a code-block starts from first section downto last section. Within each section scan starts from first row first column downto first column fourth row and starts again from first row second column and so on until all the columns are scanned. There are two modes of scan pattern, Regular mode and Vertical causal mode. In regular mode after scan of last element of a section, it start with the first row first column element of the next section, here it uses the information from the previous section. In vertical causal mode each section will be scanned as a independent module, the information will not be shared across sections. Figure 5 shows an example of scan pattern for a bit-plane of a code-block.

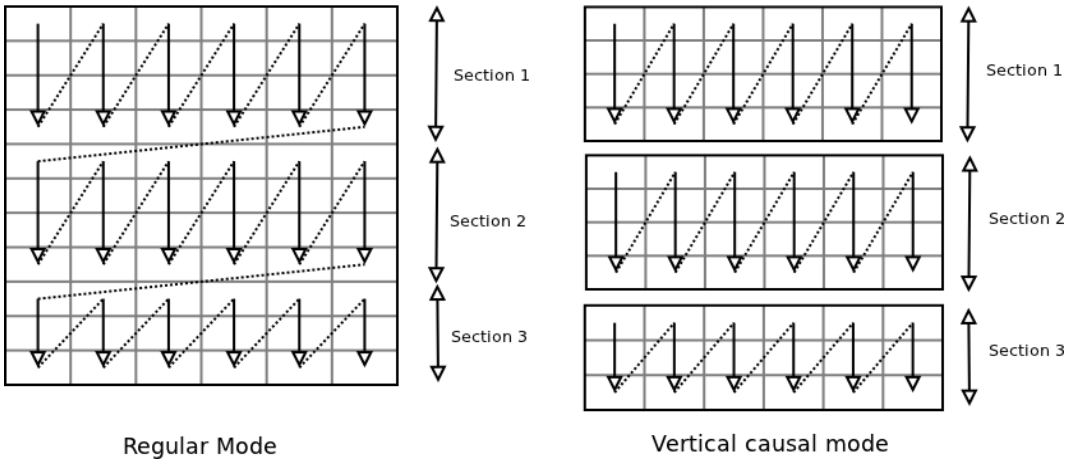

Figure 5: Bit-plane scan pattern

State Variables σ, σ' and η: These state variables are two-dimensional arrays with dimension same as that of the code-block. These state variables indicate the coding status of each element in the code-block during entropy encoding. Initially the values of these elements are cleared to zero and after completion of coding each code-block these variables are cleared again. The values of $\sigma[m,n]$ and $\sigma'[m,n]$ are set to 1 based on certain conditions but are not cleared to zero before the completion of coding of entire code-block. The values of variables $\eta[m,n]$ are cleared to zero after completion of coding of each bit-plane in a code-block. The interpretation of state variables σ, σ' and η are as below:

$\sigma[m,n]$

1: Indicates that first nonzero bit of $\upsilon[m,n]$ at row m and column n has been coded.

0: It either indicates that the first nonzero bit of $\upsilon[m,n]$ is not coded or m and n are out of range or invalid.

$\sigma'[m,n]$

1: Indicates that Magnitude Refinement Coding (MRC) has been applied to element $\upsilon[m,n]$.

0: Indicates that Magnitude Refinement Coding (MRC) has been not applied to element $\upsilon[m,n]$.

$\eta[m,n]$

1: Indicates that zero coding operation has been applied to $\upsilon^p[m,n]$ in Significant propagation pass.

0: Indicates that zero coding operation has been not applied to $\upsilon^p[m,n]$.

Preferred Neighborhood: An element $y[m,n]$ in the code-block is said to be in a preferred neighborhood if at least one of its eight adjacent neighbors has $\sigma[m,n]$ value equal to 1.

Zero Coding Tables: Zero coding tables are used for Zero coding operation. Context information is generated from zero coding operation. There are different zero coding table for different sub-bands, and the context information is generated from the significance states (σ) of the eight neighbors of an element being encoded. Table 1 shows an example of eight neighbors of an element X, if for example X is an element in LL or LH sub-band, and if the two horizontal neighbors have significance state value of 1. The context value 8 will be used as shown in Table 2. Similarly Table 3 and Table 4 are used for HL and HH sub-bands respectively.

Table 1: Neighborhood for zero coding context generation

D0	V0	D1
H0	X	H1
D3	V1	D2

Table 2: Zero Coding Context Table for Code-Blocks from LL and LH Subbands

LL and LH Subbands			Context Label
Σ H	Σ V	Σ D	CX
2	x	x	8
1	≥1	x	7
1	0	≥1	6
1	0	0	5
0	2	x	4
0	1	x	3
0	0	≥2	2
0	0	1	1
0	0	0	0

2.3. Coding operations

This subsections gives detailed explanation about the basic coding operations used by the EBCOT algorithm. There are four basic coding operations used by EBCOT depending on the coding phases, status of the state variables and current element's location. EBCOT algorithm produces context(CX) and decision bit(D) value as output. Context(CX) can take any value among 0 to 18 and decision bit(D) takes binary value 0 or 1. The basic coding operations are as below:

Table 3: Zero Coding Context Table for Code-Blocks from HL Subbands

HL Subbands			Context Label
Σ H	Σ V	Σ D	CX
x	2	x	8
≥1	1	x	7
0	1	≥1	6
0	1	0	5
2	0	x	4
1	0	x	3
0	0	≥2	2
0	0	1	1
0	0	0	0

Table 4: Zero Coding Context Table for Code-Blocks from HH Subbands

HH Subbands		Context Label
Σ(H + V)	Σ D	CX
x	≥3	8
≥1	2	7
0	2	6
≥2	1	5
1	1	4
0	1	3
≥2	0	2
1	0	1
0	0	0

Zero coding (ZC): In zero coding operation Decision bit is same as the $\upsilon^p[m,n]$. Context (CX) values is determined from one of the three Zero coding tables, depending on the type of sub-band (LL, LH, HL, HH) encoded. The context output from Zero coding can take a value from 0 to 8. The context value is decided by the significance states of eight neighbors of the current element encoded.

Sign coding (SC): SC computes CX and D value based on Horizontal reference value (H) and Vertical reference value (V). The value of H and V are calculated as shown in equation 1 and 2, with (m,n) being the current location of the element encoded.

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

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

The reference value calculated indicates three possible status of the neighbor elements. The neighbors for H reference calculation are horizontal adjacent elements and the neighbors for V reference calculation are vertical adjacent elements. A neighbor is said to be significant if state variable σ is equal to 1 and insignificant if σ is equal to 0. The H and V reference values are interpreter as below:

0: Indicates both neighbors are insignificant, or both neighbors are significant but have opposite signs.

1: indicates that one or both neighbors are significant with positive sign.

-1: indicates that's one or both neighbors are significant with negative sign.

Context (CX) and a binary value $\hat{\chi}$ is calculated from H and V values from the sign coding reference Table 5. Decision bit (D) is calculated by equation 3.

$$D = \hat{\chi} \oplus \chi[m,n] \tag{3}$$

Table 5: Sign coding Context 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 (MRC): Decision bit value (D) of an element at (m,n) is equal to value of $\upsilon^p[m,n]$, where p is the bit plane processed by MRC. The value of CX of an element at (m,n) is determined by $\sigma^{'}[m,n]$ and sum of values of state variable σ of its eight neighbors. The value of CX is determined from $\sigma^{'}$ and σ from the Table 6

Table 6: Magnitude Refinement Coding reference table

$\sigma^{'}[m,n]$	$\sigma[m-1,n]+\sigma[m+1,n]+\sigma[m-1,n-1]+\sigma[m-1,n+1]+\sigma[m+1,n-1]+\sigma[m+1,n+1]$	CX
1	x	16
0	≥ 1	15
0	0	14

Run Length Coding(RLC): Run length coding algorithm is capable of coding from one to four consecutive bit elements in the current scan pattern stripe. The number of bits coded depends on the position of first ”1” bit in the four consecutive bits. If all the four bits are ”0” then all of them are coded. If any of the bits are ”1” then all preceding ”0” until the bit ”1” is coded. In run length coding the number of D bits can be one or four depending on the number of ”1” in the scan pattern stripe.

The first D is equal to 0 if all four bits are 0, otherwise it is equal to 1. CX value in both of these cases is equal to 17. Additionally two more D bits are used to indicate the position of first 1 bit among four consecutive scan pattern bits. These two D bits are used with uniform context value 18. Table 7 summarizing the Context value for all four coding operations. Initial index portion of the Table 7 will be discussed in deatil under subsection 2.5.

Table 7: Coding Operations Context summary table

Operation	Context (CX)	Initial Index I(CX)
Zero Coding	0	4
	1	0
	2	0
	3	0
	4	0
	5	0
	6	0
	7	0
	8	0
Sign Coding	9	0
	10	0
	11	0
	12	0
	13	0
Magnitude Refinement Coding	14	0
	15	0
	16	0
Run-Length Coding UNIFORM	17	3
	18	46

2.4. EBCOT Coding Passes

EBCOT coding is done in three different coding passes, coding passes in their order of coding are Significance propagation pass (SPP), Magnitude refinement pass (MRP) and Cleanup pass (CUP). Each of these passes are applied to every bit-plane of the code-block except the most significant bit-plane. For most significant bit-plane only CUP coding pass is applied. In each

of the coding pass the bits are scanned as per scan pattern and encoded, after completion of one coding pass, the next coding pass restarts the scan from the beginning of the bit-plane of the code-block. Coding passes in order of coding in EBCOT algorithm are explained in detail below:

Significance Propagation pass (SPP): SPP is applied to every bit-plane except the most significant bit-plane. SPP checks for possibility of application of Zero coding(ZC) and Sign Coding(SC) to the current scan bit. ZC is applied if the current scan bit (m,n) is in preferred neighborhood and $\sigma[m,n] = 0$ and $\eta[m,n]$ is set to 1. SC is applied if $\upsilon^p[m,n] = 1$ and $\sigma[m,n]$ is set to 1. SPP scan is continued until all the bits in bit-plane are coded. Figure 6 shows the flow chart of SPP.

Magnitude Refinement Pass (MRP): MRP coding is applied to every bit-plane except the most significant bit-plane. MRP is applied to the current scan bit at position (m,n) if the state variables $\sigma[m,n] = 1$ and $\eta[m,n] = 0$, and $\sigma^{'}[m,n]$ is set to 1. MRP coding is continued until all the bits is the bit-plane are encoded. Figure 7 shows the flow chart of MRP.

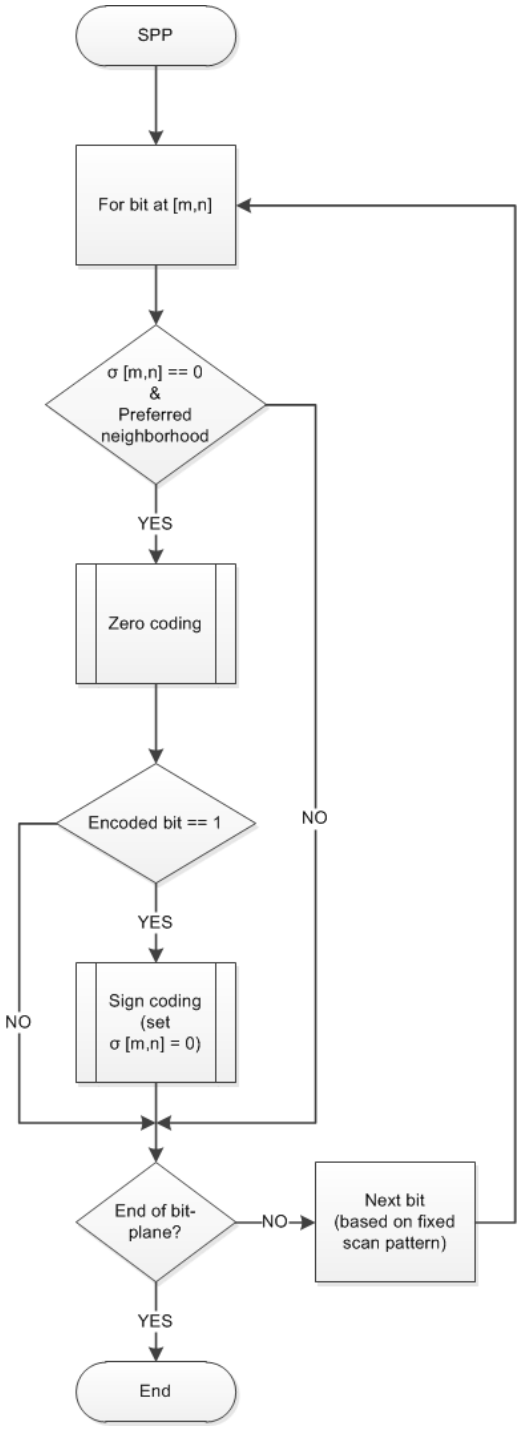

Figure 6: Significant Propagation Pass Flowchart

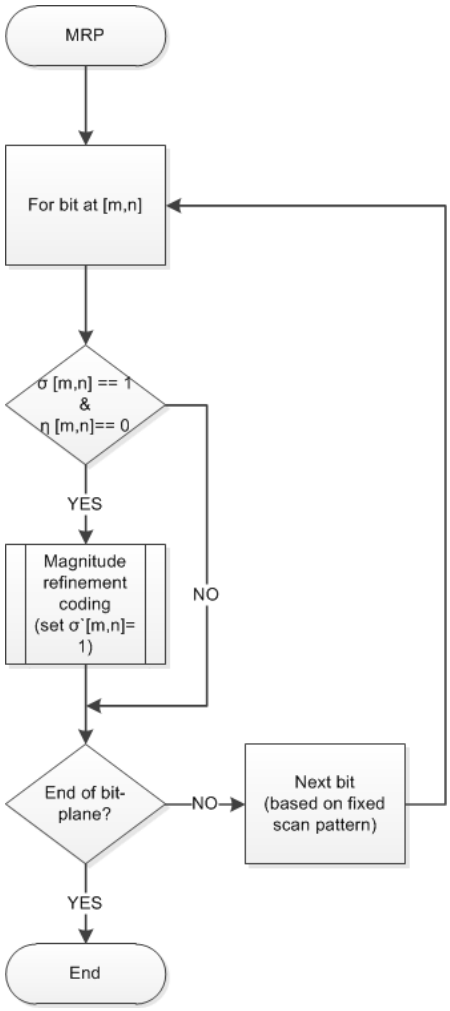

Figure 7: Magnitude Refinement Pass Flowchart

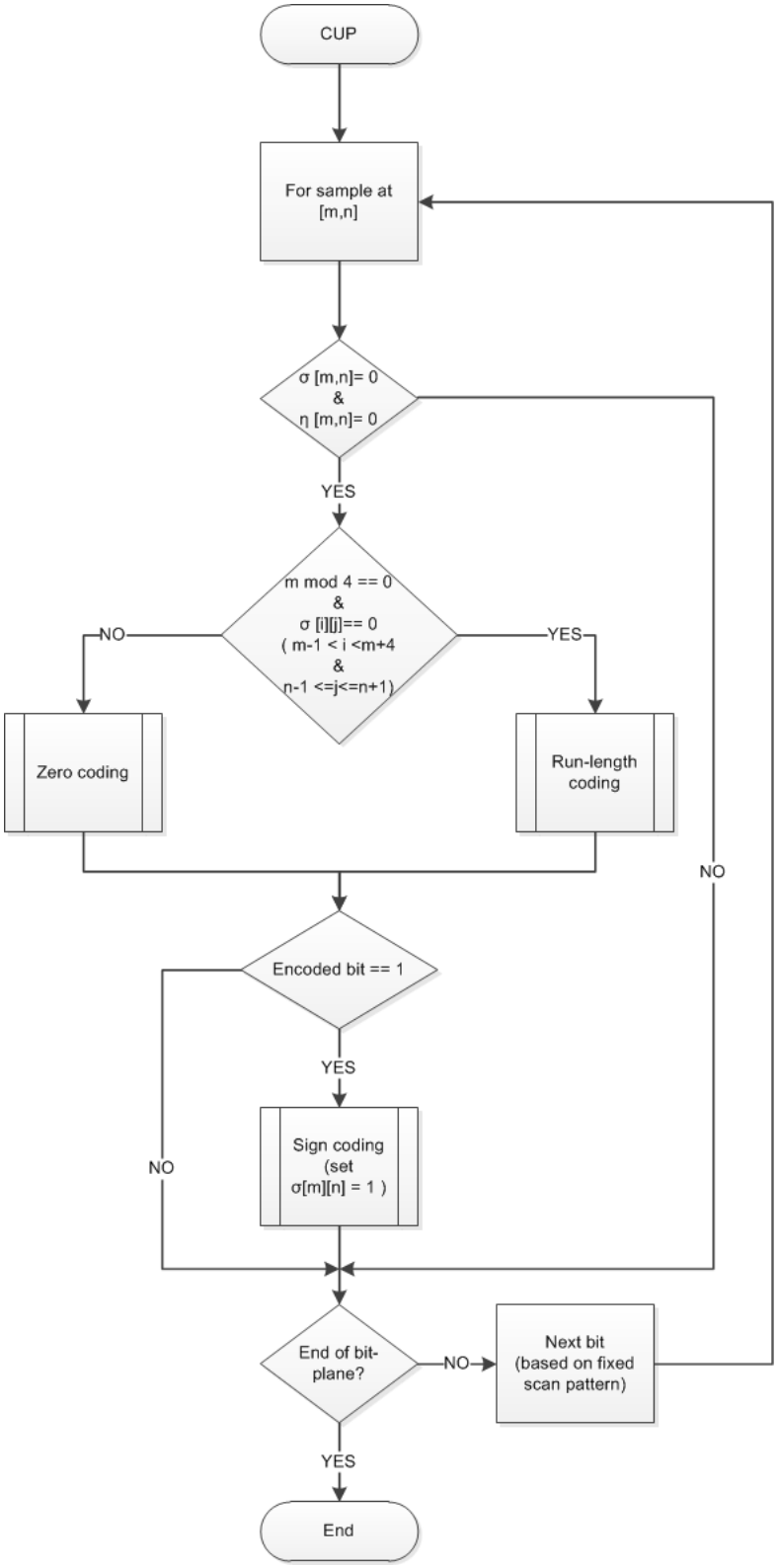

Figure 8: Cleanup Pass Flowchart

Cleanup Pass (CUP): CUP coding is applied to every bit-plane of the code-block. CUP can apply Sign Coding (SC) and one among Run-Length Coding (RLC) or Zero Coding (ZC) to the current scan element (m,n) based on the value of the state variables. If $\sigma[m,n]$ and $\eta[m,n]$ are equal to 0 then CUP applies the coding algorithms to the bit, otherwise it skips to next scan bit in the bit-plane. If $\sigma[m,n] == 0$ and $\eta[m,n] == 0$ then CUP checks to apply RLC or ZC, below three conditions should be satisfied to apply RLC, if any one of them is not satisfied then ZC is applied to current scan bit.

- m is multiple of four, including $m = 0$.
- $\sigma = 0$ for the four consecutive locations on the same column, starting from current scan position.
- $\sigma = 0$ for all the adjacent neighbors of the four consecutive bits in the column.

The number of coded bits vary depending on the type of coding applied to the scan bit. CUP after RLC or ZC coding, checks for application of SC. If $\upsilon^p[m,n]$ equals to 1 then SC is applied to the scan bit and $\sigma[m,n]$ is set to 1. CUP iterates this for all the bits in the bit-plane of the code-block, after completion of all the bits in the bit-plane, $\eta[m,n]$ is cleared to 0 for all n and m in the bit-plane before starting coding of the next bit-plane. Figure 8 shows the flow chart of CUP coding.

2.5. Binary Arithmetic Coding (BAC)

EBCOT algorithm uses Binary Arithmetic Coding (BAC) to encode/decode symbols generated from BPC. Context (CX) and decision (D) information generated from BPC is encoded by BAC to generate compressed code bytes. BAC used in Tier-1 coding is a special variant of QM-coder used in JPEG, it's a context adaptive BAC know as MQ-coder.

Arithmetic coding is a variable length encoding technique, it encodes a sequence of input symbols to generate specific code. In arithmetic coding the input symbols are encoded into an interval of real number, the encoded interval is confined to a fixed standard range associated with the algorithm. Length of the encoded interval is inversely proportional to the probability of occurrence of the symbol. More probable symbol produces smaller encoded interval than less probable symbol and hence reaching higher encoding efficiency.

Arithmetic coding is highly efficient for coding bi-level symbols as compared to other entropy encoding techniques. However arithmetic coding is computational exhaustive and is more prone to errors. Several coding techniques have been derived from arithmetic coding to overcome computation complexity, QM-coder is one such coding technique. QM-coder is an adaptive binary arithmetic coding algorithm used in JBIG2 (Joint Bi-level Image Processing Group) standard [9] for bi-level image compression. It is an incremental coding technique, where the encoder need not wait until all the symbols are received to generate encoded data and decoder need not wait until all the encoded data are received to start decoding.

The principle idea of QM-coder is to map input symbols to more probable symbol (MPS) and less probable symbol (LPS). For instance if the inputs to QM-coder are binary signals 1 and 0, bit 1 is mapped to MPS if most of the neighbors of the current bit coded are 1 and bit 0 will be mapped to LPS, similarly bit 0 is mapped to MPS if most of the neighbors of the current bit coded are 0 and bit 1 will be mapped to LPS. QM encoder decides if the next input bit belongs to MPS or LPS and compresses this information, similarly decoder determines if the decoded bit is MPS or LPS and generates the decoded symbols based on this information.

QM-coder assigns a portion of predetermined fixed interval to MPS and LPS symbols based on the probability estimates of the symbols, it uses probability estimation table for each of the symbols generated in coding process. MQ coder used in JPEG2000 is a variant of QM coder in terms of the encoding interval, probability estimation table and computation complexity of coder. In this thesis we concentrate on MQ-encoder algorithm and its implementation specifics, MQ-decoders algorithm is beyond the scope of this thesis.

2.5.1. MQ-encoder

Top level block diagram of MQ-encoder is as shown in Figure 9. Inputs to MQ-encoder are Context (CX) and Decision (D) bit generated from BPC block. MQ-encoder generates compressed data as output. MQ encoder uses several look-up-tables (LUT) for the computation of compressed data, LUTs used by MQ-encoder are probability estimate table (Qe-table), this holds the probability estimates for all possible states reached by the encoder, Index-context table (I(CX)) and More Probable Symbol context table (MPS(CX)) are used along with the Qe-table to keep track of state and the index of Qe-table for different input context value.

I(CX) is used to track the index of Qe-table, initial value of I(CX) is provided by the standard [1], it is as shown in the Table 7. MPS(CX) LUT is initialized to all zeros, it provides the sense of more probable symbol (1 or 0) of CX. NMPS(I(CX)) and NLPS(I(CX)) LUTs are used to identify next MPS/LPS index values respectively. SWITCH(I(CX)) indicates if the sense of MPS(CX) has to be inverted, next section discusses in detail about the condition under which the sense of MPS has to be inverted. LUTs of Qe, NMPS, NLPS and SWITCH with respect to index reference are as shown in Table 8.

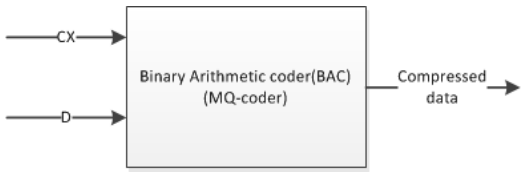

Figure 9: MQ Encoder top level block diagram[1]

MQ-encoder requires two 32-bit register for computation. Register A and register C, the structure of these are as shown in Table 9. Register A is the interval register and is initialized to 0x00008000. Register C is code word register and is initialized to 0x00000000. “a” of register A represents fractional bits. In register C “x” represents fractional bits, “s” represents space bits,

Table 8: BAC lookup table for Qe value and probabiliy estimation[1]

Index	Qe	NMPS	NLPS	SWITCH
0	0x5601	1	1	1
1	0x3401	2	6	0
2	0x1801	3	9	0
3	0x0AC1	4	12	0
4	0x0521	5	29	0
5	0x0221	38	33	0
6	0x5601	7	6	1
7	0x5401	8	14	0
8	0x4801	9	14	0
9	0x3801	10	14	0
10	0x3001	11	17	0
11	0x2401	12	18	0
12	0x1C01	13	20	0
13	0x1601	29	21	0
14	0x5601	15	14	1
15	0x5401	16	14	0
16	0x5101	17	15	0
17	0x4801	18	16	0
18	0x3801	19	17	0
19	0x3401	20	18	0
20	0x3001	21	19	0
21	0x2801	22	19	0
22	0x2401	23	20	0
23	0x2201	24	21	0
24	0x1C01	25	22	0
25	0x1801	26	23	0
26	0x1601	27	24	0
27	0x1401	28	25	0
28	0x1201	29	26	0
29	0x1101	30	27	0
30	0x0AC1	31	28	0
31	0x09C1	32	29	0
32	0x08A1	33	30	0
33	0x0521	34	31	0
34	0x0441	35	32	0
35	0x02A1	36	33	0
36	0x0221	37	34	0
37	0x0141	38	35	0
38	0x0111	39	36	0
39	0x0085	40	37	0
40	0x0049	41	38	0
41	0x0025	42	39	0
42	0x0015	43	40	0
43	0x0009	44	41	0
44	0x0005	45	42	0
45	0x0001	45	43	0
46	0x5601	46	46	0

it provides constraints on carryover, “b” represents bits of ByteOut and “c” represents the carry bit.

MQ-encoder subdivides the interval into two sub-interval for each context (CX) and decision (D) pair input. The interval register content is modified accordingly to point to the lower bound of the probability sub-interval of the symbol currently encoded. Partitioning of the sub-interval is done in such a way that the sub-interval of MPS is positioned above the sub-interval of LPS. Hence its necessary for the encoding algorithm to recognize the input symbols as MPS or LPS and keep track of the sense of MPS/LPS along with the interval associated with that. Since the encoding process works on the basis of accumulating several symbol information before finally generating compressed data byte, more probable symbols are encoded with much less than one bit per symbol.

MQ-decoder algorithm performs the inverse operation on the compressed data to generate the encoded symbols. MQ-decoder primarily determines the sub-interval pointed to by the compressed data, it also operates recursively for determination of context and decision information.

The interval considered for MQ-coder is 0.75, MQ-coder uses fixed precision integer arithmetic. Integer representation of the fractional value 0.75 is equivalent to 0x8000 in hexadecimal. Hence the A register is initialized to 0x8000 and C register is initialized to 0x0000. The algorithm is designed to keep the interval in the range of $0.75 \leq a \leq 1.5$, whenever the interval value falls below 0.75, interval correction is made by doubling it. The content of C register is also doubled whenever the A register content is doubled. Compressed data byte is generated from the higher order bits of C register, bit position of C register are as shown in Table 9. Carry over to already generated compressed byte is avoided by the algorithm.

Calculations of the sub-intervals are as shown in equations 4 to 7, a simple arithmetic approximation is used for interval sub-division. For the purpose of illustration if the interval is A and probability estimation of LPS/MPS is Qe then the sub-interval for MPS is given by equation 4 and sub-interval for LPS is given by 5. Since the value of A is close to unity equations 4 and 5 are approximated to equation 6 and 7 respectively. At each sub-interval calculation step, the value of A is checked to determine if renormalization of interval is necessary, in such a case the value of A register and C registers are renormalization.

$$MPS\ \ sub-interval = A-(Qe*A) \tag{4}$$

$$LPS\ \ sub-interval = (Qe*A) \tag{5}$$

$$MPS\ \ sub-interval = A-Qe \tag{6}$$

$$LPS\ \ sub-interval = Qe \tag{7}$$

The interval subdivision process can run into a scenario where the value of MPS is less than the value of LPS, in such a case the intervals of MPS and LPS are exchanged, such a condition can occur when renormalization is necessary. During renormalization process the value of next probability estimate is determined for the current context value being encoded. The value of next probability estimation is determined from Table 8

The flow chart of MQ-encoder is as shown in Figure 10. In the ”initialization” step the computational register A and C are set to their initial values, the counter and pointer variables used by the algorithm are set to default value and the LUTs are reset to default value. In the ”Read CX, D” step the Context (CX) and Decision (D) input symbols are read from BPC stage. The read symbols are used in the decision ”D==MPS(CX)” step to compare the D value with the sense of MPS associated with the current context values. If the D value is same as the sense of MPS associated with current CX then MPS coding is performed otherwise LPS coding is performed. After MPS or LPS coding operation, the next CX and D symbol pair is read for encoding, this cycle is repeated until all the symbols are encoded.

In MPS coding step ”CodeMPS” Qe value associated with current CX is read from the LUT Table 8. The content of C register is added with Qe value and the interval register A is modified correspondingly. Depending on the value of interval register A, the sense of MPS/LPS may be exchanged and the content of A and C registers may be renormalized. The values of next index for the current CX is determined by NMPS(I(CX)) value from the LUT Table 8.

In LPS coding step ”CodeLPS” Qe value associated with current CX is read from the LUT Table 8. The content of A and C registers are modified correspondingly. If the value of SWITCH for the index of current CX is “1” then the sense of MPS is changed. The renormalization procedure is always called in LPS coding and is called in MPS coding procedure if the value of A register is less than 0x8000. The next index for the current CX is given by NLPS(I(CX)) value in the LUT Table 8.

In renormalization procedure the content of A register is shifted left until its value is equal to or greater than 0x8000, during this shifting process the context of C register is also shifted same number of time as it is done with A register. The renormalization procedure may invoke the function to output the compressed byte from the C register, it also avoids propagation of carry into already outputted compressed byte. ”FLUSH register” step is reached after all the symbols are encoded, in this step the register C is stuffed with as many ”1” bits as possible before outputting the final byte of the compressed stream. Pseudo code[2] for ”Initialization”, ”CodeMPS”, ”CodeLPS” and ”Renormilization” is as shown under Appendix C.

Table 9: BAC Encoder Register Structures

32-Bit Register	MSB LSB
C (Code Register)	0000 cbbb bbbb bsss xxxx xxxx xxxx xxxx
A (Current Interval Value)	0000 0000 0000 0000 aaaa aaaa aaaa aaaa

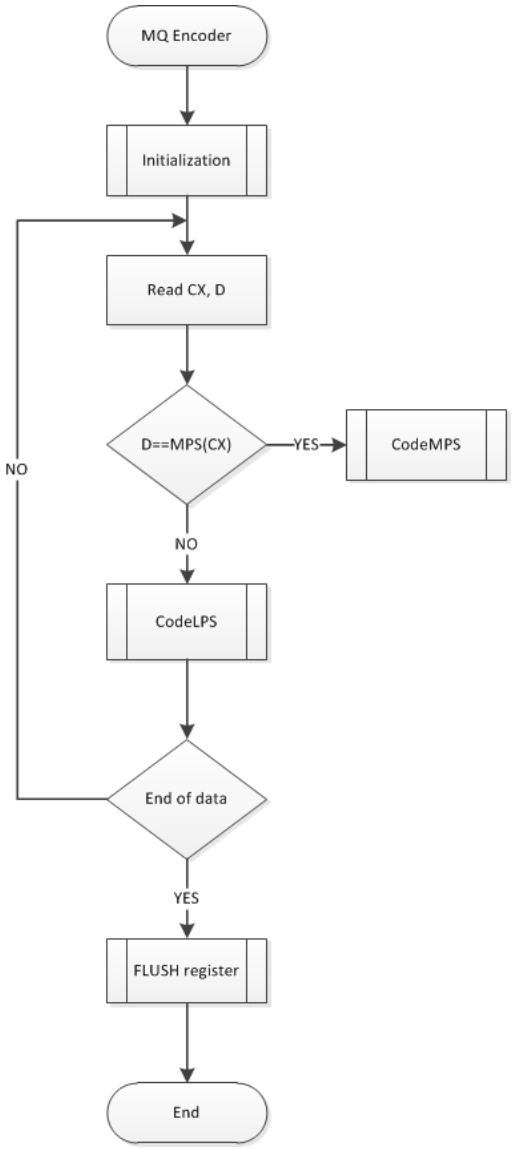

Figure 10: MQ-Encoder flow chart[2]

3. EBCOT ENCODER IN SOFTWARE

The EBCOT hardware architecture design in this thesis is verified against a reference model from OpenJPEG [10]. OpenJPEG is an open-source library for JPEG2000 encoder and decoder. OpenJPEG is implemented in C language. OpenJPEG apart from JPEG2000 codec features, it also supports Motion JPEG2000 (MJ2) features. Some of the features supported by OpenJPEG software are:

- Lossless and lossy compression with different compression ratios.
- Different input image formats like ”pgx”, ”pnm”, ”pgm”, ”ppm”, ”bmp”, ”tif”, ”raw”, ”tga”, ”png”
- Ouput image formats like J2k and JP2.
- Supports image size up to $(2^{32}-1) \times (2^{32}-1)$ and image components up to 2^{14}
- Code-block size in powers of 2 upto maximum image size.
- Supports tile sizes in powers of 2 upto maximum image size.
- ROI processing support.
- Use of irreversible DWT 9-7
- Cinema 2k and 4k resoultion modes, supports only 24 or 48 frames per second video formats.
- Markers in main header and tile header (SOC SIZ COD QCD COM).
- Progression order (LRCP, RLCP, RPCL, PCRL, CPRL).
- SOP and EPH markers in the code streams.
- Coding of multiple images at a time(10 images at one go, sequentially)

3.1. Architecture

Top level software architecture of OpenJPEG is as shown in Figure 11. In Initialization step, event call-back functions are defined, encoding parameters are set to default values and the indexes used during encoding/decoding are initialized. Command line parser parses the user commands for input image with coding type and coding parameters. Read input image block reads the image to be processed from the specified directory. OpenJPEG can process upto ten images with one command, these images are encoded/decoded sequentially one after the other. Coding type parameter is used to select encoder or decoder function for processing the input image in next step. The generated output stream is written into specified output file in write to outfile block.

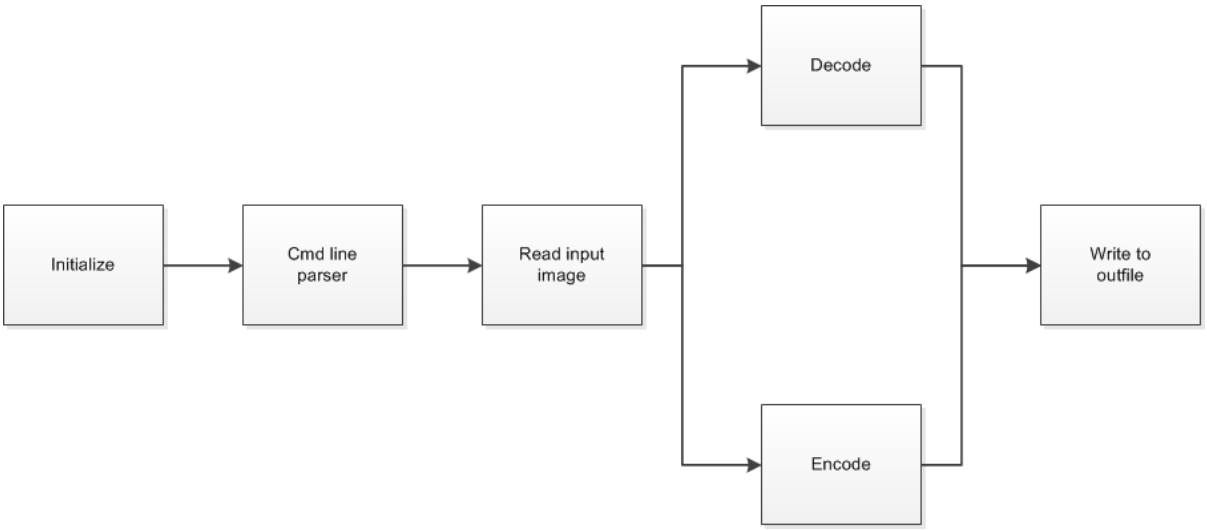

Figure 11: Software Architecture block diagram

We focus our discussion on Encoder block in this thesis. Figure 12 shows the block diagram of Encoder. Write markers block writes several markers such as Start of Code stream (SOC marker), Image size and tile size (Size marker), Coding style (COD marker) and quantization default (QCD marker) to code stream. Create tile encoder creates the data structure for tile encoder and returns the handler for further use. Memory required for processing of tile is allocated by allocate memory block, memory required for one tile is allocated and is reused for rest of the tiles.

Start of tile and start of data markers are written in write SOT and SOD markers blocks. Encoding of the data is done in Tile encode block (TCD), TCD encode block encodes the data from input image into JPEG2000 code stream and writes the encoded code stream into buffer. TCD encoding is repeated for all the tiles and the code stream is written into the outfile. Memory allocated for TCD block is freed after completion of encoding operation and is done in Free memory block. EOC marker block writes End of Code stream marker into the code stream.

Figure 12: Encoder block diagram

TCD encoder is the block resposible for implementation of coding phases of EBCOT. Block diagram of TCD encoder is as shown in Figure 13. Input image is extracted and processed in terms of tiles, this is done by extract tile data block. Reversible multi-component transform is

applied to the extracted data in MCT encoder. Discrete wavelet transform is applied to MCT encoded data in DWT encode.

Tier 1 encoding is done on the DWT encoded data to produce Context (CX) and Decision value bit (D). Rate allocation block is responsible for construciton of bit-stream from encoded code-blocks for a give bit-rate with least distortion. Tier-2 encoding is primarily responsible for efficiently representing layer and block summary information for each code-block. Tag Tree data structure is used in Tier-2 encoding for efficient code stream representation. In clean up step the memory allocated for coding is cleared and the parameters are initialized to default values.

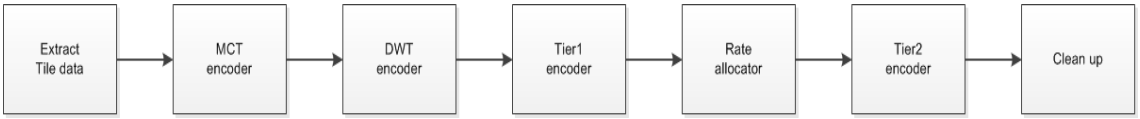

Figure 13: TCD Encoder block diagram

3.2. Profiling

This subsection deals with profiling of JPEG2000 encoder implemented in software. The aim of profiling is to analyze different processing blocks in JPEG2000 encoder in terms of the relative processing time, processing dependencies with respect to different code-block sizes, image sizes, color components, image properties etc. Several images with different characteristics were used for profiling. Characteristics of images used for profiling are shown in Table 10. Images used for profiling are shown under Appendix B.

Table 10: Image characteristics table

Image	Dimension (pixles)	Size (bytes)	Bits per component	No of components
Bretagne1.ppm	640x480	921638	8	3
Bretagne2.ppm	2592 x 1944	15116584	8	3
Cevennes1.ppm	2592 x 1944	15116584	8	3
bike.pgm	2048x2560	20263700	8	3
target.pgm	512x512	950471	8	1
lena.ppm	512x512	2883036	8	3
lena.pgm	512x512	983503	8	1
cwheel.ppm	800x600	4599557	8	3
cwheel.pgm	800x600	1686904	8	1
frymire.ppm	1118x1105	11781663	8	3
frymire.pgm	1118x1105	4103510	8	1

Each of the images in Table 10 were encoded and profiling statistics were collected. Encoding of these images were done with three different code-block size of 64×64, 32×32 and 16×16. The tile size for all the images was configured to 1. J2K Output image format was configured for all image encoding. Three different compression ratio of 100, 20 and 2 were used for encoding and reversible DWT 5-3 was used for encoding of all images.

3.3. Profiling results

Profiling of JPEG2000 software implementation is done with images of different compression parameters, each of the profiling experiments and their results are discussed in detail in the below subsections.

3.3.1. Compressed image size

This subsection discusses the dependencies of compressed image size on different compression parameters. Compression algorithm is ran on the images in Table 10 for three different iterations with code-block size of 64×64, 32×32 and 16×16 respectively. The size of the compressed images is collected, alone with the size of original uncompressed image.

Compressed image size along with original image size in bytes is plotted against the images. The plot is as shown in Figure 14. The plot shows that increase in code-block size decreases the size of the compressed image since the number of code-blocks formed for the images decrease and hence the size of encoded bit stream. The size of the compressed image also depends on the characteristics of the image; in general synthetic images are compressed better as compared to nature images.

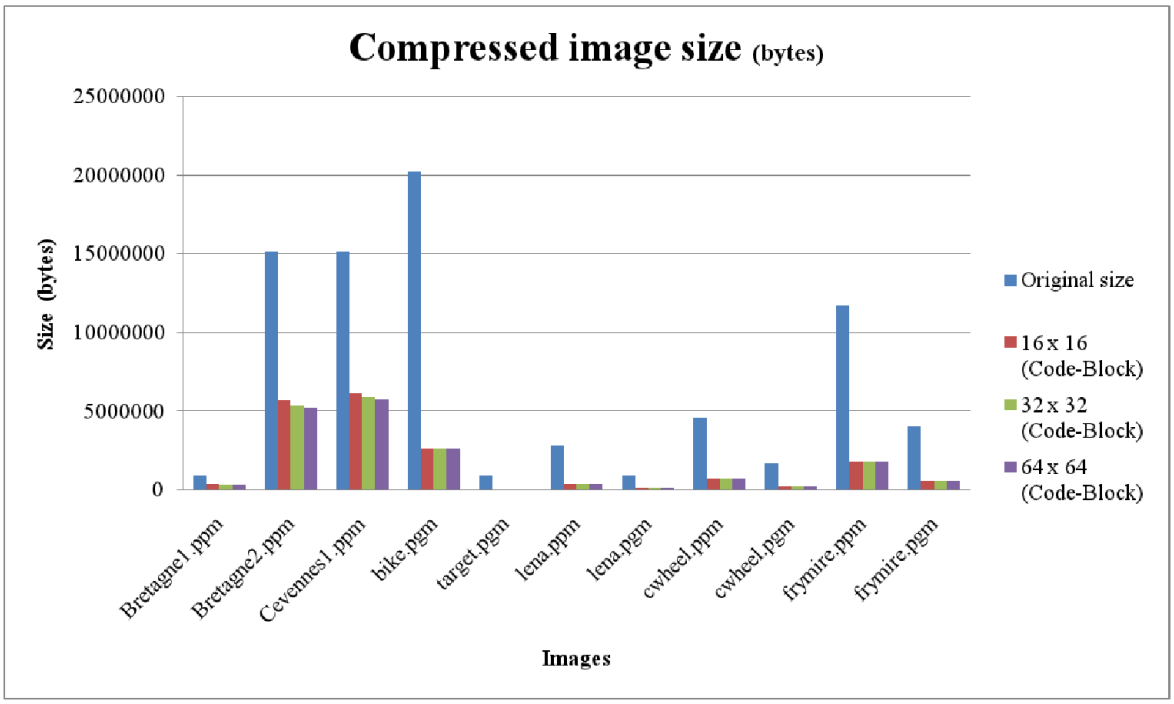

Figure 14: Compressed image Size

3.3.2. Compression time

Dependency of compression time on the size of the code-blocks is analyzed in this subsection. The images in Table 10 are compressed with code-block sizes 64×64, 32×32 and 16×16 and the time required to compress in each iteration are noted. Figure 15 shows the plot of compression time against images.

Plot shows that the compression time increases with decrease in code-block size. When the code-block size is decreased the number of code-blocks formed per image increases and processing blocks like Tier-1 and rate allocation consumes significant amount of compression time (discussed in detail in subsection 3.3.3). For the set of images in Table 10, on an average it is observed that compression time with code-block size 32×32 decreases by 60% as compared to compression time with code-block size 16×16 and compression time with code-block size 64×64 decreases by 75% as compared to compression time with code-block size 16×16.

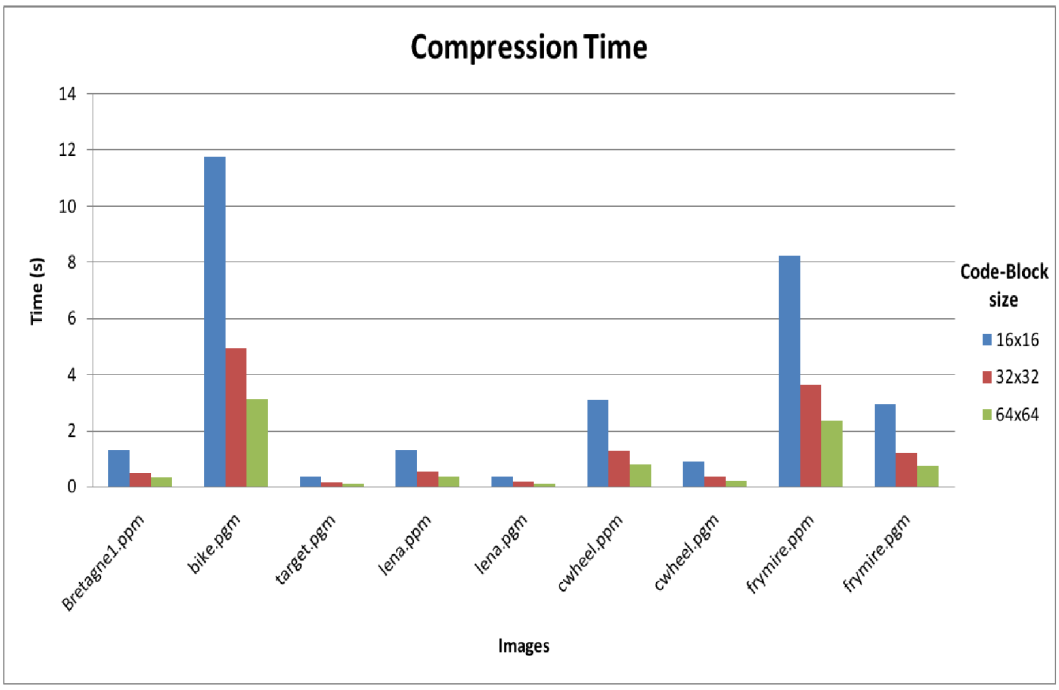

Figure 15: Compression time

3.3.3. Block processing time

This subsection deals with analysis of processing time consumed by individual blocks of JPEG2000 encoder and the effect of code-block size on processing time of these block. Figure 16 shows the plot of processing time against images, the processing time consumed by each block for a particular image is plotted.

Plot shows that for code-block size 64×64 processing time consumed by Tier-1 encoder is higher than the rest of the processing blocks irrespective on the image characteritics. For the set of images in Table 10, on a average it is observed that for code-block size 64×64 Tier-1 consumes 65% of total compression time required for each image. Figure 17 and 18 shows block prcessing time plot for clode-block size 32×32 and 16×16 respectively. It is observed that rate allocation block's processing time increases significantly however Tier-1 processing time marginally increases with increase in code-block size.

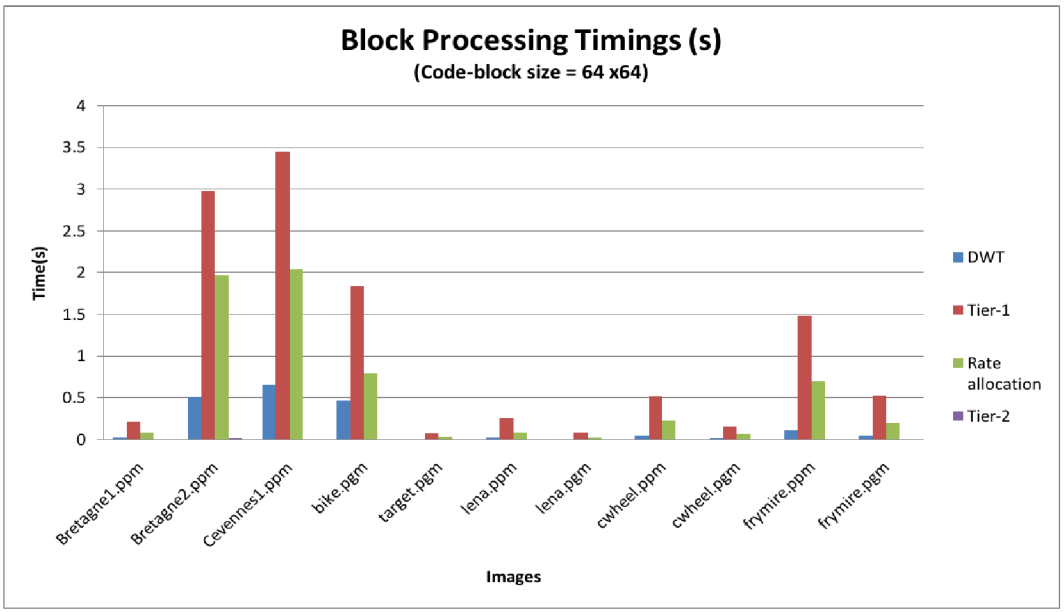

Figure 16: Block processing time (code-block 64×64)

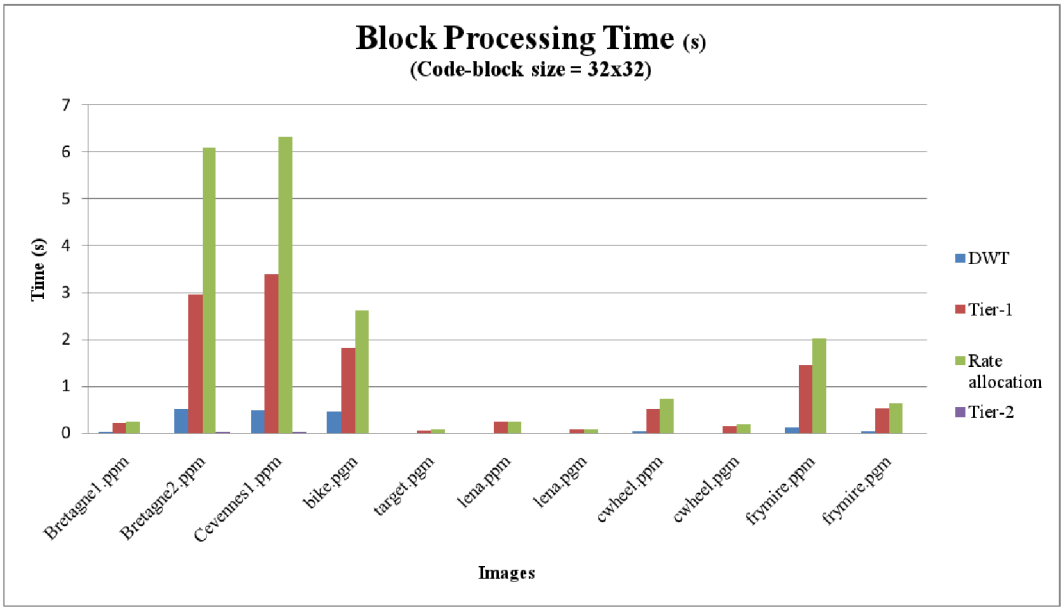

Figure 17: Block processing time (code-block 32×32)

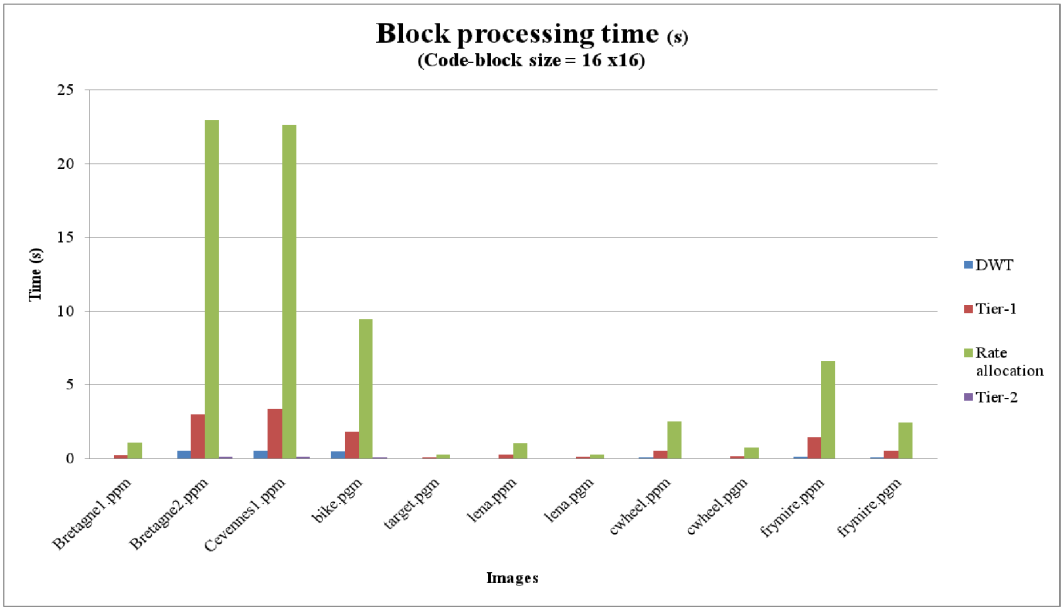

Figure 18: Block processing time (code-block 16×16)

3.3.4. Conclusion

Profilng results have shown that Tier-1 encoding block consumes significant amount of compression time. Section 2 gives detailed explanation about the resources requirements for implementation of EBCOT in hardware. EBCOT encoder consumes significant amount of computational resource in JPEG2000. The aim of this thesis is to propose parallel architectural design to optimize resource requirement and processing time of EBCOT encoder.

4. EBCOT ENCODER HARDWARE IMPLEMENTATION

This Section discusses the hardware implementation of EBCOT Tier-1 encoder. it discusses the top level architecture of major processing components of Tier-1 encoder in detail. Subsection 4.1.1 to 4.1.6 discusses in detail the architecture and hardware structure of individual components of BPC and BAC. Subsection 4.2 discusses the verification methodology adapted for verification of individual components of BPC and BAC. Simulation and synthesis results of BPC and BAC design are discussed under subsection 4.3.

4.1. Architecture

This subsection discusses the hardware architecture and implementation of BPC and BAC in detail. Architecture is designed keeping in mind the feasibility of implementation on a FPGA (Field Programmable Gate Array), it also takes advantage of the flexibility offered by Xilinx FPGA in terms of block RAM and LUTs available on FPGA. A VHDL package of subtypes with meaningful names is created and these subtypes are used in the design files for better readability, ease of use and to reduce coding complexity. The VHDL package used in the design is given in appendix A.

4.1.1. BPC TOP

Hardware is designed and implemented in a hierarchical fashion. Basic encoding algorithms like sign coding (SC), zero coding (ZC), run length coding (RLC) and magnitude refinement coding (MRC) are at the leaf level of the hierarchy. Instances of these leaf level blocks are used in SPP, MRP and CUP coding passes which forms the next hierarchy. Instances of coding passes and block RAMs along with control, input and output logic are used in top level and it forms the highest level of hierarchy. Hierarchy structure of BPC is as shown in Figure 19. The top level architecture of BPC is as shown in Figure 20. Processing blocks depicted in Figure 20 are explained in detail in the subsections below.

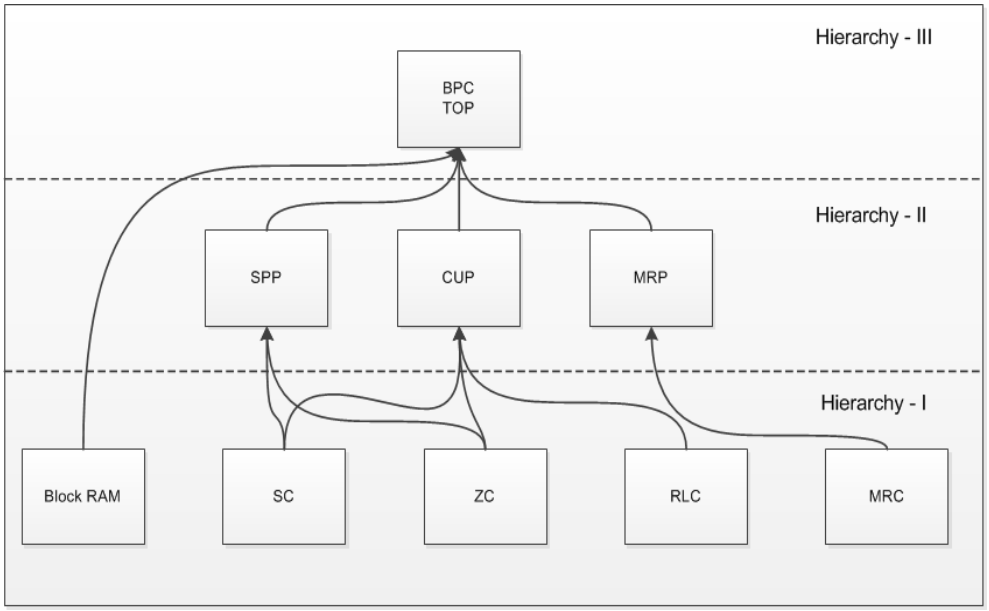

Figure 19: Hierarchy of BPC design

Input Block

Input block reads the quantized wavelet coefficients for entropy encoding. Input block reads the coefficients associated with the entire code-block in single iteration. The read coefficients are decomposed into bit-planes and stored in the block RAM associated with the bit-plane. BPC algorithm does processing on each of the bit-planes associated with one code-block. Input block is described along with a flow chart under subsection 4.1.2

State Variable ($\sigma, \sigma^{'}$, η, υ, χ) Block RAM

Block RAMs(Random Access Memory) in FGPA are used to store the values associated with the state variables used in BPC entropy encoding. Each of these block RAMs are accessed by several processing block in BPC. The blocks accessing state variable's block RAMs are Input block, SPP, MRP and CUP. BPC architecture is designed with the assumption that the maximum code-block size processed by the system is 32×32 pixels. However the code-block's size variables are parameterized in the design file, hence the system can be easily enhanced to higher code-block size by replacing the block RAMs with relevant dimension and assigning corresponding values to code-block's size parameters.

Memory Arbiter

Block RAMs used by state variable are accessed by several processing blocks. Memory Arbiter arbitrates among the port requesting access to block RAMs. Each input port requesting access is associated with a control signal, on assertion of this control signal the request to access port is granted to the requesting port. Detailed discussion about memory arbiter is done under subsection 4.1.4.

SPP, MRP and CUP

SPP, MRP and CUP are the coding passes of BPC algorithm. Each of these coding passes performs encoding operation on bit-planes of the code-block to generate encoded symbols. Section 2.4 covers the detailed description of these coding passes. Coding passes are controlled by the control block; control block controls the sequence of encoding of each of the bit-planes and also the coding passes to be enabled for each of the bit-plane. The encoded symbols generated by these coding passes are passed on to the output block.

Output Block

The encoded symbols generated by the coding passes are collected by the output block and are written into the output memory. These encoded symbols are later accessed by the BAC block for arithmetic encoding.

Control Block

Control block generates all the control signals necessary for encoding process of BPC. It generates initialization signals for different blocks at different stages of encoding process. Control block initiates reading of quantized wavelet coefficients through input block. It controls the sequences of coding passes during encoding, it also initiates writing out of generated output symbols through output block. Details of control block along with the flow chart is explained under subsection 4.1.3.

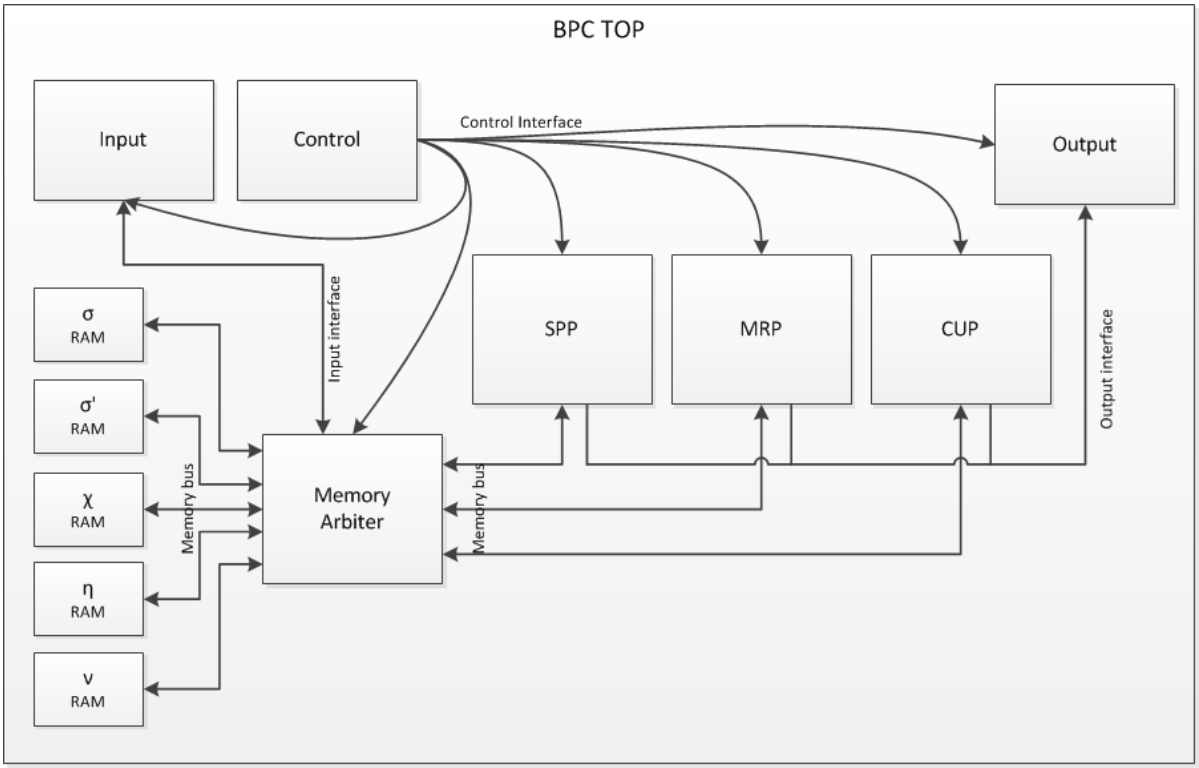

Figure 20: Top Level Architecture of BPC

4.1.2. Input Block

Flow chart of input block is as shown in Figure 23. Input block is enabled by a start signal from control block. Input block waits in the initial state until the start signal is asserted. After the start signal is asserted input block initializes the pointers and counters used in the state machine. It then reads the quantized wavelet coefficients from the memory.

Memory is read through a memory bus protocol, protocol is described in detail under subsection 4.1.4. The read coefficients are processed and temporary registers of sign array (χ) and bit-plane (υ^p) are updated. The cycle is repeated until all the coefficients of a code-block are read. After the code-block is read completely the content of χ and υ^p block RAMs are updated with the temporary χ and υ^p register contents.

4.1.3. Control Block

Flow chart of control block of BPC is as shown in Figure 24. Control block enables the input block to read the input coefficients and update χ and υ^p RAMs. It then initiate reading of state variables and bit-plane necessary for BPC encoding from the block RAMs. If the bit-plane is MSB bit-plane then only CUP encoding is done otherwise SPP, MRP and CUP encoding is done in sequence respectively. This cycle is repeated until all the bit-planes are encoded. Once encoding of all the bit-planes is completed, control blcok initiates writing of encoded symbols into the symbol memory. This encoded symbol memory is accessed by BAC to generate compressed bit-stream.

4.1.4. Memory Arbiter

Memory arbiter in BPC arbitrates the requests for the state variable's block RAMs. State variable block RAM is accessed by severable blocks of BPC at different processing instance. Memory arbiter has special memory bus interface [3] to block RAMs, read/write to the block RAM is through memory bus. Memory arbiter functions like a multiplexer; it establishes the connection between the block RAM memory interface and processing block's memory interface when the control signal associated with the processing block is asserted. These control signals are asserted by the control block, it decides which processing block has to be enabled at what processing instance. Memory arbiter is designed with the assumption that no two processing block will request for access to block RAM at the same time.

Figure 21: Single-port RAM

Block diagram of block RAM used in BPC is as shown in Figure 21. It is a single-port RAM with 32 memory locations of 32 bits each and is configured in read first mode. An example waveforms of memory bus protocol configured in read first mode is as shown in Figure 22. This waveform is an example used for explanation of the protocol, width of the signal bus in the Figure 22 does not correspond to the actual bus width used in BPC.

When the enable (ENA) signal is de-asserted, read/write to RAM is disabled. When ENA is asserted and write enable (WEA) is low, RAM is in read mode, it reads from the location addressed by address bus (ADDRA) and the read data is available on output data bus (DOUTA). When ENA is high and WEA is also high, RAM is in write mode and the data on data input bus (DINA) is written into the RAM at the location addressed by the ADDRA bus. Read first configuration mode of block RAM means that the old value of RAM at address X is read out on to DOUTA bus in case of write access to address X of block RAM. The read/write latency of the block RAM is one clock cycle.

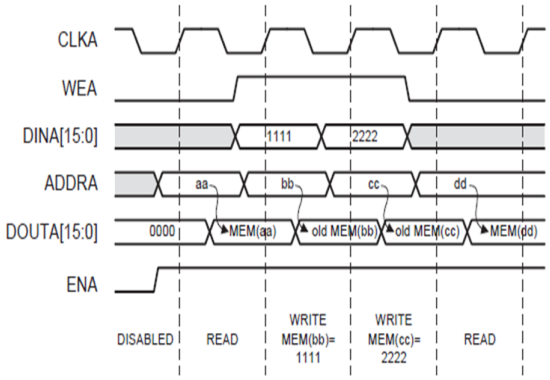

Figure 22: Read first mode single-port RAM [3]

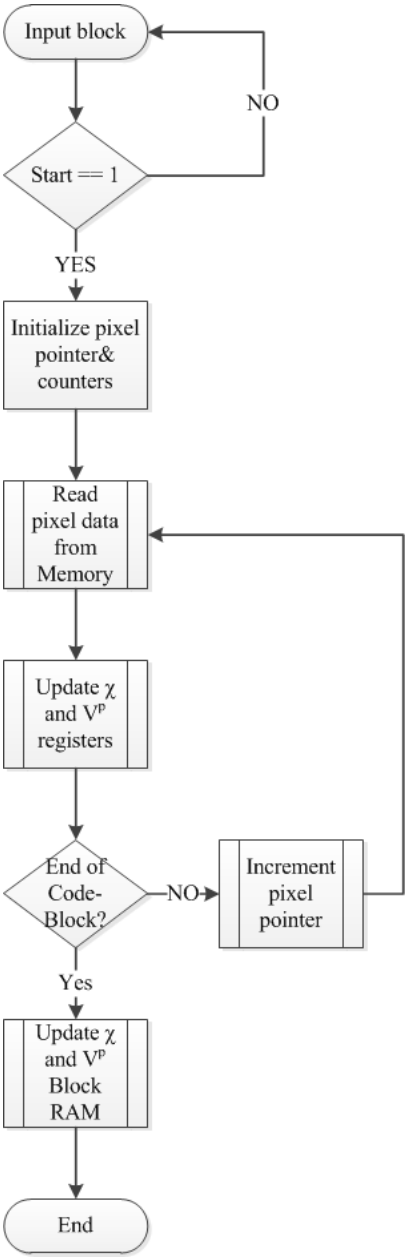

Figure 23: Input block of BPC

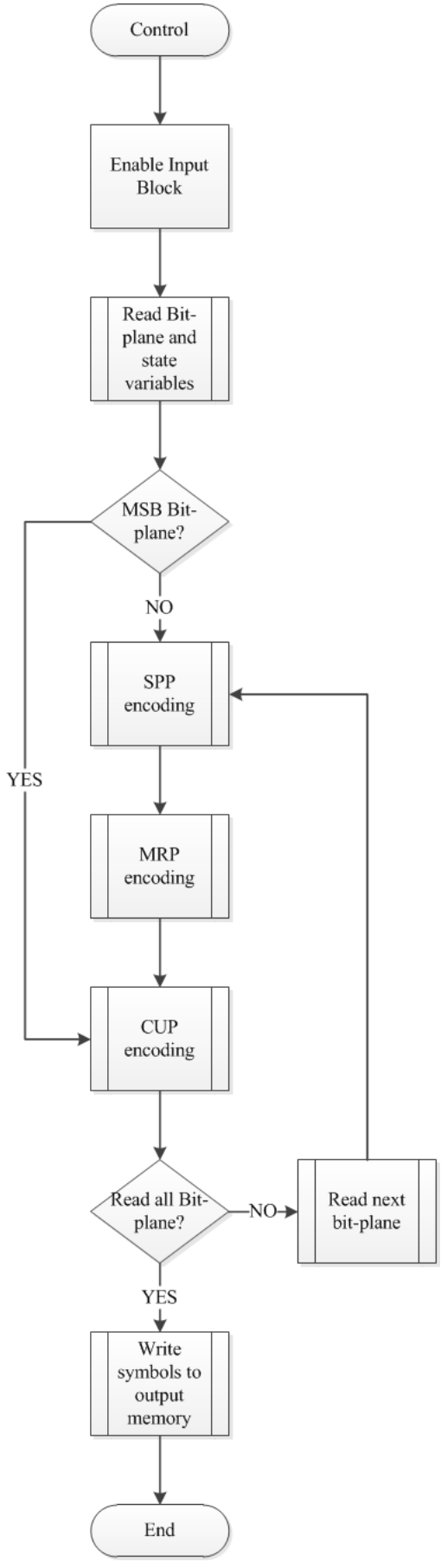

Figure 24: Control block of BPC

4.1.5. SPP, MRP and CUP

The hardware structure of SPP, MRP and CUP is as shown in the Figure 25, 26, and 27 respectively. Coding passes along with the flow chart are described in detail under subsection 2.4. These hardware structures are not the result of synthesizing VHDL code of coding passes, these structure are conceptual and are used for better understanding of the coding pass hardware.

SPP and MRP coding passes are not applied to MSB bit-plane; this can be observed from the Figure 25 and 26 respectively. The first multiplexer (MUX) bypasses the whole processing components if the bit-plane is MSB bit-plane. CUP coding pass is applied to all the bit-planes of the code-block. SPP coding pass applies zero coding (ZC) and sign coding (SC) encoding algorithms on the bit-planes, CUP applies run length coding (RLC), sign coding (SC) and zero coding (ZC) on bit-planes and MRP applies magnitude refinement coding (MRC) on bit-planes. The conditions for application of these encoding algorithms in each coding passes are explained in detail in subsection 2.4.

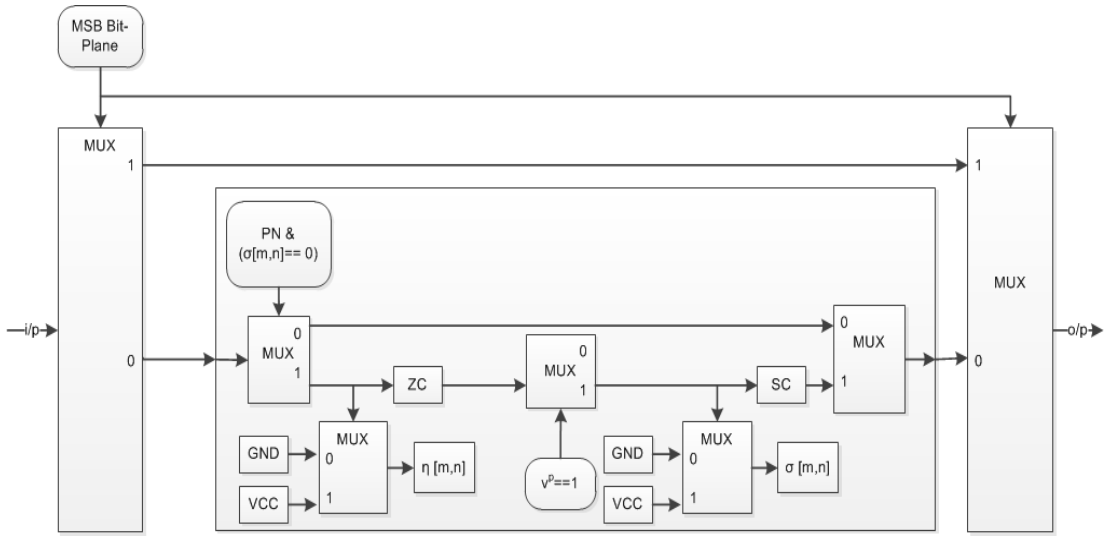

Figure 25: SPP Hardware structure

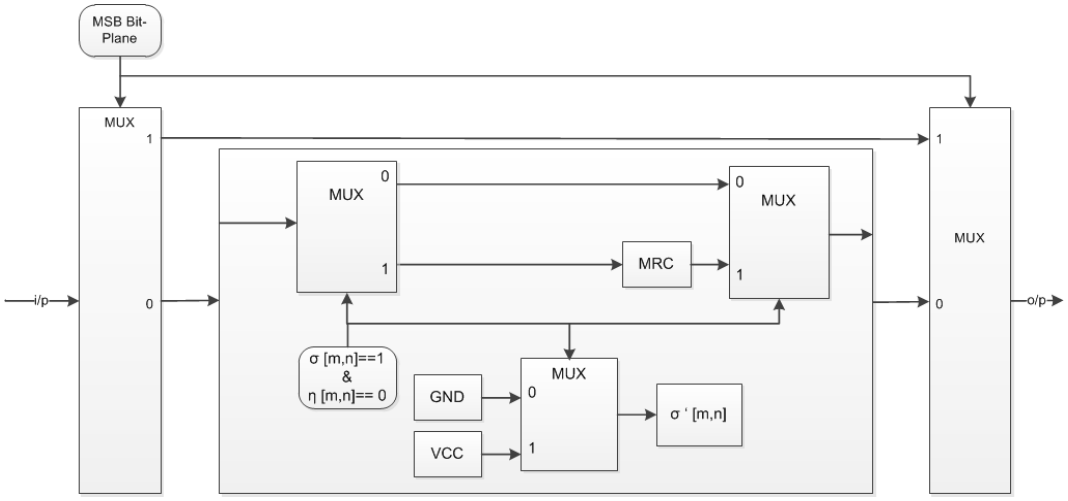

Figure 26: MRP Hardware structure

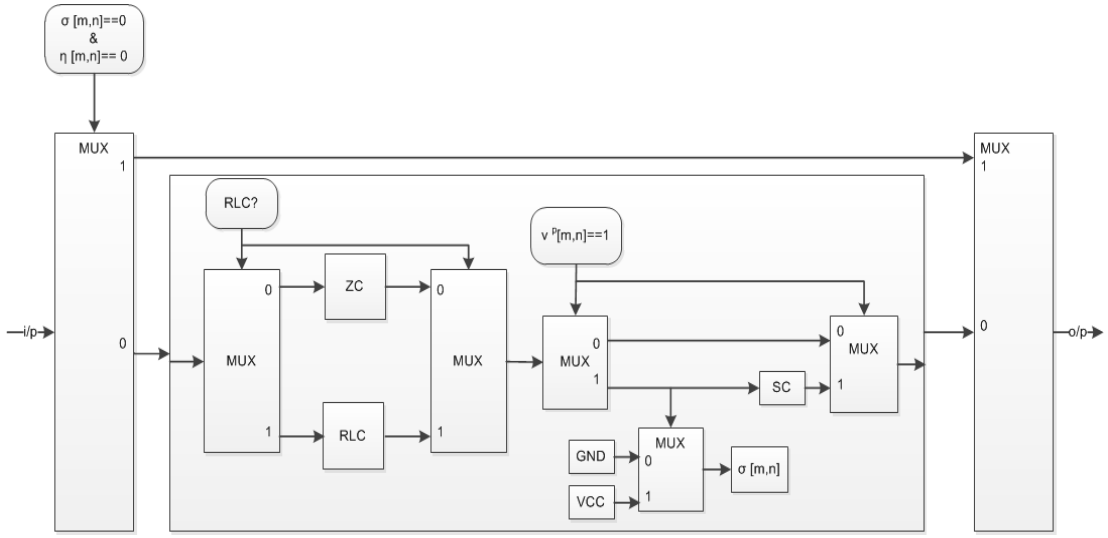

Figure 27: CUP Hardware structure

4.1.6. BAC TOP

The architectural block diagram of BAC is as shown in Figure 28. BAC is also designed and implemented in hierarchical fashion. BAC has only two level of hierarchy. The leaf level blocks are the LUTs used in the design. Input, computation and output blocks uses instances of LUTs, these form the highest level of hierarchy in BAC. BAC algorithm along with the flow chart is described in detail under section 2.5.

All LUTs used in the design has read interface. Index and MPS LUTs have both read and write interface since the value of these LUT table can be updated by MPS and LPS computational blocks. Input block initializes the counters, register, LUTs and pointers used in the algorithm. MPS and LPS are the main computation block in BAC. The StrucBtyte block is responsible for generation of compressed byte depending on the computational results of MPS and LPS blocks. Flush block is responsible for generation of the last byte when all the input symbols are encoded. Control block generates control signal to sequence the operation of input, computation and output blocks.

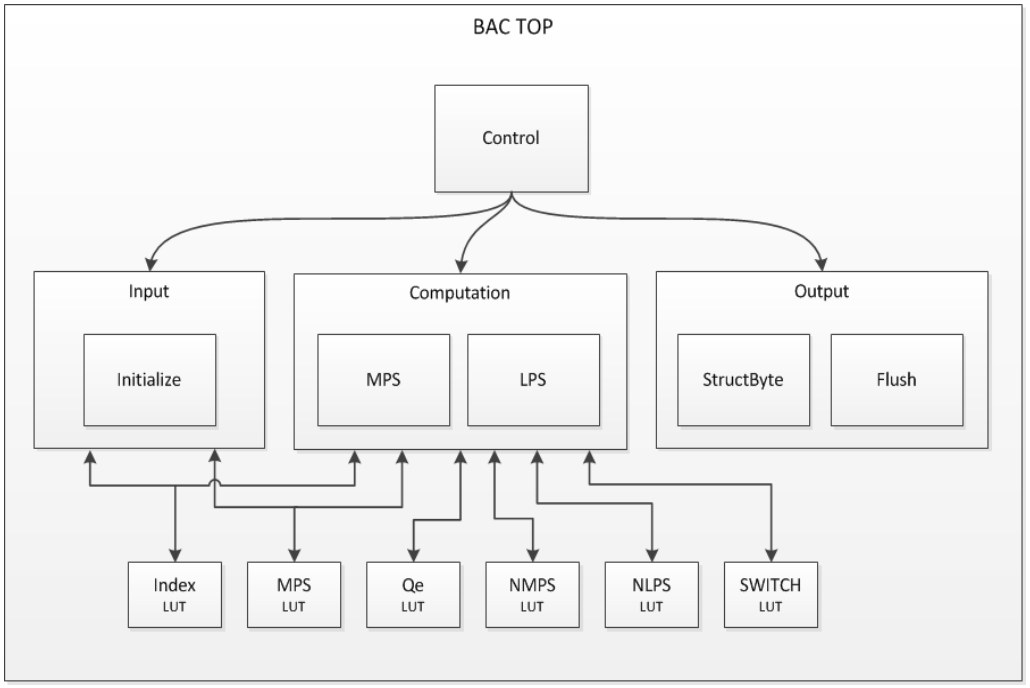

Figure 28: Top Level Architecture of BAC

4.2. Verification Methodology

Verification methodology followed in this thesis is shown as a block diagram in Figure 29. Verification is done stand alone for individual components at each hierarchy levels of BPC and BAC, also for top level BPC and BAC. Test bench used for verification is designed to be self checking test bench. Device under test (DUT) can be any of the components of BPC or BAC.

Test input represents all possible valid combinations of input vectors. Test inputs are created either manually or through a reference C-model. Test bench needs a reference output to check the correctness of the response generated by DUT for a particular test input vector. Test bench reads the test inputs from a file for verification of top level BPC and BAC components. Test bench reads the reference output from a file; reference output is created either manually or through reference C-model. Reference C-model used in this thesis is discussed in detail under section 3.

Test bench forces the test input into DUT, response generated by DUT is collected by checker and checker also reads the reference output for the particular input. Checker compares the response from DUT with reference output and updates the test result. Test result can be pass or fail; it also logs the information about input vector, generated response and reference output at the point of test failure. Xilinx ISE 13.2 [11] simulation tool is used for verification.

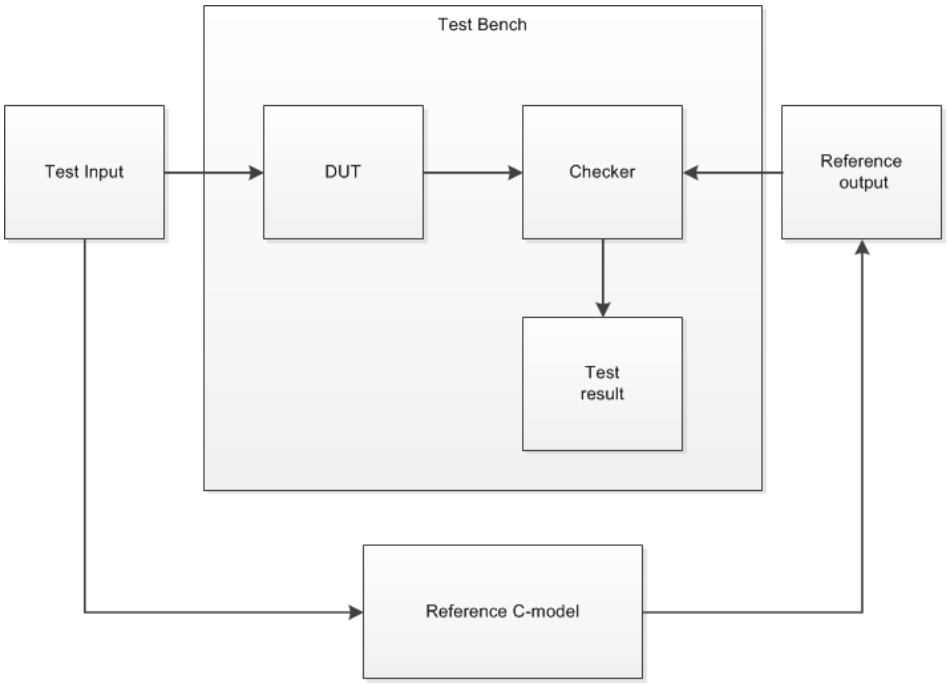

Figure 29: Test bench block diagram

4.3. Simulation and Synthesis Results

Simulation and synthesis results of BPC and BAC design are discussed under this subsection. This section also discusses important control signal associated with BPC and BAC with waveforms. Device utilization and timing details of synthesized BPC and BAC design are discussed in detail. The Target FGPA device for which BPC top and BAC top is synthesized is "$Virtex-5$" Xilinx FPGA family with device id "$XC5VLX110T$", package id "$FF1136$" and speed grade "-1" [12].

4.3.1. BPC

Entity of top level BPC VHDL design is as shown in Table 11. *clk* and *reset* input signals are driven by the testbench. These signals are used to drive *clk* and *reset* inputs of all the instances in BPC design. BPC top has a memory interface to read pixel data associated with the code-block. Input and output ports associated with memory interface are defined as sub-types *mem_port_out* and *pixel_mem_port_in* respectively. Description of these subtypes can be found under appendix A. *pixel_data_valid* control signal is an input to BPC top, it is used to indicate that the pixel data of code-block is valid to be read and encoded.

Bit-planes of the code-block are encoded in three coding passes as discussed under subsection 2.4. Simulation waveform of BPC coding passes is as shown in Figure 30. Control signals $start\_spp/start\_mrp/start\_cup$ and $done\_spp/done\_mrp/done\_cup$ are used to indicate beginning and end of coding pass SPP/MRP/CUP respectively. All bit-planes apart from the MSB bit-plane are encoded by SPP, MRP and CUP coding pass in sequence as shown in Figure 30.

Simulation waveforms of SPP, MRP and CUP encoding are as shown in Figure 31, 32 and 33 respectively. At the start of each of these encoding pass, the memories corresponding to state variables and bit-planes necessary for encoding are read and the state associated with this is *read_mem*. At the end of each of these encoding pass, the memories corresponding to state variables are updated and the state associated with this is *write_mem*. States in between *read_mem* and *write_mem* are encoding states of coding pass. Encoding states of SPP, MRP and CUP coding pass are discussed in detail under subsection 2.4.

Maximum and minimum number of clock cycles required to encode a bit in a bit-plane of a code-block for SPP, MRP and CUP coding pass is as shown in Table 12. The numbers of clock cycles mentioned in the Table 12 are number of clock cycles required for pure encoding computation. It does not include the number of clock cycles required to read state variables and bit-plane from RAMs or writing back state variables into RAMs after encoding.

Minimum number of clocks required for all three coding pass is 1 clock cycle. Input bit is checked with certain condition at the beginning of encoding algorithm in each coding pass. If the condition is satisfied then the input bit is encoded otherwise encoding is skipped and algorithm checks with next input bit. This condition check on input bit requires one clock cycle, which is the minimum number of clock cycles required. Maximum number of clock cycles required to encode a bit depends on the number of states associated with each coding pass. The states associated with SPP, MRP and CUP coding pass can be found in Figure 6, 7, and 8 respectively.

Summary of BPC synthesis report is as shown in Table 13, 14, 16 and 15. Device resource utilization is as shown in Table 13. BPC utilizes 21% of FPGA LUT resource, however considerable amount of these resources are used by distributed RAMs associated with state variables and bit-planes. There are 14 RAMs in the design intended for state variables and bit-planes, out of which one is synthesized as block RAM and the rest are synthesized as distributed RAMs.

Macro statistics of BPC synthesis is as shown in Table 14. Synthesis tool has recognized 5 FSMs in BPC design, 1 FSM from each of the coding pass and 2 FSMs from BPC top. Look

up tables used for SC, ZC and MRC are synthesized as ROMs, 4 such ROMs are recognized in BPC design. Count of computational elements such as adder/subtractor, counter, comparator, multiplexer, register and xor synthesized in BPC design are as shown in Table 14.

BPC synthesis timing details are as shown in Table 15 and 16. Maximum operating frequency of BPC design is 128.758 MHz. BPC design requires setup time of 2.784 nanoseconds and hold time of 3.270 nanoseconds. Critical path details in the design are as shown in Table 16. It has delay of 7.766 nanoseconds and 9 levels of logic. CUP coding pass in BPC has the critical path, the combinational logic responsible for checking condition for applying RLC coding is the critical path in the design. Condition check for applying RLC coding is shown in flowchart 8. *Rowpositioncount* register of the current sample is the source flip flop and *nextstate* register is the destination flip flop.

Table 11: Entity of BPC top

Generic		
Generic name	Generic type	Default value
CB_SIZE	*integer*	31
Port		
Port name	Port mode	Port type
reset	*in*	*std_logic*
clk	*in*	*std_logic*
pixel_ram_out	*in*	*mem_port_out*
pixel_data_valid	*in*	*std_logic*
pixel_ram_in	*out*	*pixel_mem_port_in*

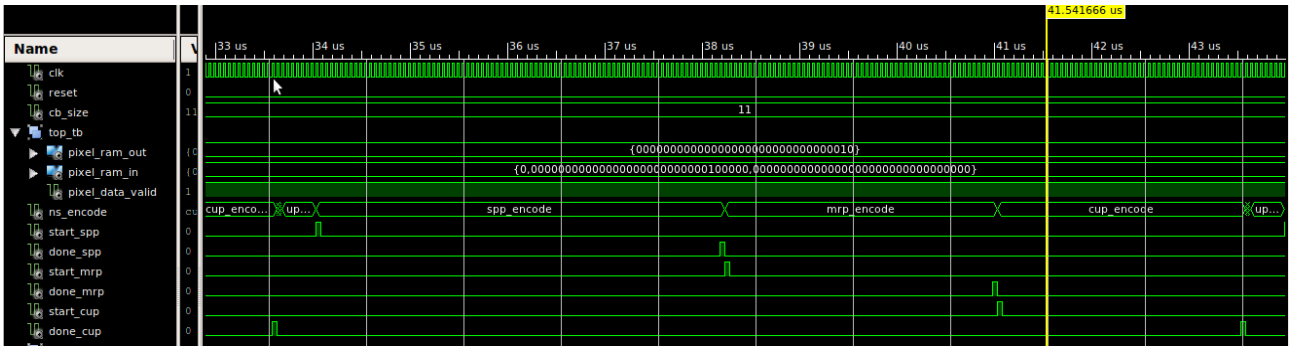

Figure 30: BPC simulation waveform with coding pass sequence

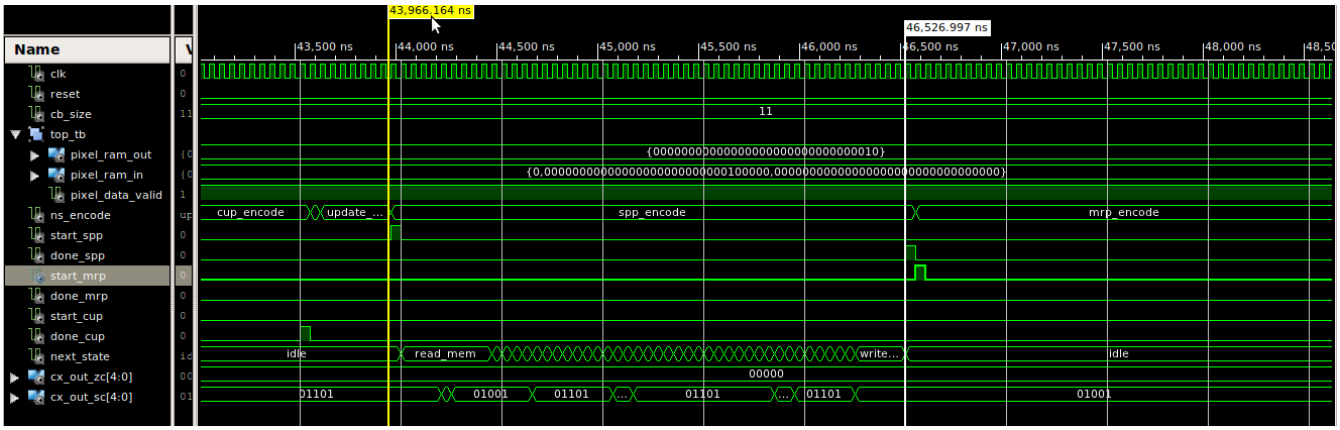

Figure 31: BPC SPP simulation waveform

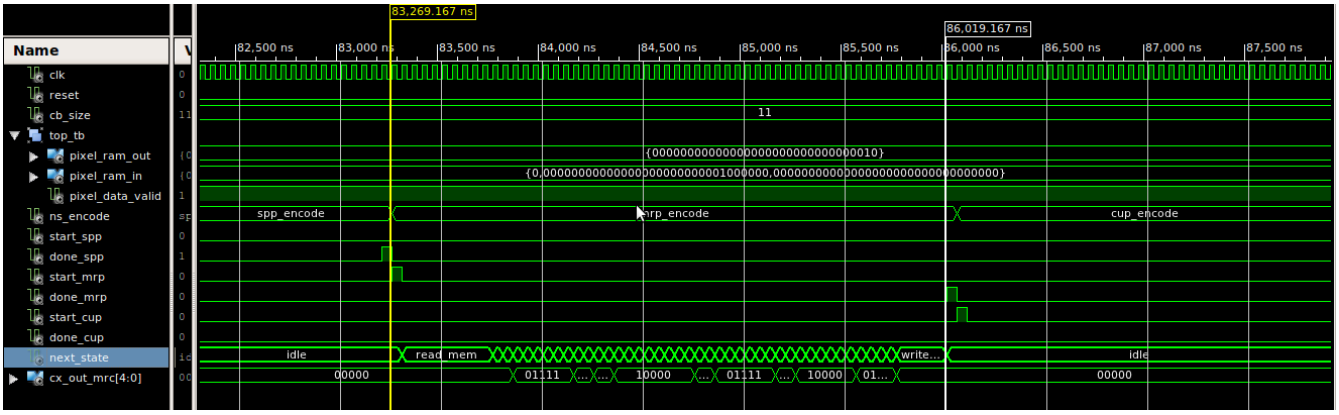

Figure 32: BPC MRP simulation waveform

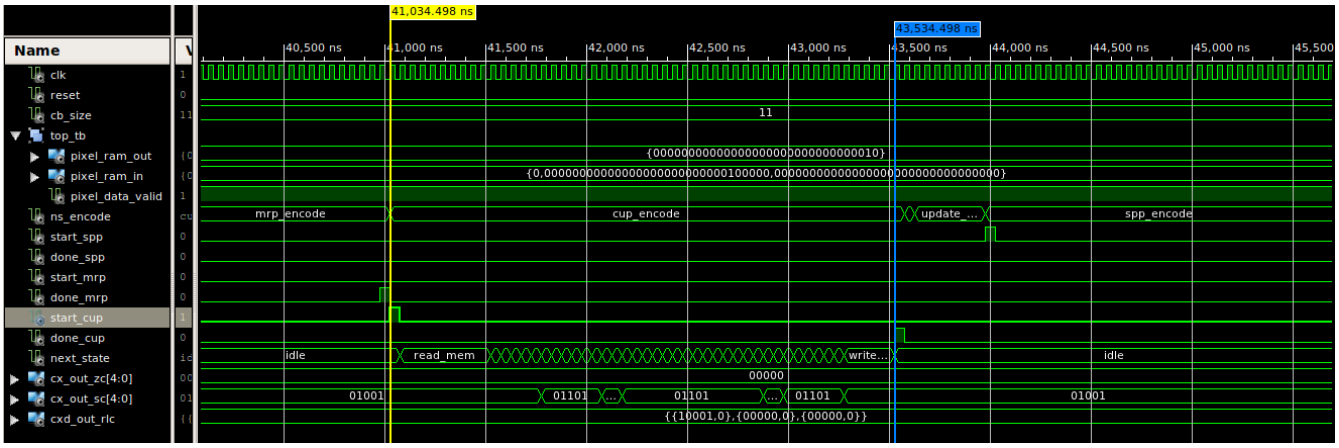

Figure 33: BPC CUP simulation waveform

Table 12: BPC encoding profile

Coding pass	Minimum clock cycles	Maximum clock cycles
SPP	1	6
MRP	1	4
CUP	1	6

Table 13: BPC synthesis device utilization summary

Device utilization summary		
Slice Logic Utilization	Number of Slice Registers	12949 out of 69120 (18%)
	Number of Slice LUTs	15096 out of 69120 (21%)
	Number used as Logic	14976 out of 69120 (21%)
IO Utilization	Number of IOs	100
	Number of bonded IOBs	78 out of 640 (12%)
Specific Feature Utilization	Number of BUFG/BUFGCTRLs	2 out of 32 (6%)
	Number of Block RAM/FIFO	1 out of 148 (0%)

Table 14: BPC synthesis macro statistics

Macro Statistics		
FSMs	—	5
RAMs	32x32-bit dual-port block RAM	1
	32x32-bit dual-port distributed RAM	13
ROMs	4x32-bit ROM	4
Adders/Subtractors	1-bit adder carry out	7
	2-bit adder	3
	2-bit adder carry out	5
	3-bit adder	6
	32-bit subtractor	1
	5-bit adder	11
	6-bit adder	7
	8-bit adder	9
	8-bit subtractor	7
Counters	32-bit up counter	2
Registers	Flip-Flops	15176
Comparators	2-bit comparator greatequal	4
	3-bit comparator greatequal	11
	3-bit comparator greater	4
	32-bit comparator greatequal	3
	32-bit comparator greater	10
	32-bit comparator less	7
	32-bit comparator lessequal	10
	8-bit comparator less	8
Multiplexers	1-bit 32-to-1 multiplexer	83
	1-bit 34-to-1 multiplexer	42
	32-bit 32-to-1 multiplexer	10
	34-bit 34-to-1 multiplexer	14
Xors	1-bit xor2	2

Table 15: BPC synthesis timing summary

Timing summary	
Minimum period	7.766 ns
Maximum Frequency	128.758 MHz
Minimum input arrival time before clock	2.784 ns
Maximum output required time after clock	3.270 ns
Maximum combinational path delay	0.550 ns
*ns :- nanoseconds, *MHz :- Mega Hertz	

Table 16: BPC critical path details

Critical path	
Delay	7.766ns
Levels of Logic	9
Source	cup_inst_top/reg_row_count_reg_0 (FF)
Destination	cup_inst_top/next_state_FSM_FFd5 (FF)
*FF :- Flip Flop	

4.3.2. BAC

Entity of BAC top VHDL code is as shown in Table 17. Input port *cxd_in* receives CX and D value to be encoded from the testbench. *cxd_in* signal input value is validated by input control signal *cxd_valid*. *cxd_done* input signal indicates the end of the encoded symbols (*cxd_in*) associated with the code block. *reset* and *clk* input signals are generate by the testbench, these signals are also connected to entities of the instances in BAC top. *cxd_read* output control signal is asserted by BAC top to indicate that the current value of *cxd_in* is read and the testbench can drive the next *cxd_in* input value. Compressed byte generated by BAC is driven on *byteout_out* output signal. *byteout_valid* ouput control signal is used to validate *byteout_out* signal. BAC top asserts *done* output control signal after encoding of the entire code block. Description of derived data types used in BAC top can be found under appendix A.

Simulation waveform of BAC testbench is as shown in Figure 34. BAC may encode multiple symbols before generation of one compressed byte, one such example is shown in the Figure 34. BAC top reads valid *cxd_in* input, encodes the input symbol, if the compressed byte is not generated in the current iteration then reads the next *cxd_in* to be encoded. If the compressed byte is generated then BAC top drives compressed byte on *byteout_out* and asserts the *byteout_valid* signal. BAC testbench reads valid *byteout_out* and compares it with the reference signal *datareadout* and updates the test results in case of comparison mismatch.

Maximum number of clocks required to encode a symbol are 17 clocks. States associated with BAC top state machine are as shown in Figure 10. State $ReadCX, D$ and $D == MPS(CX)$ consumes one clock and state $codeMPS/codeLPS$ can consume up to 15 clocks. Renormalization stage in state $codeMPS/codeLPS$ is responsible for large number of clocks required in this state. Renormalization stage iterates in a loop for computation of compressed byte.

Minimum number of clocks required to encode a symbol are 3 clocks. State associate with such encoding scenario are $(ReadCX, D)$, $(D == MPS(CX))$ and $codeMPS$, where $codeMPS$ state does not invoke renormalization computational stage.

Number of clocks requires to generate a compressed byte can vary depending on the input symbols being encoded. A compressed byte may be generated after encoding one symbol; however it may also be generated after encoding several symbols. Number of symbols required to generate a compressed byte purely depends on the values of input symbols.

Summary of synthesis report is as shown in Table 18, 19, 21 and 20. NMPS, NLPS, SWITCH and Qe LUTs are inferred as ROMs during synthesis. Index and MPS LUTs are inferred as latches since these LUTs are accessed both in write and read mode and are designed as combinational logic. Synthesis results associated with LUTs are as shown in Table 19.

Timing details of the synthesis report are as shown in Table 20. Maximum operating frequency of the design is 112.927 MHz. The critical path in the design is depicted by blue line in Figure 35. The path starts from input block where the CX value is read, Index associated with CX is read from index LUT, read index value is used for accessing Qe associated with CX, read Qe value is used by subtractor and comparator in MPS computation block to calculate A and C register values. Critical path details with source and destination flip flop is as shown in Table 21.

Table 17: Entity of BAC top

Port name	Port mode	Port type
cxd_in	*in*	*cxd_pair*
cxd_valid	*in*	*std_logic*
cxd_done	*in*	*std_logic*
reset	*in*	*std_logic*
clk	*in*	*std_logic*
cxd_read	*out*	*std_logic*
byteout_out	*out*	*std_logic_vector*(7*downto*0)
byteout_valid	*out*	*std_logic*
done	*out*	*std_logic*

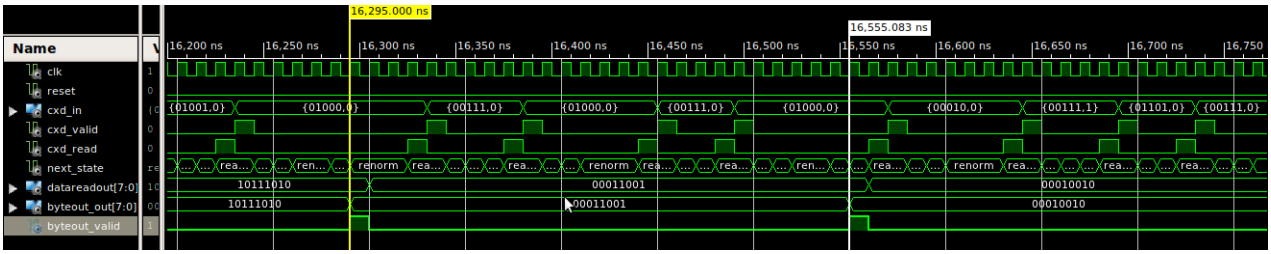

Figure 34: BAC simulation waveform

Table 18: BAC synthesis device utilization summary

Device utilization summary		
Slice Logic Utilization	Number of Slice Registers	257 out of 69120 (0%)
	Number of Slice LUTs	635 out of 69120 (0%)
	Number used as Logic	635 out of 69120 (0%)
IO Utilization	Number of IOs	21
	Number of bonded IOBs	21 out of 640 (3%)
Specific Feature Utilization	Number of BUFG/BUFGCTRLs	2 out of 32 (6%)

Table 19: BAC synthesis macro statistics

Macro Statistics		
FSMs	—	1
ROMs	47x1-bit ROM	1
	47x16-bit ROM	1
	47x6-bit ROM	2
Adders/Subtractors	32-bit adder	1
	32-bit subtractor	1
	4-bit subtractor	1
	8-bit adder	1
Registers	Flip-Flops	113
Latches	1-bit latch	19
	6-bit latch	19
Comparators	32-bit comparator greatequal	3
	32-bit comparator less	4
Multiplexers	1-bit 19-to-1 multiplexer	1
	6-bit 19-to-1 multiplexer	1
Logic shifters	32-bit shifter logical left	1
	32-bit shifter logical right	1

Table 20: BAC synthesis timing summary

Timing summary	
Minimum period	8.855 ns
Maximum Frequency	112.927 MHz
Minimum input arrival time before clock	9.103 ns
Maximum output required time after clock	3.264 ns
*ns :- nanoseconds, *MHz :- Mega Hertz	

Table 21: BAC critical path details

Critical path	
Delay	8.855ns
Levels of Logic	33
Source	cxd.cx_3 (FF)
Destination	c_reg_0 (FF)
*FF :- Flip Flop	

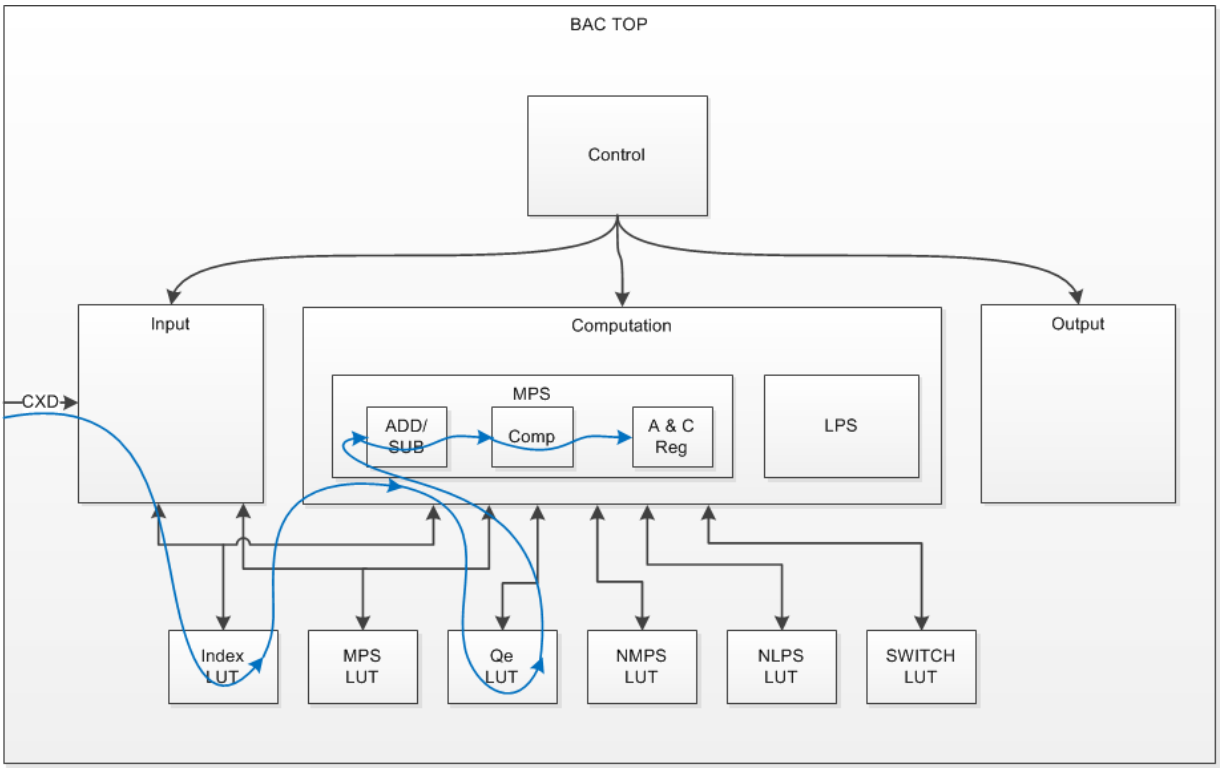

Figure 35: BAC critical path

5. EBCOT ENCODER PARALLEL ARCHITECTURE

Tier-1 encoder consumes considerable amount of overall JPEG2000 compression time as shown in subsection 3.3.3, in order to reduce compression time it is necessary to have parallel hardware architecture for EBCOT encoder. This section discusses several parallel architectural proposal, their advantages and drawbacks in detail.

5.1. Serial hardware architecture overview

This subsection gives an overview of the JPEG2000 compression system architecture in serial mode. Figure 1 shows the block diagram of major components of JPEG2000 encoder, the block diagram remains same for serial hardware architecture with only difference being Tier-1 encoder block implemented in software is replaced now by Tier-1 encoder implemented in hardware. Rest of the components of Figure 1 still remains software implemented.

Hardware implemented Tier-1 gives considerable increase in performance with respect to compression time as compared to software implementation of Tier-1. However JPEG2000 architecture exhibits great potential for parallel hardware architecture. Block diagram 36 shows JPEG2000 serial hardware encoder architecture with decomposition of image into sub-bands and code blocks. Decomposition of image by DWT is described in detail under subsection 2.1. For simplicity the sub-bands are shown as array in Figure 36.

Sub-bands generated by DWT are further divided into code-blocks of fixed sizes. Code-blocks are further decomposed into bit-planes by Tier-1 encoder. Tier-1 encoder does BPC encoding on each of these bit-planes to generate encoded symbols. In serial architecture only one hardware instance of Tier-1 encoder is used as shown in Figure 36. Hence encoding of all the code-blocks has to be done sequentially. It can be observed from the Figure 36 that serial architecture has potential for parallelization. Several parallel architectures are discussed in detail in the subsequent subsections.

Table 22: State variable access table

Processing Blocks	State Variables									
	η		σ		σ'		υ		χ	
	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read
ZC								✓		
SC				✓*						✓
MRC				✓*		✓				
RLC								✓		
SPP	✓		✓	✓*						
CUP		✓	✓	✓**						
MRP		✓		✓	✓					

$*$ Accesses 8 surrounding neighbors of the current element(shown in 37 a)).

$**$ Accesses 4 consecutive elements on the same column and also all adjacent neighbors of the four consecutive elements (shown in 37 b)).

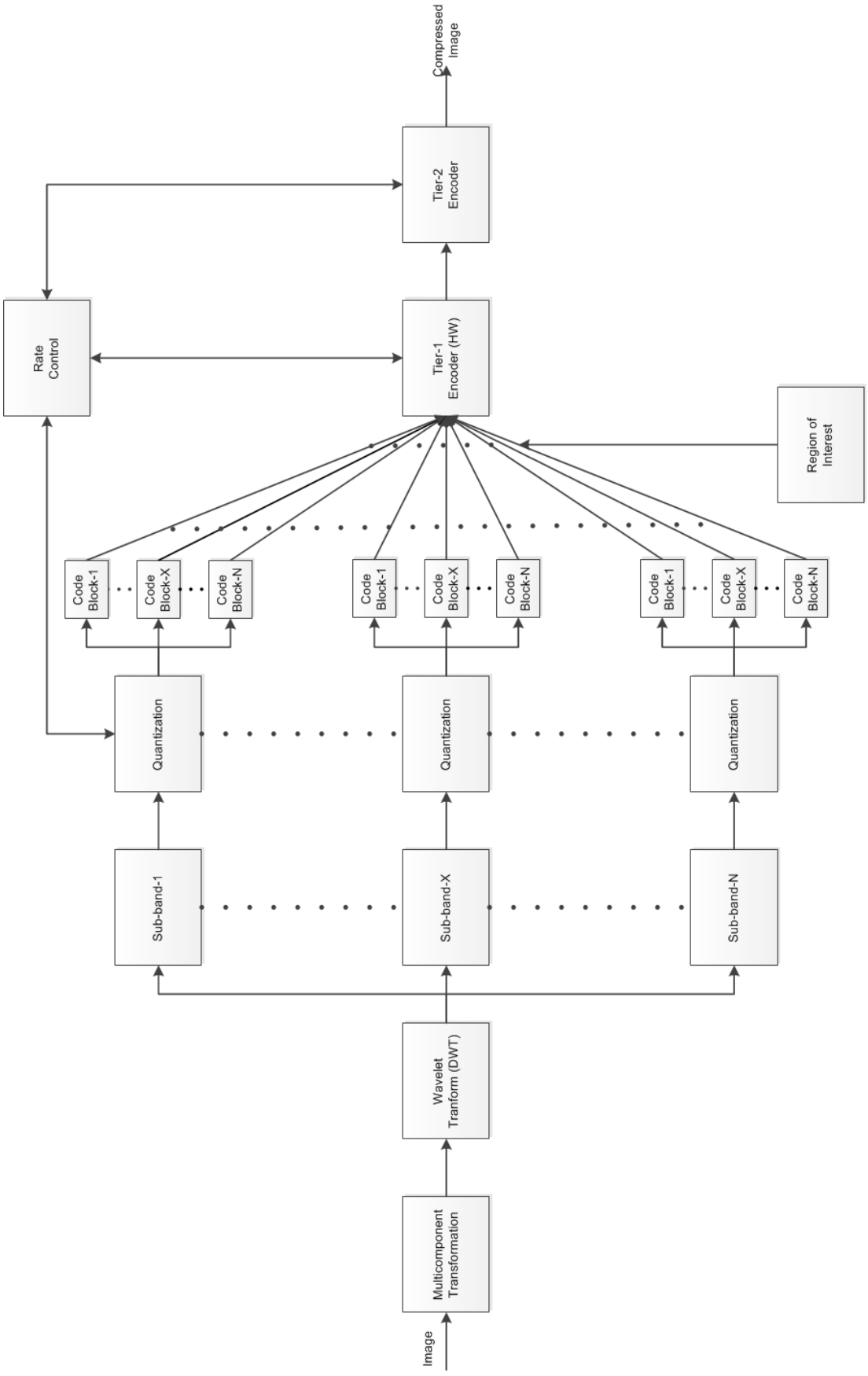

Figure 36: JPEG2000 encoder compression system architecture in serial mode

5.2. Parallel coding pass architecture

BPC decomposes each code-block into bit-planes. SPP, MRP and CUP coding passes encode each bit-plane in sequence respectively, with the exception of MSB bit-plane. Only CUP coding pass encodes MSB bit-plane. In serial hardware architecture coding pass encode bit-plane sequentially, that is MRP coding pass has to wait until SPP coding pass has encoded the bit-plane completely and CUP coding pass has to wait until MRP coding pass has encoded the bit-plane completely.

The primary reason for sequential coding pass is that the state variable used during encoding are shared between coding passes and bit-planes of the code-block. State variables hold vital information about the bit-planes, these informations are used by coding passes for taking encoding decisions. Hence to parallelize coding passes it is important to understand the interdependences of coding passes because of state variables.

Table 22 summarizes the accesses done to each state variable by different encoding blocks of Tier-1. A check mark against write/read column of a state variable means that the state variable's location corresponding to the location of element being encode was accessed in write/read mode. Some encoding blocks may have to access more than one location of state variable for encoding one element of the bit-plane; these exceptions are mentioned in the foot note of the Table 22.

It is observed from the Table 22 that $\sigma, \sigma^{'}$ and η state variables are accessed in both write and read mode by the encoding blocks and state variables υ^{p} and χ are accessed in read mode only. State variables accessed in write mode are of primary importance for parallelization. In case of $\sigma^{'}$ and η state variable arrays, only the state variable element's location corresponding to location of the element being encoded in bit-plane has to be updated, before being encoded by the next coding pass.

Neighbor elements accessed in σ array is as shown in Figure 37. Blocks with red color corresponds to the location of element being encoded in bit-plane and blocks with blue color corresponds to neighbor locations being accessed by encoding blocks. Figure 37 a) represents the access pattern of σ array elements by SPP, MRC and SC blocks in read mode. Figure 37 b) represents the access pattern of σ array elements by CUP block in read mode.

It is evident from the Figure 37 that to parallelize coding passes at least the elements depicted in Figure 37 b) has to be updated by previous encoding pass before the next coding pass starts encoding the bit-plane in parallel. To reduce the complexity of architecture the condition for parallelization can be simplified to "At least four rows of σ array has to be updated by previous coding pass before next coding pass starts encoding the bit-plane in parallel".

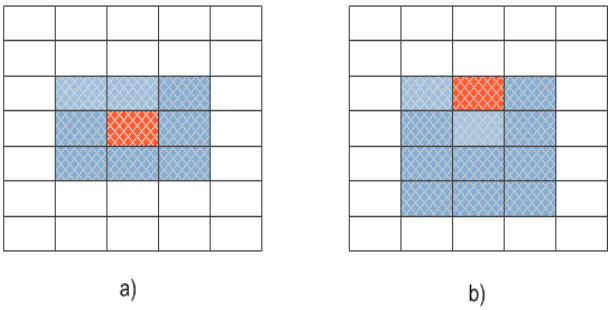

Figure 37: Neighbor elements accessed in σ array

Figure 38 shows the conceptual timing diagram of coding pass. "t" in the Figure 38 represents time consumed by the coding pass to encode a line of a bit-plane. "T" represents the time consumed by the coding pass to encode an entire bit-plane of a code-block. To reduce the complexity of analysis we assume all coding pass consumes same amount of time to encode a line of bit-plane and also same amount of time to encode entire bit-plane. Architecture does not parallelize encoding of MSB bit-plane since it is encoded only by CUP coding pass. Encoding of all bit-planes of a code-block is parallelized except the MSB bit-plane.

Figure 38 a) shows the time consumed by coding passes to encode a bit-plane in case of serial architecture. Total time consumed for encoding a bit-plane is given by equation 8, since the coding passes encode bit-plane sequentially. Figure 38 b) shows the time consumed by coding passes to encode a bit-plane in case of parallel coding pass architecture. Total time consumed for encoding a bit-plane is given by equation 9

$$3T \tag{8}$$

$$T + 4t + 4t \tag{9}$$

For simplicity of calculation, assume encoding of each sample in a bit-plane consumes one clock cycle for all coding passes. Let the code-block size be 32×32. Number of clock cycles required to encode a bit-plane in case of serial architecture is given by equation 12. Number of clock cycles required to encode a bit-plane in case of parallel coding pass architecture is given by equation 13. Speedup factor achieved by parallel coding pass architecture with respect to serial architecture in encoding a bit-plane is given by equation 16. Assuming number of bit-plane in the code-block to be 7. Number of clock cycles required to encode entire code-block in case of serial architecture is given by equation 14. Number of clock cycles required to encode a code-block in case of parallel coding pass architecture is given by equation 15. Speedup factor achieved in encoding a complete code-block is given by equation 17.

$$T = 32 \times 32 = 1024 \tag{10}$$

$$t = 32 \tag{11}$$

substituting 10 in equation 8

$$3 \times 1024 = 3072 \tag{12}$$

substituting 10 and 11 in equation 9

$$1024 + 4 \times 32 + 4 \times 32 = 1280 \tag{13}$$

$$1024 + 3 \times 1024 \times 6 = 19456. \tag{14}$$

$$1042 + 1024 + 4 \times 32 + 4 \times 32 + 5 \times 12 \times 32 = 4224. \tag{15}$$

$$Speedup\ \ w.r.t\ \ bit - plane = \frac{Serial\ \ architecture\ \ clocks}{Parallel\ \ architecture\ \ clocks} = \frac{3072}{1280} = 2.4 \tag{16}$$

$$Speedup\ \ w.r.t\ \ code - block = \frac{Serial\ \ architecture\ \ clocks}{Parallel\ \ architecture\ \ clocks} = \frac{19456}{4224} = 4.6 \tag{17}$$

Parallel coding pass architecture has a speedup factor of 2.4 and 4.6 as compared to serial architecture in encoding a bit-plane and code-block respectively. Speedup achieved by parallel coding pass in case of encoding a code-block is higher than encoding a bit-plane because encoding of a subsequent bit-plane by all three coding pass put together consumes only $12 \times t$ clocks.

Parallel coding pass architecture needs additional control signals to synchronize the coding passes for encoding; however the hardware logic overhead added by this architecture is negligible as compared to speedup achieved. Parallel coding pass architecture only parallelizes the coding algorithms it does not change the encoding flow in itself, hence the decoder will be transparent to this architectural changes in encoder. Downside of parallel coding pass architecture is that coding passes were assumed to consume same number of clocks to encode a bit and to encode a complete line, however in practice this may not be true and architecture may need additional logic to synchronize the coding passes.

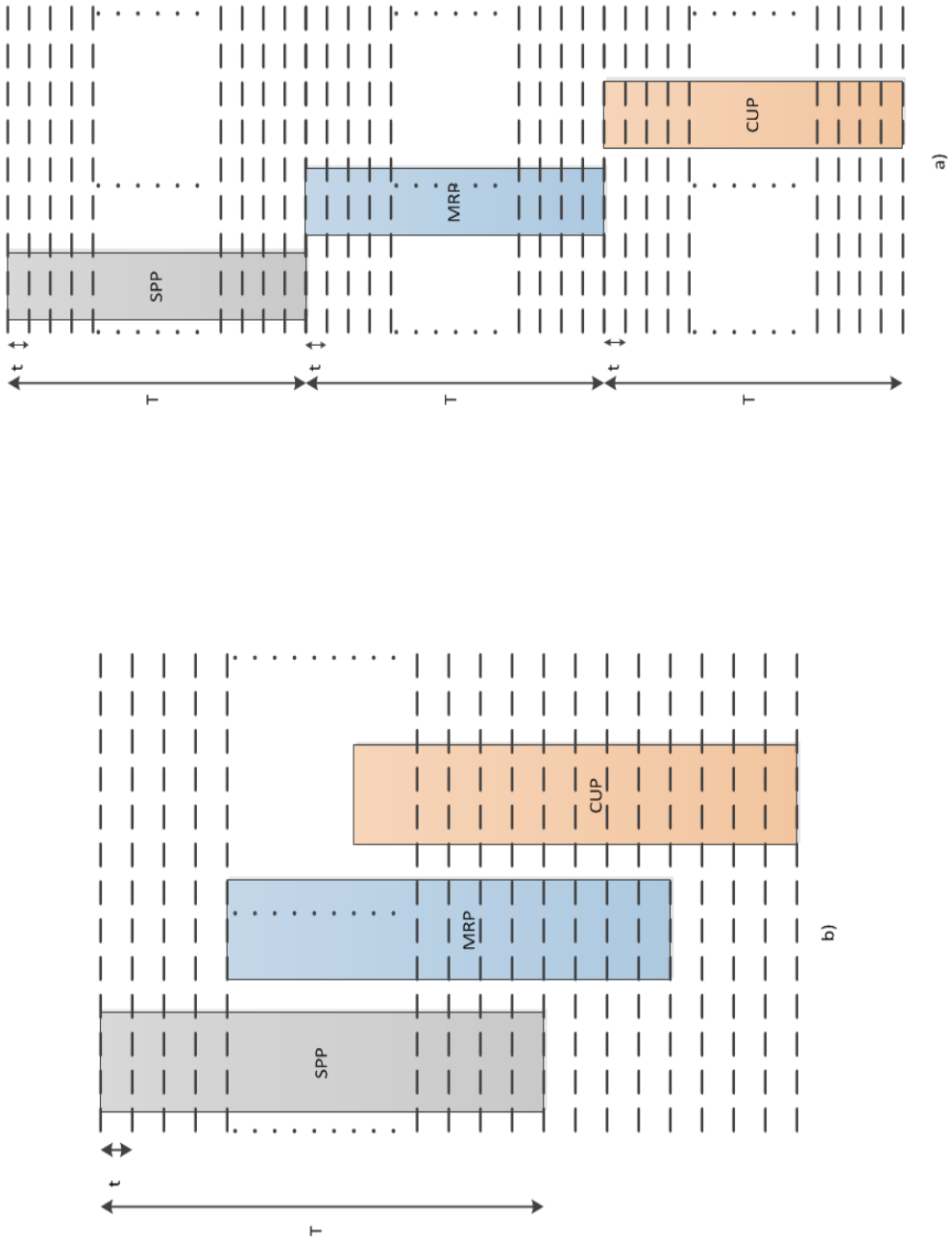

Figure 38: Conceptual timing diagram of coding passes

5.3. Parallel Sub-band architecture

Discrete wavelet transform (DWT) block in JPEG2000 decomposes image into low frequency and high frequency sub-bands. Subsection 2.1 discusses images decomposition in detail. At higher level of decomposition of image, sub-band LL is recursively decomposed into next level LL, LH, HL and HH sub-bands. Sub-bands LH, HL and HH are encoded by the algorithm and sub-band LL is used by DWT to generate higher level sub-bands.

Parallel sub-band architecture block diagram is as shown in Figure 39. Sub-bands LH, HL and HH are encoded in parallel, hence this architecture needs 3 instances of quantization block and Tier-1 encoding blocks [13, 14]. Since the sub-bands are encoded in parallel, this architecture has the potential to achieve maximum speedup factor of 3 as compared to serial hardware architecture.

In practice effective speedup of parallel sub-band architecture may be slightly less than 3 because of single instance of Tier-2 encoder. Tier-2 encoder in parallel sub-band architecture has to sequentially process output generated from three instance of Tier-1 encoder. However processing time consumed by Tier-2 encoder is not significant as compared to rest of the processing blocks; hence the negative effect on speedup factor of parallel sub-band architecture is not significant.

Parallel sub-band architecture exhibits great prospects in reducing JPEG2000 algorithm compression time. However the downside of this architecture is that it requires significant amount of hardware resource for parallel processing of sub-bands. This architecture may also need slight modification in DWT block however it does not affect the encoding algorithm in itself, hence this architecture is transparent to JPEG2000 decoder.

5.4. Parallel BPC architecture

Parallel BPC architecture block diagram is as shown in Figure 40. Parallel BPC architecture is an extension of parallel sub-band architecture. Parallel BPC architecture has instance of BPC for every code-block branch from a sub-band. Hence Tier-1 block has "N" instance of BPC block and one instance of BAC block.

Tier-1 consumes considerable compression time as discussed in subsection 3.3.3. In Tier-1 BPC is more computational expensive as compared to BAC. Parallelizing Tier-1 block with one instance of BPC for every code-block has positive effect on decreasing computational time of Tier-1 block. However the encoded symbols generated by BPC are still sequentially processed by BAC to generated compressed bytes.

BAC block in Tier-1 cannot be instantiated for very code block of the sub-band because of the inherent nature of BAC algorithm. BAC algorithm is a kind of cumulative algorithm, BAC accumulates the information of several input symbols during processing before generation of single compressed byte and hence having parallel instance of BPC alters the JPEG2000 encoding algorithm.

Parallel BPC architecture greatly reduces the computation time consumed by BPC phase of Tier-1; however the downside is that the parallel BPC architecture requires huge hardware resources. Advantage over computational time achieved by parallel BPC can be brought down because of single instance of BPC. Since the encoding algorithm's integrity is still maintained, parallel BPC architecture is transparent to JPEG2000 decoder.

5.5. Parallel BAC architecture

Parallel BAC architecture is an extension of parallel BPC architecture. Parallel BAC architecture is as shown in Figure 41. It has one instance of BPC and BAC for every code-block of sub-bands. It is as good as having Tier-1 encoder block for every code-block of sub-band. Theoretically parallel BAC architecture consumes same amount of time to encode an entire image and to encode a single code-block, since all the code blocks are encoded in parallel.

Biggest downside of parallel BAC architecture is that, it changes JPEG2000 encoder algorithm by parallelizing BAC block. Parallel BAC architecture has one instance of BAC for every code block, because of this compression computational information carried over from a code-block to the subsequent code-block is lost. This divergence in the algorithm causes divergence in compliance with standard JPEG2000 encoder. Hence parallel BAC architecture is not transparent to JPEG2000 decoder. Compressed image size of parallel BAC architecture can be higher than the standard JPEG2000 compressed image size.

Parallel BAC architecture also has huge hardware resource requirements. However it exhibits very high compression speed and can be used in customized hardware applications for real time image compression. Parallel BAC architecture needs corresponding architectural changes in decoder for recovering the original image.

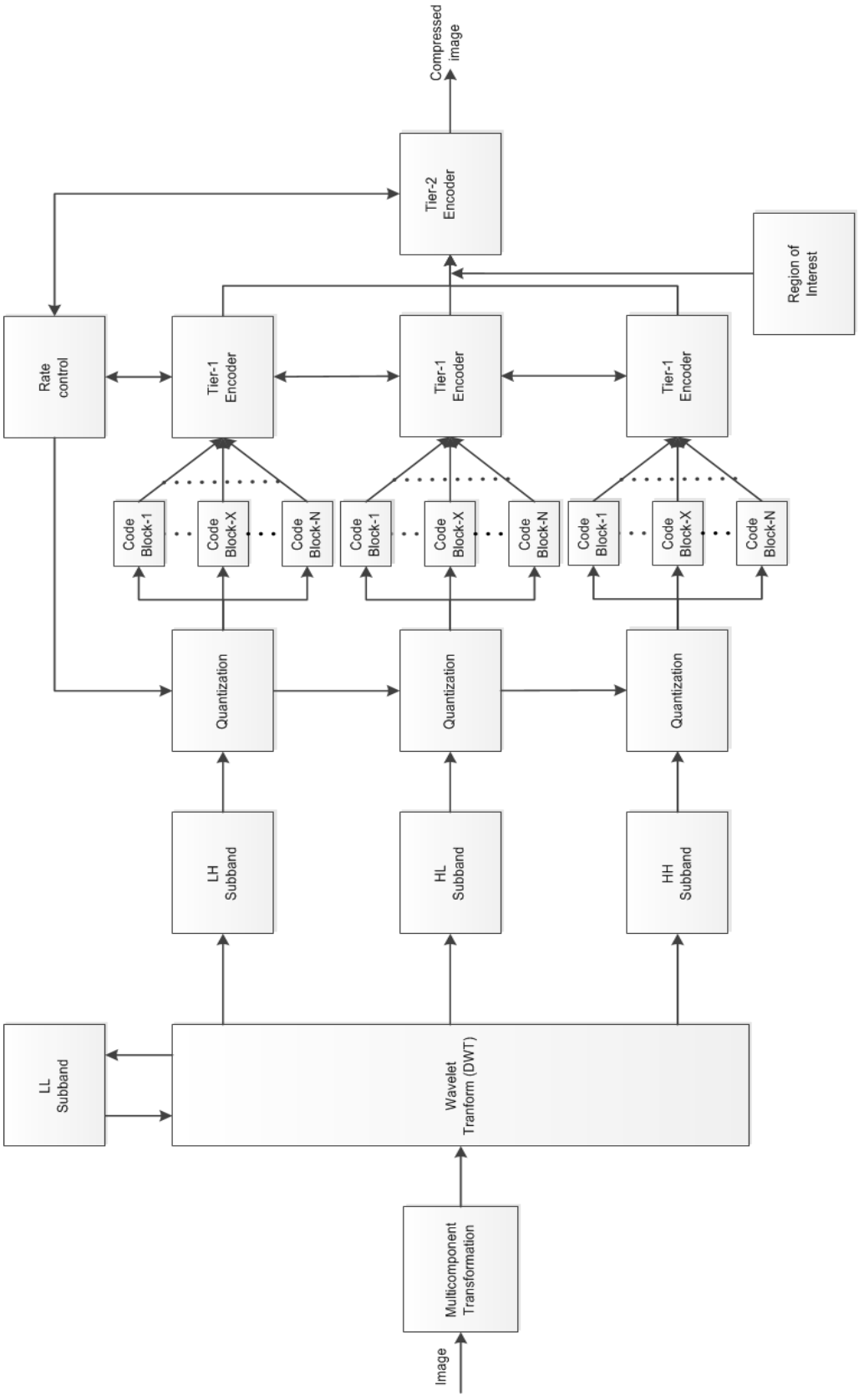

Figure 39: Parallel sub-band architecture block diagram

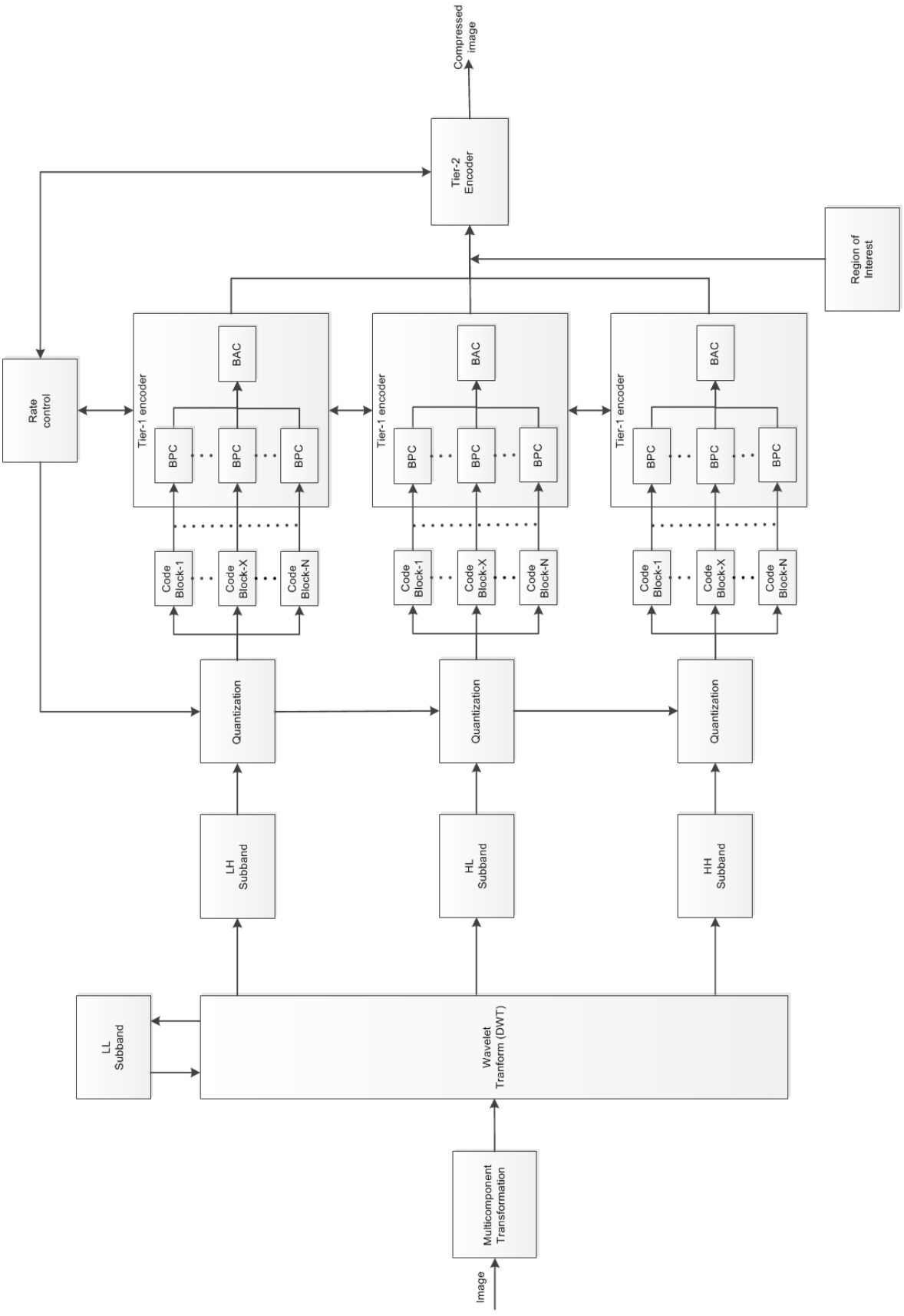

Figure 40: Parallel BPC architecture block diagram

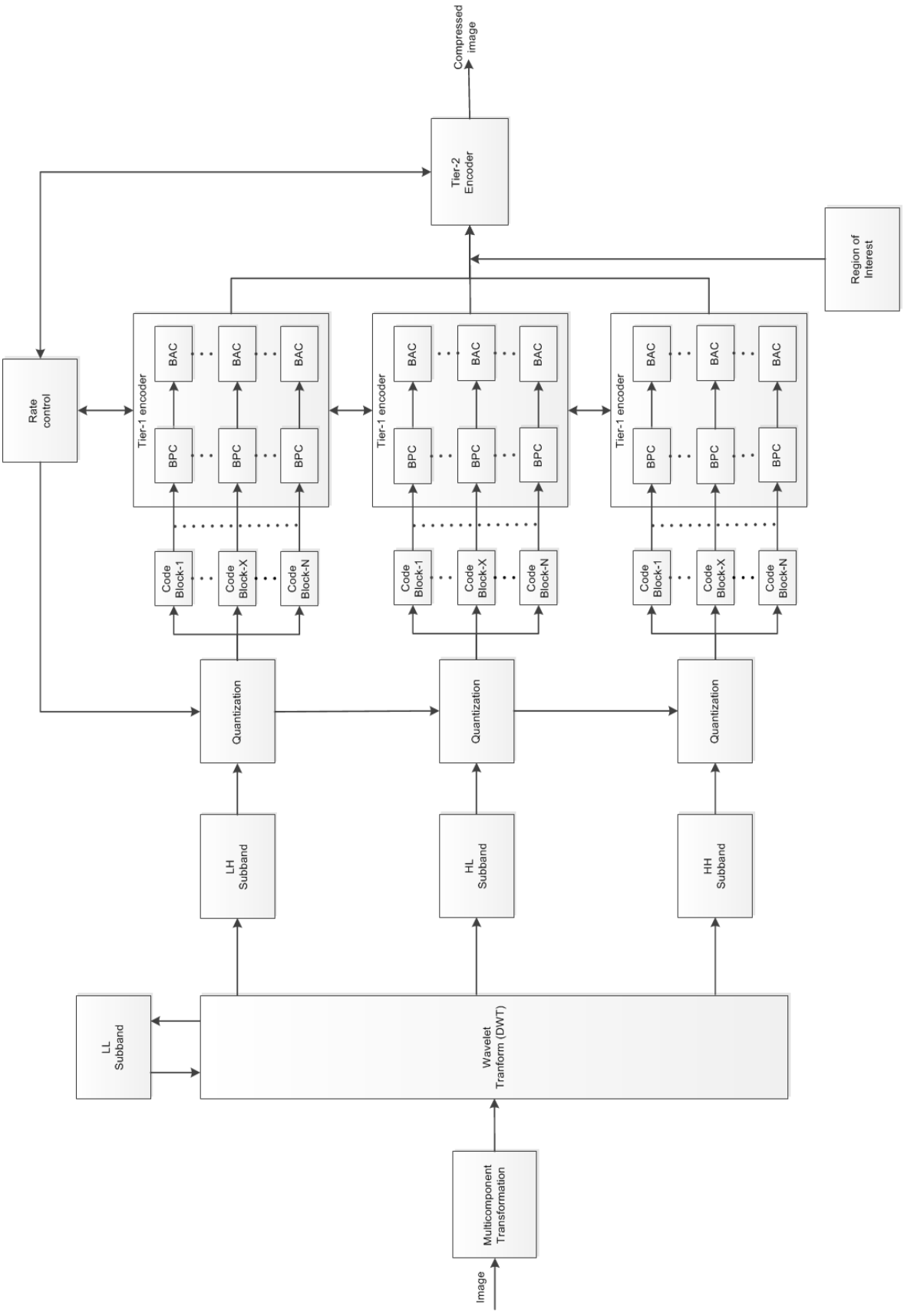

Figure 41: Parallel BAC architecture block diagram

6. CONCLUSION

Thesis introduces JPEG2000 compression algorithm. JPEG2000 salient features were discussed and compared with other famous compression algorithms. EBCOT and MQ-encoder algorithms were presented in detail under section 2. JPEG2000 software implementation was presented under section 3. Profiling of JPEG2000 software was conducted and profiling results were discussed in detail under subsection 3.2. Profiling results of JPEG2000 software implementation has shown that Tier-1 coding phase of JPEG2000 consumes considerable portion of overall compression time.

Hardware design of BPC and BAC were presented in detail under section 4. Simulation and synthesis results were also presented under subsection 4.3. It was found that BPC is computationally exhaustive as compared to BAC from the FPGA resource utilization. Synthesis results has shown that BPC and BAC hardware design can work at maximum clock frequency of 128.758 MHz and 112.927 MHz. Hence hardware implementation of BPC and BAC exhibits great prospects in reducing compression time as compared to software implementation of BPC and BAC.

JPEG2000 compression algorithm architectures were presented for real time image compression in section 5. Architectures proposed in this section have great potential in reducing the compression time considerably in order to do image compression in real time. Conclusion from profiling results of software implementation was the basis for parallel architectural proposals.

Parallel coding pass architecture presented in subsection 5.2 offers a speedup factor of upto 4.6 as compared to serial hardware architecture. Parallel sub-band architecture offers great potential in reducing compression time by compressing sub-bands in parallel. Parallel BPC architecture takes advantage of JPEG2000 algorithm flow by processing the code-blocks in parallel. Parallel BAC architectures exhibits highest level of parallel processing of image however this architecture is not compliant to JPEG2000 standard, while rest of the parallel architectures proposed in this thesis are compliant to JPEG2000 standard and are transparent to JPEG2000 decoder.

Architectures derived from the combination of two parallel architectures proposed in this thesis, can offer higher compression performance. Parallel coding pass architecture combined with parallel sub-band or parallel BPC or even both can offer much higher performance, however these parallel architectures comes at the expense of hardware resources. Hence based on the application requirements, architecture with optimal combination of hardware resources and compression performance can to be chosen.

A. VHDL Subtype Package

```
package types is
  subtype context is std_logic_vector(4 downto 0);
  subtype sigma_dash is std_logic; -- used in MRC coding
  subtype vp is std_logic; -- Vp[m, n]

  -- Two dimensional array register used for Sigma state variable
  subtype word34 is std_logic_vector (33 downto 0);
  type sigma_array is array (33 downto 0) of word34;

  -- Two dimensional array register used for sigma_dash, eta,
  -- vp state variable
  -- signal (state_reg_array_number) (word32_number) <= 'bit'
  subtype word32 is std_logic_vector (31 downto 0);
  type state_reg_array is array (31 downto 0) of word32;

  subtype subband is std_logic_vector(1 downto 0);

  -- CX and D output for each bit of bit-plane in SPP
  type spp_cxd is record
  valid_sc : std_logic; -- CX and D value for SC coding valid status
  cx_sc    : context;
  d_sc     : std_logic;
  valid_zc : std_logic; -- CX and D value for ZC coding valid status
  cx_zc    : context;
  d_zc     : std_logic;
  end record spp_cxd;

  -- one row of Code block
  type spp_out_row is array (31 downto 0) of spp_cxd;
  -- 32x32 Code block ouput array
  type spp_out is array (31 downto 0) of spp_out_row;

  -- CX and D output for each bit of bit-plane in MRP
  type mrp_cxd is record
  valid_mrc : std_logic; -- CX and D value for MRC coding valid status
  cx_mrc    : context;
  d_mrc     : std_logic;
  end record mrp_cxd;

  -- one row of Code block
  type mrp_out_row is array (31 downto 0) of mrp_cxd;
  -- 32x32 Code block ouput array
  type mrp_out is array (31 downto 0) of mrp_out_row;
```

```
  type neighbor is record          -- neighboring 8 pixels
    d0  : std_logic_vector(0 downto 0);     -- D0 V0 D1
    d1  : std_logic_vector(0 downto 0);     -- H0 X  H1
    d2  : std_logic_vector(0 downto 0);     -- D3 V1 D2
    d3  : std_logic_vector(0 downto 0);
    h0  : std_logic_vector(0 downto 0);
    h1  : std_logic_vector(0 downto 0);
    v0  : std_logic_vector(0 downto 0);
    v1  : std_logic_vector(0 downto 0);
  end record neighbor;

  type neighbor_conv is record          -- neighboring 8 pixels
    d0  : std_logic_vector(1 downto 0);  -- D0 V0 D1
    d1  : std_logic_vector(1 downto 0);  -- H0 X  H1
    d2  : std_logic_vector(1 downto 0);  -- D3 V1 D2
    d3  : std_logic_vector(1 downto 0);
    h0  : std_logic_vector(1 downto 0);
    h1  : std_logic_vector(1 downto 0);
    v0  : std_logic_vector(1 downto 0);
    v1  : std_logic_vector(1 downto 0);
  end record neighbor_conv;

-- chi neighbor values used in sign coding
  type chi is record
    nplus : std_logic;              -- chi[m, n-1]
    nminus: std_logic;              -- chi[m, n+1]
    mplus : std_logic;              -- chi[m-1, n]
    mminus : std_logic;             -- chi[m+1, n]
    mn     : std_logic;             -- chi[m, n]
  end record chi;

  -- sigma neighbor values used in sign coding
  type sigma is record
    nplus : std_logic;              -- sigma[m, n-1]
    nminus: std_logic;              -- sigma[m, n+1]
    mplus : std_logic;              -- sigma[m-1, n]
    mminus : std_logic;             -- sigma[m+1, n]
  end record sigma;

 -- chi neighbor values used in sign coding
  type chi_local is record
    nplus : std_logic_vector(1 downto 0);          -- chi[m, n-1]
    nminus: std_logic_vector(1 downto 0);          -- chi[m, n+1]
    mplus : std_logic_vector(1 downto 0);          -- chi[m-1, n]
    mminus : std_logic_vector(1 downto 0);         -- chi[m+1, n]
  end record chi_local;
```

```
-- sigma neighbor values used in sign coding
type sigma_local is record
  nplus : std_logic_vector(1 downto 0);          -- sigma[m, n-1]
  nminus: std_logic_vector(1 downto 0);          -- sigma[m, n+1]
  mplus : std_logic_vector(1 downto 0);          -- sigma[m-1, n]
  mminus : std_logic_vector(1 downto 0);         -- sigma[m+1, n]
end record sigma_local;

-- sigma neighbor values used in MRC coding
type sigma_mrc is record
  mminus_n       : std_logic_vector(0 downto 0);  -- sigma[m-1, n]
  mplus_n        : std_logic_vector(0 downto 0);  -- sigma[m+1, n]
  mminus_nminus : std_logic_vector(0 downto 0);  -- sigma[m-1, n-1]
  mminus_nplus   : std_logic_vector(0 downto 0);  -- sigma[m-1, n+1]
  mplus_nminus   : std_logic_vector(0 downto 0);  -- sigma[m+1, n-1]
  mplus_nplus    : std_logic_vector(0 downto 0);  -- sigma[m+1, n+1]
end record sigma_mrc;

-- Input 4 consecutive bits for RLC coding
type rlc_data is record
  one   : std_logic_vector(0 downto 0);   -- first bit
  two   : std_logic_vector(0 downto 0);   -- second bit
  three : std_logic_vector(0 downto 0);   -- third bit
  four  : std_logic_vector(0 downto 0);   -- fourth bit
end record rlc_data;

type cxd_pair is record -- used in RLC coding
  cx : std_logic_vector(4 downto 0);
  d  : std_logic;
end record cxd_pair;

-- (CX, D) data set output from RLC coding
type cxd_rlc is record
  first : cxd_pair;  -- (17,0/1), (18, 0/1) and (18, 0/1).
  second: cxd_pair;
  third : cxd_pair;
end record cxd_rlc;

-- 8 input neighbors of the bit being encoded
type sigma_pref is record
  one   : std_logic;   -- first neighbor
  two   : std_logic;   -- second neighbor
  three : std_logic;   -- third neighbor
  four  : std_logic;   -- fourth neighbor
  five  : std_logic;   -- fifth neighbor
  six   : std_logic;   -- sixth neighbor
  seven : std_logic;   -- seventh neighbor
```

```
  eight : std_logic;   -- eight neighbor
end record sigma_pref;

-- CX and D value for SC coding valid status
type cup_cxd is record
  valid_sc : std_logic;
  cx_sc    : context;
  d_sc     : std_logic;
  valid_zc : std_logic; --  ZC coding valid status
  cx_zc    : context;
  d_zc     : std_logic;
  valid_rlc: std_logic;
  first_rlc    : cxd_pair;  -- (17,0/1), (18, 0/1) and (18, 0/1).
  second_rlc   : cxd_pair;
  third_rlc    : cxd_pair;
end record cup_cxd;

-- one row of Code block
type cup_out_row is array (31 downto 0) of cup_cxd;
-- 32x32 Code block ouput array
type cup_out is array (31 downto 0) of cup_out_row;

type mem_port_in is record
 --clka   : std_logic;
      wea   : std_logic_vector(0 downto 0);
      addra : std_logic_vector(4 downto 0);
      dina  : std_logic_vector(31 downto 0);
end record mem_port_in;

type mem_port_out is record
      douta : std_logic_vector(31 downto 0);
end record mem_port_out;

type pixel_mem_port_in is record
 --clka   : std_logic;
      wea   : std_logic_vector(0 downto 0);
      addra : std_logic_vector(31 downto 0);
      dina  : std_logic_vector(31 downto 0);
end record pixel_mem_port_in;
```

-- BAC

```
-- index value used in BAC
subtype index is integer range 0 to 46;
```

```
    -- array of index-context for lut
    subtype lut_index is integer range 0 to 46;
    type lut_cxi is array (0 to 18) of lut_index;

    -- array of MPS-context for LUT
    type lut_mpscx is array (0 to 18) of std_logic;

    -- array for NMPS/CX LUT
    type lut_switch is array(0 to 46) of std_logic;

    subtype nmps_index is integer range 0 to 46;
    type lut_nmpscx is array(0 to 46) of nmps_index;

    subtype nlps_index is integer range 0 to 46;
    type lut_nlpscx is array(0 to 46) of nlps_index;

    subtype qe is std_logic_vector (15 downto 0);
    type lut_qeicx is array(0 to 46) of qe;

end package types;

package body types is
end package body types;
```

B. Profiling Images

Figure 42: Bretagne1.ppm

Figure 43: Bretagne2.ppm

Figure 44: Cevennes1.ppm

Figure 45: bike.pgm

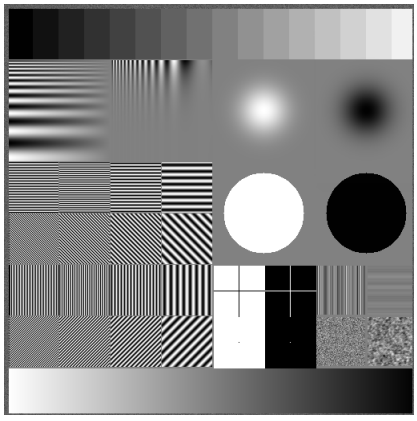

Figure 46: target.pgm

Figure 47: lena.ppm

Figure 48: lena.pgm

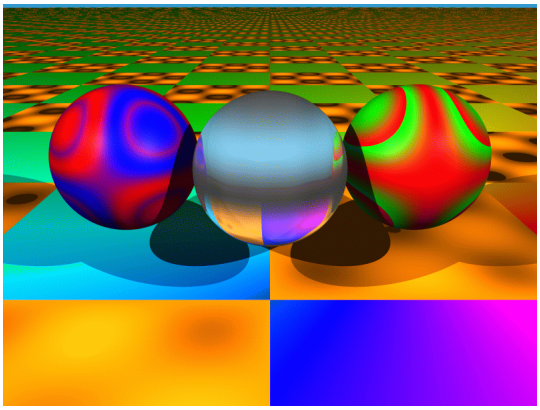

Figure 49: cwheel.ppm

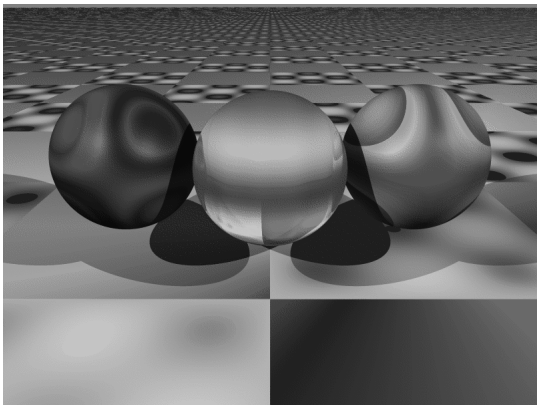

Figure 50: cwheel.pgm

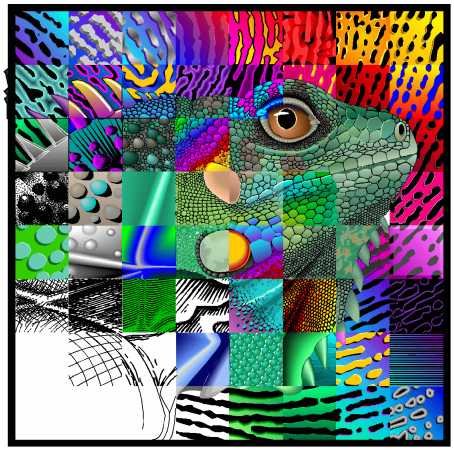

Figure 51: frymire.ppm

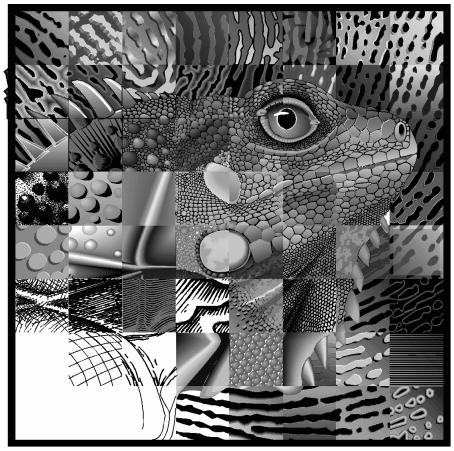

Figure 52: frymire.pgm

C. Pseudo Code of BAC

```
Initialization ()
{
        A = 0x00008000;
        C = 0x00000000;
        BP = BPST - 1;
        CT = 12;
        if ( B == 0xFF CT = 13;
        reset I(CX) and MPS(CX) with  their initial values.
}

CodeMPS ( )
{
       qe = Qe(I(CX));
       A = A - qe;       /* new subinterval for MPS */
       if ( A < 0x8000 ) {
         if ( A < qe )   /* condition exchange */
           A = qe;
         else
           C = C + qe;
         /* choose next index for MPS */
         I (CX) = NMPS(I(CX));

         call RenormalizationENC();
       }
       else
         C = C + qe;
}
```

```
CodeLPS( )
{
      qe = Qe(I(CX));
      A  = A - qe; /* new subinterval for MPS */

      if ( A >= qe )
        A = qe;     /* C is left unchanged */
      else          /* conditional exchange */
        C = C + qe;

          if (switch(I(CX)) == 1)
        /* change the sense of MPS(CX) */
        MPS(CX) = 1 - MPS(CX);

      /* choose next index for LPS */
      I(CX) = NLPS (I(CX)) ;

      call RenormalizationENC() ;
}


RenormalizationENC()
{
      do
      {
        A = A << 1; /* left shift 1 bit */
        C = C << 1; /* left shift 1 bit */
        CT = CT -1;
        if ( CT == 0 ) call ByteOut();

      } while ( A < 0x8000
}



ByteOut ()
{
      if ( B == 0xFF ) call bit_Stuffing();
      else {
        if ( C < 0x08000000 ) /* no carry bit */
          call no_bit_Stuffing();
        else {
          B = B + 1; /* add carry bit to B */
          if ( B == 0xFF ){
            C = C & 0x07FFFFFF;
             call bit_Stuffing();
          }
```

```
          else
            call no_bit_Stuffing();
        }
    }
}


bit_Stuffing()
{

     BP = BP + 1;  /* output B*/
     B = C >> 20;  /* "cbbb bbbb" bits of C */
     C = C & 0x000FFFFF;
     CT = 7;
}


no_bit_Stuffing()
{

     BP = BP + 1; /* output B */
     B = C >> 19; /* "bbb bbbb b" bits of C */
     C = C & 0x0007FFFF;
     CT = 8;
}


FLUSHregister()
{
     TempC = C + A ;
     C = C | 0x0000FFFF;
     if ( C >= TempC) C = C - 0x00008000;

     C = C << CT;
     call ByteOut();
     C = C << CT;
     call ByteOut();

     if ( B == 0xFF)
       discard B;
     else
       BP = BP + 1;  /* output B */
}
```

References

[1] E. Majani, "JPEG2000 Part I Final Committee Draft Version 1.0," *ISO/IEC JTC1/SC29/WG1/N1646R*, vol. 1.0, no. March, 2000.

[2] T. Acharya and P.-S. Tsai, *JPEG2000 Standard for Image Compression:Concepts, Algorithms and VLSI Architectures*. John Wiley & Sons, 2005.

[3] "Logicore ip block memory generator v4.3 product specification, www.xilinx.com," June 2011.

[4] D. Taubman, "High performance scalable image compression with EBCOT.," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 9, pp. 1158–70, Jan. 2000.

[5] F. De Simone, M. Ouaret, F. Dufaux, A. G. Tescher, and T. Ebrahimi, "A comparative study of JPEG2000, AVC/H.264, and HD photo," *Proceedings of SPIE*, vol. 6696, pp. 669602–669602–12, 2007.

[6] T. D. Tran, L. Liu, and P. Topiwala, "Performance comparison of leading image codecs: H.264/AVC Intra, JPEG2000, and Microsoft HD Photo," *Proceedings of SPIE*, vol. 6696, pp. 66960B–66960B–14, 2007.

[7] D. Marpe, "Performance evaluation of Motion-JPEG2000 in comparison with H.264/AVC operated in pure intracoding mode," *Proceedings of SPIE*, vol. 5266, pp. 129–137, 2004.

[8] D. S. Cruz and T. Ebrahimi, "A study of JPEG2000 still image coding versus other standards," *ISO/IEC JTC1/SC29/WG1/N1814*, July 2000.

[9] "Lossy/lossless coding of bi-level images," *ISO/IEC 14492-1*, 2000.

[10] "Openjpeg, http://www.openjpeg.org/," November 2011.

[11] "Xilinx ise 13.2 user guide, http://www.xilinx.com/support/documentation/dt_ise13-2_userguides.htm," June 2011.

[12] "Xilinx university program xupv5-lx110t development system, http://www.xilinx.com/univ/xupv5-lx110t.htm," June 2011.

[13] K. Andra, T. Acharya, and C. Chakraborti, "A High Performance JPEG2000 Architecture," *Proc. of the IEEE Intl, Symposium on Circuits and Systems(ISCAS 2002)*, pp. 765–768, May 2002.

[14] K. Andra, T. Acharya, and C. Chakraborti, "A High Performance JPEG2000 Architecture," *IEEE Intl,Transactions of Circuits and Systems for Video Technology*, vol. 13, pp. 209–218, March 2003.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Mahesh Krishnappa)