EFFICIENT VLSI IMPLEMENTATION OF BIT PLANE CODER OF JPEG2000

Kishore Andra[*], Tinku Acharya[*,+] and Chaitali Chakrabarti[*]
[*] Arizona State University, Tempe, Arizona, 85287, USA
[+] Intel Corporation, Chandler, Arizona, 85226, USA

ABSTRACT

To overcome many drawbacks in the current JPEG standard for still image compression, a new standard, JPEG2000, is under development by the International Standard Organization. Embedded bit plane coding is the heart of the JPEG2000 encoder. This encoder is more complex and has significantly higher computational requirements compared to the entropy encoding in current JPEG standard. Because of the inherent bit-wise processing of the entropy encoder in JPEG2000, memory traffic is a substantial component in software implementation. However, in hardware implementation, the lookup tables can be mapped to logic gates and memory accesses for the state bit computation can be reduced significantly by careful design. In this paper, we present an efficient VLSI architecture for embedded bit-plane coding in JPEG2000 that reduces the number of memory accesses. To better understand the interaction of this architecture with the rest of the coder, we also present a system level architecture for efficient implementation of JPEG2000 in hardware.

Keywords : JPEG2000, bit plane coding, EBCOT architecture.

1. INTRODUCTION

The differences in the computing power, bandwidth and memory of wireless and wired devices, as well as emergence of diverse imaging application requirements in this Internet age, have made resolution scalability and quality scalability essential in today's still image compression standards. Although these properties can be attained with present JPEG, they cannot be achieved in a single bit stream [1]. To overcome these drawbacks, the upcoming still image compression standard JPEG2000 has been designed [2]. Error resilience, manipulation of images in compressed domain, region of interest coding, non-iterative rate control etc., are some of the other important features of the JPEG2000 standard. All these features are possible due to adaptation of the Discrete Wavelet Transform (DWT), and intra-subband entropy coding along the bit planes using a combination of Bit Plane Coder (BPC) and a Binary Arithmetic Coder (BAC) in the core algorithm. The BPC is proposed to be performed using *Embedded Block coding with Optimized Truncation* (EBCOT) algorithm [3], while BAC is proposed to be performed using the MQ coder [2]. All three core blocks, namely, the DWT, BPC and BAC blocks are the computational and memory intensive blocks of the JPEG2000 standard. Of these blocks, DWT is very symmetrical in nature and can be handled either by dedicated hardware [5][6], DSP processors or even general purpose processors. In contrast, both BPC and BAC are very much control intensive and have to be performed in a sequential fashion. Furthermore, memory accesses are substantial in EBCOT implementation of BPC. The above reasons led us to conclude that specialized hardware implementations of BPC and BAC are required for a high performance JPEG2000 kernel.

The system architecture for JPEG2000 kernel, as shown in Fig.1, primarily consists of three modules: the DWT module, the BPC module and the BAC module. The modules interface with each other via memory and buffers. This paper briefly describes the system architecture but focuses on the architecture of the bit plane coder which implements the EBCOT algorithm. The EBCOT algorithm performs the bit plane coding in 3 non-overlapping passes (i.e. each bit is coded only once in one of the 3 passes) and generates the context and data bit pair, which is fed to the BAC coder. The context and data bit pair is formed using 4 state bits and the magnitude bit of the bit position being coded. The proposed architecture consists of combinational logic blocks that map the look up tables required to form the contexts from the state bits and memory blocks to hold the state bits. The architecture also consists of specialized registers to help in processing the state bits. The state bits of the 8 neighbors of the bit position being coded are required by the algorithm for context modeling. We have designed the registers in such a way that the memory interactions to fetch and write back the state bits are minimized. Also, the boundary conditions are handled in an innovative way. All the blocks are controlled by a 24 state controller state machine.

The rest of the paper is organized as follows. Section 2 gives a brief description of the proposed system level architecture for JPEG2000. The lifting based DWT architecture is explained in section 3. Section 4 contains an introduction to the EBCOT algorithm followed by details of the proposed architecture for the BPC coder. Section 5 briefly describes the MQ coder architecture. Section 6 concludes the paper.

2. JPEG2000 SYSTEM ARCHITECTURE

In JPEG2000, the image is broken up into tiles and coding is performed on the individual tiles. The first step of the coding is implementation of Discrete Wavelet Transform (DWT) where three high frequency subbands (HL, LH, HH) are generated at each level. All 4 subbands (one low frequency subband LL and three high frequency subbands HL, LH, HH) are generated at the last level. Quantization, if required, is applied on each of the DWT subbands. Each subband is then divided into rectangular structures called code blocks. The code blocks are coded individually by entropy coders consisting of the Bit Plane Coder (BPC) and the Binary Arithmetic Coder (BAC). The BPC, is implemented using the EBCOT algorithm [3] and generates the context and data pair from the neighborhood of the bit position being coded. The BAC, proposed to be carried out using the MQ coder [2], uses the context and data pair and the statistics of the data previously coded to generate the code stream. A rate distortion measure for a fixed number of rates and size of the code stream of individual code blocks are collected for all the code blocks. Based on the available rate and quality required, a final bit stream is formed using "layers" which contain contributions from the individual code blocks. The bit stream is scalable both in resolution and quality. In other words, a single bit stream can be used to generate both a complete image at the highest resolution and a part of the image at the lowest resolution.

We propose an architecture capable of performing the coding process described above. The input to the architecture is an image tile and the outputs are three code streams (one for each subband). The division of the image into tiles and formation of the layers at the end of coding process are handled by software. The block diagram of the proposed architecture is shown in Fig.1.

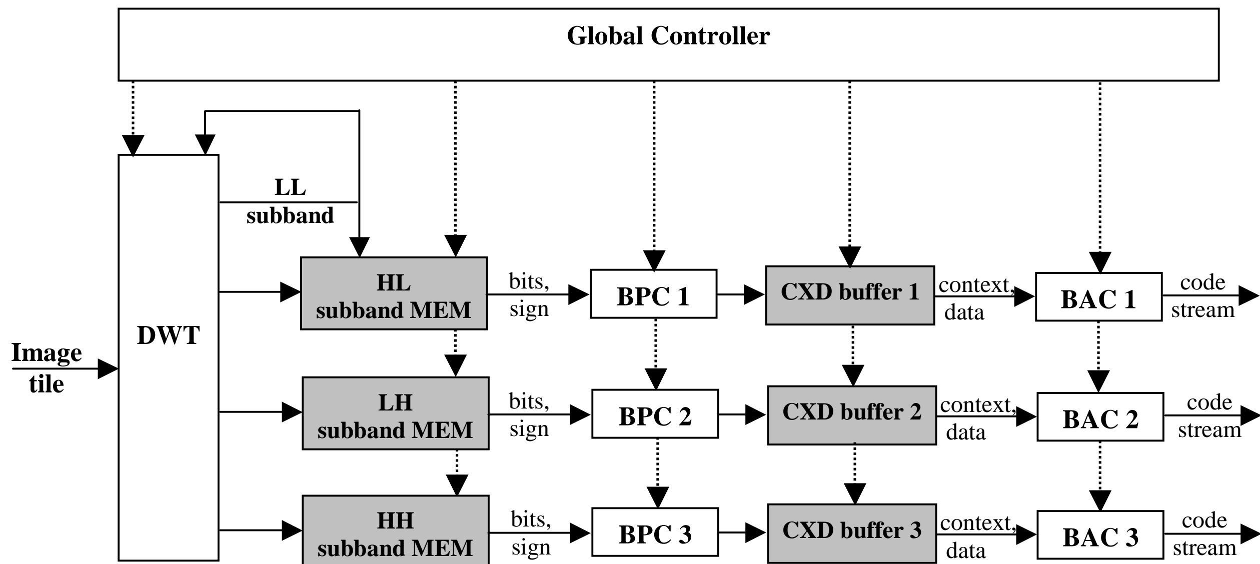

Fig.1. Proposed architecture for JPEG2000 encoder

The architecture consists of a DWT coder and three pairs of BPC and BAC coders. It also consists of six memory blocks: three subband MEM blocks between the DWT coder and three BPC coders and three CXD buffers between the BPC and BAC coders. A global controller is required to control the interactions between all these blocks. The data flow of the architecture is as follows. DWT is applied on the image tile to generate the three subbands at each level. The LL subband data is written back to the DWT block for the next level of decomposition. The data from the code blocks (formed from the quantized subband data) is written into subband MEMs. Each BPC reads the data from the corresponding subband MEM and writes the context-data pairs into the corresponding CXD

buffer. BAC reads from the CXD buffer and generates the code stream for each code block. At the last level, the LL subband is entropy coded using the HL entropy coder pair.

Three sets of BPC and BAC pairs are needed to handle the large number of computations during entropy coding. For instance to code a NxN code block, with one bit position being coded in each cycle, NxNx16x3 cycles are required. This is because the internal precision is 16 bits for lossless performance [6] and BPC performs the coding in 3 passes [3]. On top of this, the BAC requires at least two table lookups and two additions per bit [2]. Note that the number of computations can be significantly reduced if the by-pass mode proposed in [2] is used, wherein after the 4th bit plane the output of BPC in two of the passes is directly sent out to the bit stream by-passing the BAC. Fortunately, the entropy coding process can be easily implemented in parallel as the code blocks are entropy coded independently. So we chose to have three sets of coders to perform the entropy coding of the subbands in parallel. Further parallelization is possible in coding individual code blocks. But this level of parallelization does not help much when the size of the code blocks becomes small as in higher level computation of DWT.

The decoder architecture is similar to the encoder architecture with data flow in the opposite direction. The CXD buffer is replaced by a single register to hold the context. This is because the bit plane decoding can not proceed before the data is obtained from the binary arithmetic decoder. The by-pass mode significantly helps in the accelerating the decoding process.

3. LIFTING BASED DWT

The Discrete Wavelet Transform (DWT) is implemented using the lifting scheme in JPEG2000. In the lifting scheme [7], the filter implementation is mapped into banded matrix multiplications. The advantages of this scheme include in-place computation, fewer computations compared to the convolution based implementation, symmetric inverse etc. Two schemes are possible based on whether the low pass terms or the high pass terms are calculated first. Fig. 2 describes the latter scheme. Here, in the first step, low-pass samples (even terms) are multiplied by the time domain equivalent of $\tilde{t}(z)$, and are added to the high pass samples (odd terms). In the second step, the updated high pass samples are multiplied by the time domain equivalent of $\tilde{s}(z)$ and are added to the low-pass samples. The inverse is obtained by traversing in the reverse direction; changing the K_1 *and* K_2 factors to $1/K_1$ *and* $1/K_2$, and reversing the signs of coefficients in $\tilde{t}(z)$ and $\tilde{s}(z)$.

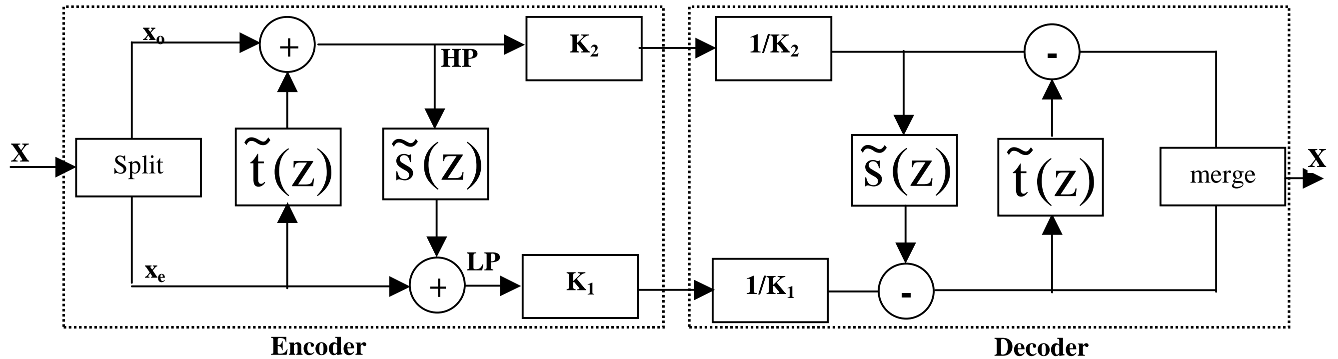

Fig. 2. Lifting based DWT scheme

A specialized architecture for implementing the wavelet filters (5,3), (9,7), S(13,7), C(13,7), (2,6), (2,10) and (6,10), has been proposed in an earlier paper by the authors [5]. The architecture consists of four processors and two memory modules each consisting of four banks. Each processor consists of two adders, a shifter and a multiplier. Each bank in the first module is of size N$\times\lceil N/2 \rceil$, where as each bank in the second module is of size N to 4N based on the filter, where N is the size of a row. The architecture generates two samples (from two different subbands) each cycle. However for JPEG2000, the DWT coder does not need to generate outputs at such high rates since the BPC will not be able to absorb the generated data at the same rate. It would be more efficient to have one processor to perform the DWT at a slower rate (compared to the architecture in [5]). Also, a single processor architecture with single memory module is better suited for in-place computation.

4. BIT PLANE CODER

4.1 EBCOT Algorithm

The bit plane coder is implemented using the EBCOT algorithm as explained in [2][3]. It is summarized here for the sake of completeness. Each bit plane is coded in 3 passes: *Significance Pass* (SP), *Magnitude Refinement Pass* (MRP) and *Clean up Pass* (CP). In each pass, only a part of the bit plane is coded and each bit position is coded only once by one of the three passes. The BPC works on strips of 4 elements along the rows as shown in Fig.3. The code block scan is carried from left to right. Two modes of coding ("*Regular*" and "*Vertical Causal*") as shown in Fig.3. are possible. The proposed architecture assumes *Vertical Causal* (VC) mode though the *Regular* mode can easily be supported at the expense of extra memory.

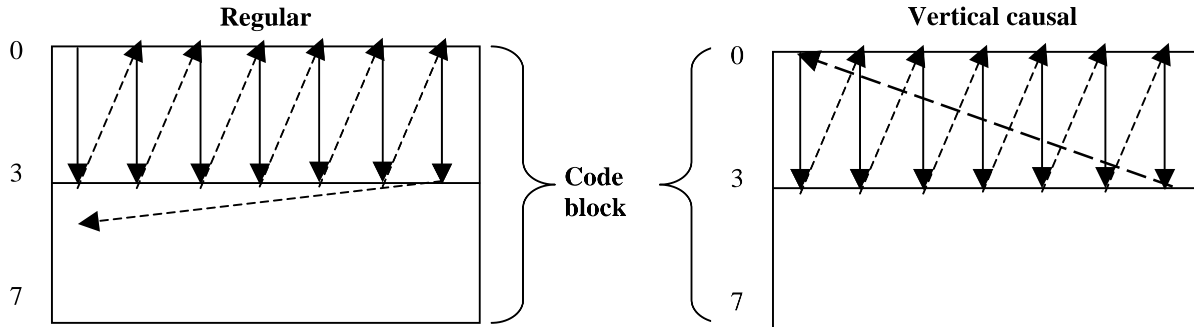

Fig.3. Coding modes

The BPC requires 4 state information bits and 1 magnitude bit (υ) for each bit position. The state information bits determine in which pass each bit is coded and also help in the generation of context and data. The 4 state information bits are as follows:

Significance bit (σ) – This bit is set whenever the magnitude bit of the corresponding subband coefficient is '1' for the first time.

Visited once bit (η) - This bit is set when the bit is coded in a pass.

Magnitude refinement coded bit (σ') - This bit is set the first time the magnitude refinement primitive (explained below) is used.

Sign bit (χ) – This bit is 0 for positive numbers and 1 for negative numbers and is obtained from the sign-magnitude representation of the subband values.

All the state bits except for η bits are maintained across all the bit planes. The η bits are reset at the end of each bit plane. It should be noted that, σ and χ for the neighbors that are outside the strip are assumed to be 0. All the three passes make use of one or more of the following four primitives – *Zero Coding*, *Sign Coding*, *Magnitude Refinement Coding* and *Run Length Coding*.

4.2 Primitives

Zero Coding (ZC) **-** The ZC uses 9 (0-8) out of possible 19 contexts. Context for the data in bit position X is formed from the 8 neighboring values *(D_0-D_3, H_0,H_1,V_0,V_1)* in the σ matrix as shown in Table 1. The data is the magnitude of the bit position X.

D_0	V_0	D_1
H_0	X	H_1
D_3	V_1	D_2

Table 1 : Neighborhood to consider to code the input bit at position X.

Sign Coding (SC) **-** The SC uses 5 contexts (contexts 9-13) and is a two step process. In the first step, the σ and χ of the horizontal and vertical neighbors are used to form the horizontal and vertical *'contributions'* and a '*xor*' bit [2][3]. In the second step, context is formed from the two contributions and data is formed by *exclusive OR* operation of the *sign bit* and the *xor bit*.

Magnitude Refinement Coding (MRC) - The MRC uses 3 contexts (contexts 14-16). The contexts are formed based on whether it is the first time the magnitude refinement is being used on a certain position and its 8 immediate neighbors. The data is the bit itself.

Run Length Coding (RLC)- The RLC uses the remaining 2 contexts (contexts 17-18). It is invoked only at the beginning of a strip if the σ of all the 8 neighbors is 0 for all the bits in a strip. If none of the bits in the strip become significant, context 17 with data = 0 is used. On the other hand, if any bit does become significant, context 17 with data = 1 is used. This is followed by MSB and LSB of Zero Index (ZI) (00-11) of the bit position which contains the '1' bit. Context 18 is used for ZI bits.

4.3 Coding passes

As mentioned earlier, each bit plane is coded in 3 passes. The first bit plane is coded just with the cleanup pass (CP). In the significance pass (SP), all the bits whose σ = 0 and who have at least one of the immediate 8 neighbors with σ = 1 are coded using ZC primitive. If the bit becomes significant, the SC primitive is used and σ of the bit being coded is set to 1. When ZC is applied, the corresponding η is set. In MRP, all the bits with corresponding η = 0 and σ = 1 are coded using MR primitive. The corresponding σ' bit is set to 1. In CP, if η = 0 and σ = 0 for the first element in the strip, the RLC condition is checked. If the RLC condition (mentioned in the RLC primitive) is satisfied, the RLC primitive is used. If one of the bits in the strip become significant, then SC is used and σ is set for that bit. This is followed by application of ZC + SC for the rest of the bits in the strip. If the RLC condition is not satisfied, then ZC + SC is used for all the elements with η = 0 and σ = 0. The σ and χ of the neighbors which are outside the boundaries of the code block are assumed to be zero in all the primitives.

4.4 Proposed VLSI Architecture

The block diagram of the proposed architecture for the EBCOT encoder is shown in Fig.4. The architecture consists of the following key building blocks - (i) three combinational logic blocks to determine the contexts for ZC, MRC and SC (the contexts for RLC are hard coded), (ii) five shift registers of varying sizes and functionality to store the state variables σ, η, σ', υ, χ, (iii) 5 memory blocks (for the magnitude bits and 4 corresponding state bits) each of size *N* x 4, where *N* is the number of columns in the block. In addition to these key building blocks, there is a multiplexer (MUX) to select the right context for the bit to be coded from the various contexts based on the coding pass. The architecture also contains a counter to keep track of the number of strips processed and also the coding *pass* being used. A controller is present to control the shift registers and the mux. It also generates the read and write signals for the memories. Functionality of these building blocks have been described below.

4.4.1 Combinational logic blocks

The tables provided in [2][3] to form the context for each of the primitives can be expressed in terms of simple logic operations. The combinational logic blocks consists of gates that map the state variables to a context.

ZC context block: The input to this block is σ of the eight neighbors of the bit being coded and magnitude of the bit position. The output is the ZC context and data pair. The logic equations for generating the context for the LL subband are listed below. The notation used is as follows: *xc0* is true if both the bits are zero, *xc11* is true if one or more bits are 1, *xc1* if only one bit is 1, *xc2* if both the bits are 1 and *xc22* if two or more bits are 1. The prefixes *h,v and d* are used for horizontal, vertical and diagonal neighbors respectively.

Logic equations -

$hc11 = h_1 + h_2$; $hc2 = h_1 * h_2$; $vc11 = v_1 + v_2$; $vc2 = v_1 * v_2$;
$dc11 = d_0 + d_1 + d_2 + d_3$; $dc22 = d_0*(d_1 + d_2 + d_3) + d_1*(d_2 + d_3) + d_2*d_3$;
$hc0 = not(hc11)$; $hc1 = not(hc2) * hc11$; $vc0 = not(vc11)$; $vc1 = not(vc2) * vc11$
$dc0 = not(dc11)$; $dc1 = not(dc11) * dc22$

Then the contexts (for LL subband) are formed as –
$cx8 = hc2$; $cx7 = hc1 *vc11$; $cx6 = hc1 * vc0 * dc11$; $cx5 = hc1 * vc0 * dc0$;
$cx4 = hc0 * vc2$; $cx3 = hc0 * vc1$; $cx2 = hc0 * vc0 * dc22$; $cx1 = hc0 * vc0 * dc1$;

The binary representation of the context is then formed as –
$cx(4) = 0$; $cx(3)=cx8$; $cx(2)= cx7 + cx6 + cx5 + cx4$;

cx(1)= cx7 + cx6 + cx3 + cx2;
cx(0)= cx7 + cx5 + cx3 + cx1;

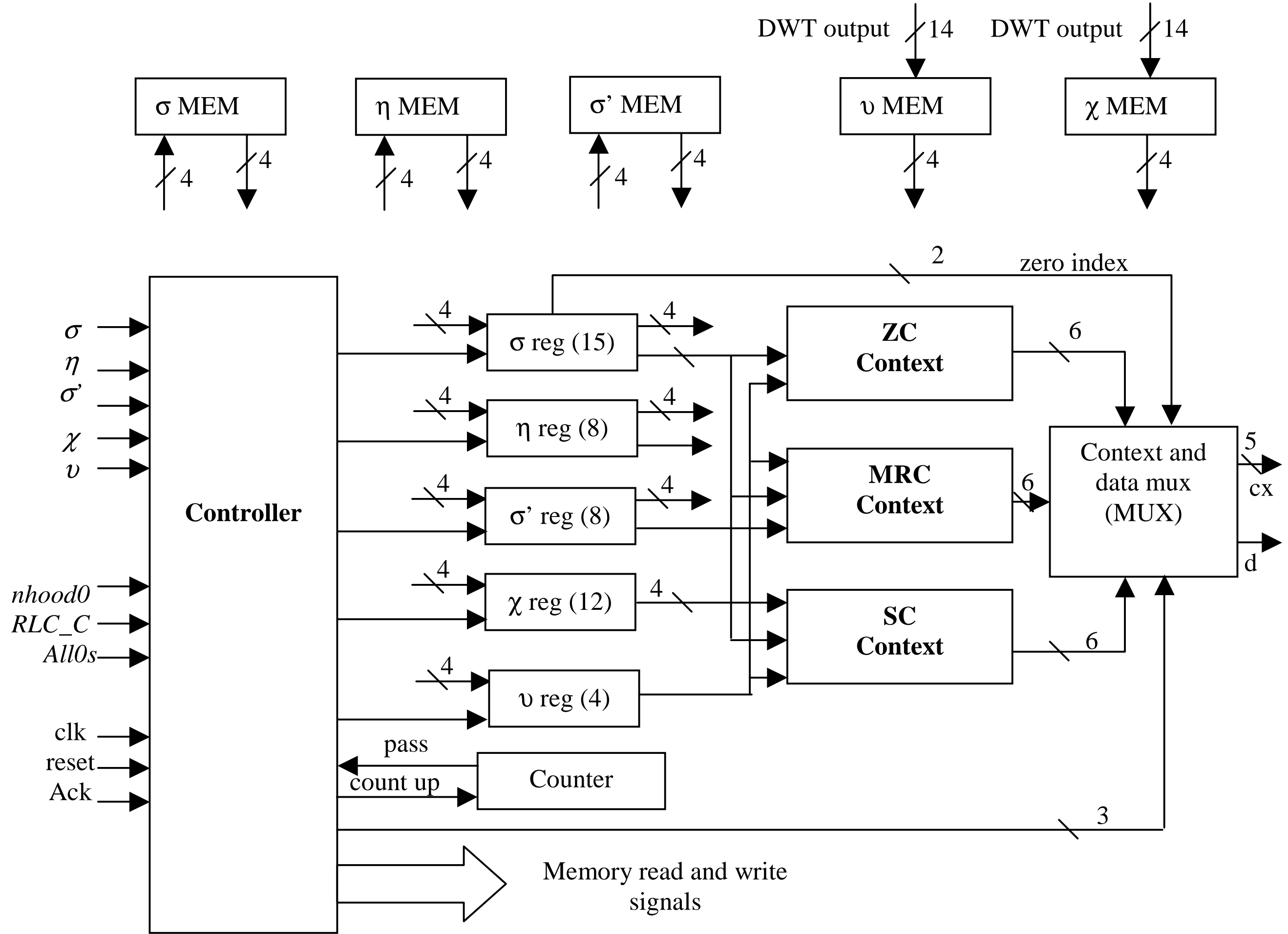

Fig.4. Proposed architecture for EBCOT encoder

Sign Coding context block: Inputs to this block are the σ (v_0, v_1, h_0, h_1) and χ (sv_0, sv_1, sh_0, sh_1) from σ and χ registers respectively. The output is the SC context and the sign data bit. A four gate delay is required to form the context in this case.

Magnitude refinement coding context block: The two inputs to this block are σ' and the *nhood0* bit (which indicates if the 8 neighbors are all 0's) from the σ register. The output is context and data pair for MRC.

Run length coding contexts: The 4 possible contexts are – RLC condition satisfied and strip is all 0's, RLC condition satisfied and the strip contains at least one '1' bit. The latter case is followed by context 18 and two bits of the Zero Index (supplied by the υ register) of the bit position that is '1'. These contexts and data pairs are hard coded.

4.4.2 Registers

There are five registers of varying sizes to store the state variables (see Fig.4). All the registers are capable of 1 bit left shift. For initialization and run length coding, the σ register and χ register are capable of 5 bit and 4 bit left shift respectively. The σ, η and σ' registers have an "update" position where a '1' is written to set the corresponding state variable, when required. Data from the (corresponding) memory is written into 4 least significant bit positions in the registers. But data can be read from different positions of the registers and written into memory. The registers are read and written at the end of coding of each strip. The order in which the bits of a 4x3 code block are coded is

shown in Table 2. The functionality of all the registers is explained based on this order. It is assumed that the bit in position ‘5’ is being coded.

1	5	9
2	6	10
3	7	11
4	8	12

Table 2 : Coding order for a 4x3 code block

σ register - This register is 15 bits wide and contains σ of 3 rows of data with data arranged as shown in Fig.5.

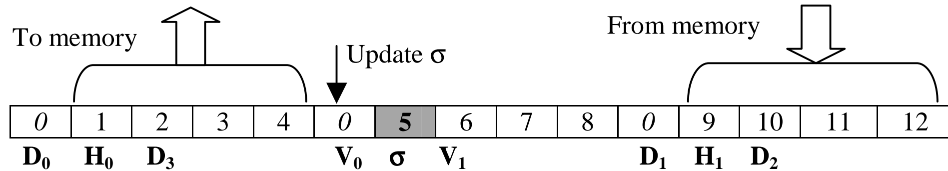

Fig.5. σ register

For ZC, the σ of 8 neighbors, around the bit being coded (X) is required (see Table 1). As mentioned earlier, the σ for the neighbors that are outside the block boundaries is assumed to be 0. So D_0, V_0 and D_1 for the first element of each strip (i.e. 1,5,9 in Table 2) are 0. This is achieved by introducing *‘0’* ahead of each strip of data as shown in Fig.5. This results in performing an extra shift at the end of coding of each strip to align the data properly. Similarly, D_1, H_1 and D_2 for the last element of each strip (4,8,12 in Table 2) are zero. The “update” bit is set during the shift operation to change the σ of a particular bit.

The boundary conditions due to the first strip and last strip can be handled by initialization and termination. During initialization, the register is reset and the first strip is written and shifted by 5 bits. Then the second strip is written. This results in proper alignment of data. For termination, no data is read when coding the last strip.

Two zero detectors are connected to the output of the σ register to generate the ‘*nhood0*’ signal which helps determine whether a bit is coded in SP, and the ‘*RLC_C*’ signal which helps to determine if RLC condition is satisfied in CP. The *nhood0* signal is generated if σ of the 8 neighbors is 0 and the *RLC_C* signal is generated if σ of all the elements in the 3 strips is 0. When the RLC condition is satisfied and the strip consists of all 0’s, then no further coding is required and the data is shifted left by 5 bits.

η,σ’ registers - Both the registers consist of 8 bits and the data is arranged as shown in Fig.6. The data from the corresponding memories is written to the 4 LSBs (assuming the right most bit is the LSB) and the updated data is read from the 4 MSBs if the data has to be written back. The “update” position is the 5th LSB as shown in Fig. 4. The boundary conditions do not affect both the registers as the neighboring η nor σ’ is required.

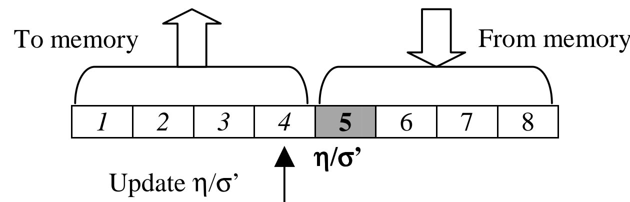

Fig.6. η/σ’ register

χ register - The χ register has 12 bits. The data arrangement is as shown in Fig. 7. The data arrangement and the boundary conditions for the χ register are similar to the σ register. But when a neighbor is insignificant, the sign of the neighbor does not play a role in forming the context. This property helps to have 12 bits unlike the special

arrangement required in the σ register. Also this saves the extra shift required at the end of coding a strip. The initialization, termination and RLC coding is similar to σ register except for that the data is shifted by 4 bits.

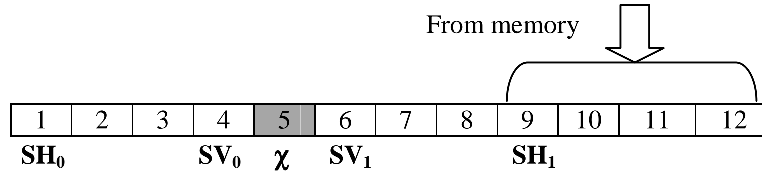

Fig. 7: χ register

υ register **-** It consists of 4 bits and is just a shift register with no additional functionality. A zero detector is attached to the output to generate the '*All0s*' signal which helps to determine what path the RLC primitive takes in CP. Also a 4 bit to 2 bit encoder is present to generate the ZI of the position of the first '1' bit in strip (with 1st position encoded as 00 and 4th position encoded as 11) for the RLC.

4.4.3 Memory blocks

Five memory blocks each of size Nx4 are used to store the state variables and the magnitude bits of one bit plane. The υ MEM and χ MEM are written from the DWT coder, bit plane by bit plane. The other 3 memories are written by the corresponding internal registers. All the memories have a single read and write port as shown in Fig.4.

4.4.4 Context and data mux

The multiplexer chooses the context from the outputs of ZC context block, SC context block, MR context block or the hard coded RLC contexts (17-18). The data bit is chosen from the υ, sign data, χ, hard coded RLC data bits (0,1) or the ZI (MSB, LSB) bits. The mux is controlled with a 3 bit word (*cntrl_cx*). Based on the pass being performed, the controller generates the control word. The contexts and data for different values of cntrl_cx are given in Table 3.

cntrl_cx	context	Data
000	-	-
001	ZC cx	υ
010	SC cx	sign data bit
011	MC cx	υ
100	17	0
101	17	1
110	18	ZI [MSB]
111	18	ZI [LSB]

Table 3: Output of the context and data mux for different values of cntrl_cx

4.4.5 Counter

It keeps tracks of the element in the strip being coded, the number of strips coded in each pass, the pass being processed and the bit plane being processed. This information is required for the state machine.

4.4.6 Controller

The state machine consists of 24 states. The state machine can be divided into 5 phases – initialization phase, ZC and SC primitive, MRC primitive, RLC primitive and termination phase. Each phase is explained below. It should be noted that states are numbered continuously across the phases.

Initialization phase (Fig.8) **–** This is invoked at the beginning of each bit plane. It reads the data from DWT coder to υ memory. It also reads data to χ memory if the first bit plane is to be coded (state 0). All the registers and memory address pointers are reset (state 1). The first strip is read into σ and χ registers and a *Ishift* (initialization shift, 5 bit left shift for σ and 4 bit left shift for χ) is performed to align the data (state 3). It should be noted that, for the first bit plane the σ memory is not accessed as data wouldn't be available yet. Then based on the pass that needs to be performed (supplied by the counter), one of the three paths is chosen.

ZC and SC primitive (Fig.9) **–** This primitive is invoked during the SP and during the CP when RLC condition is not satisfied or when RLC condition is satisfied and the strip contains a '1' bit. The primitive is started by reading the σ, χ and υ memories and writing the data into corresponding registers (state 4). If pass is SP, and if $\sigma = 0$ and σ of one of the eight neighbors is 1, then ZC context and data are written out (state 6), otherwise the bit is not coded in the SP pass. If the SP condition is satisfied and the bit becomes significant (i.e. is $\upsilon = 1$), then the SC context and data are written out (state 7) and σ and η are set (state 8). If $\upsilon = 0$, then only η is set. In all the cases, at the end of coding, the registers are shifted left by 1 bit and the counter is incremented (states 5,8,9). The counter output is checked at the end of coding each bit, to see if the strip is finished (end of row signal). If it is finished, then the termination phase is invoked (state 10) else the coding continues (states 6,5).

MRC primitive (Fig.10) **–** This is invoked during the MRP. The primitive is started by reading the σ, η, σ' and υ memories and writing the data to the corresponding registers (state 12). If the $\sigma = 1$ and $\eta = 0$ for a particular bit, it is coded in this primitive. The MRC context and data are written out and the corresponding σ' is set (states 14,15). At the end, the registers are shifted and counter is updated (states 5,15). The termination condition is checked; if it is not true, the coding continues (states 14,5) else the termination phase is initialized (state 10).

RLC primitive (Fig.11) **–** This phase is invoked during the CP and only if the first bit of the strip is being coded. The primitive is initialized by reading σ, η, χ and υ memories and writing into corresponding registers (state 16). If $\sigma = 0$ and $\eta = 0$ for the first bit in the strip, then the RLC condition is checked otherwise the bit is not coded in the pass (state 5). The rest of the bits are checked for the CP condition and are coded using ZC+SC primitives.

If RLC condition is not satisfied then ZC primitive is invoked (state 6). If the condition is satisfied, then it is determined if there is a '1' in the strip. If all the bits in the strip are 0, then context =17 and data = 0 are written out, the *Ishift* is performed (state 17), and the termination process is invoked (state 11). On the other hand if there is a '1' in the strip then context =17 and data =1 is written out (state 18). This is followed by context =18 and MSB and LSB of ZI (states 19,20). Also the signal *RLC_cb* is set (state 19) to indicate that ZC+SC needs to be applied on the rest of the bits in the strip without checking for the CP condition. Based on the ZI, required number of shifts are performed on the registers, and counter is incremented and SC is invoked (state 7).

Termination Phase (Fig. 12) – This phase is invoked at the end of each strip (based on end of row signal from the counter). The extra shift required by σ register is performed and the appropriate data from the registers is written to the memories based on the pass being performed (state 11). If it is the end of the block and required number of bit planes are coded, then coding stops (stop), otherwise if the current pass is CP, the next bit plane is read in and the coding starts with SP (state 0). If the current pass is either SP or MRP, then the next pass is started on the same bit plane (state 1). If end of the block is not reached, then based on the current pass, the coding starts on the next strip (state 11).

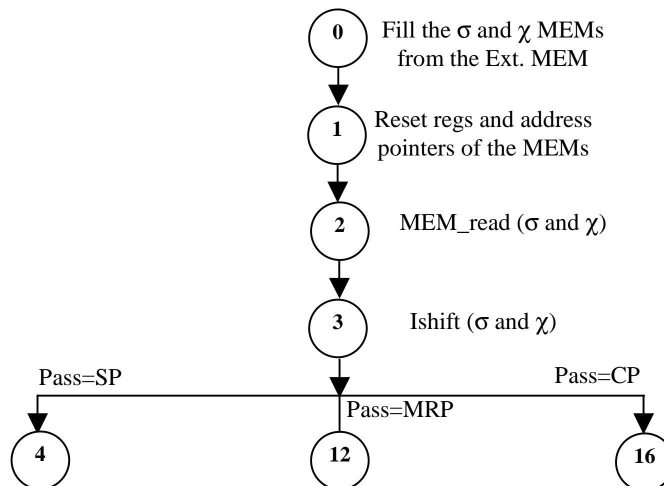

Fig.8. Initialization Phase

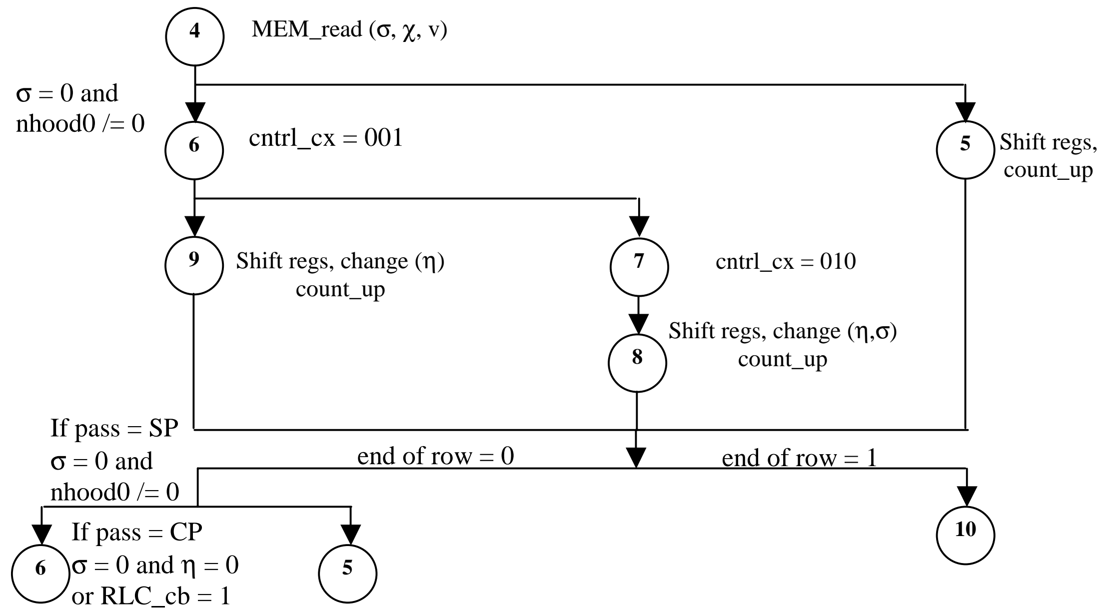

Fig.9. ZC and SC primitive

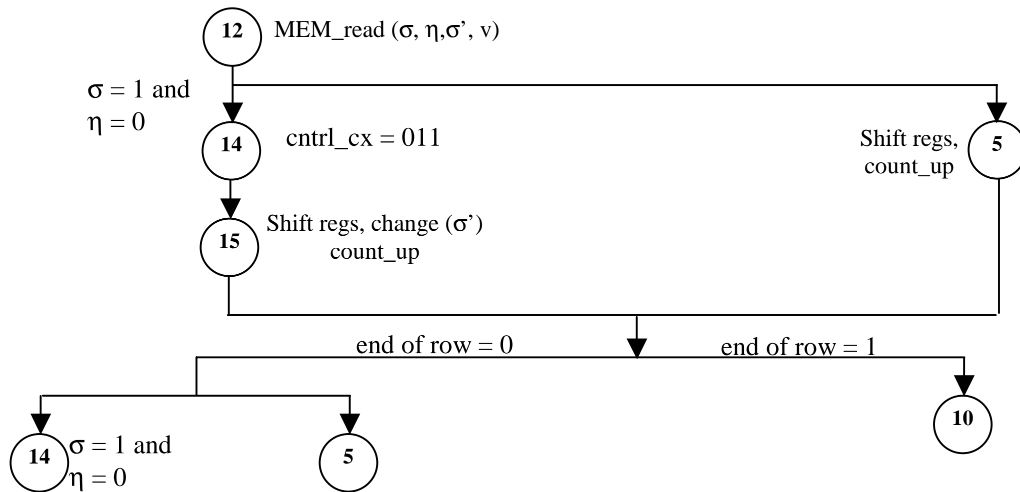

Fig.10. MRC primitive

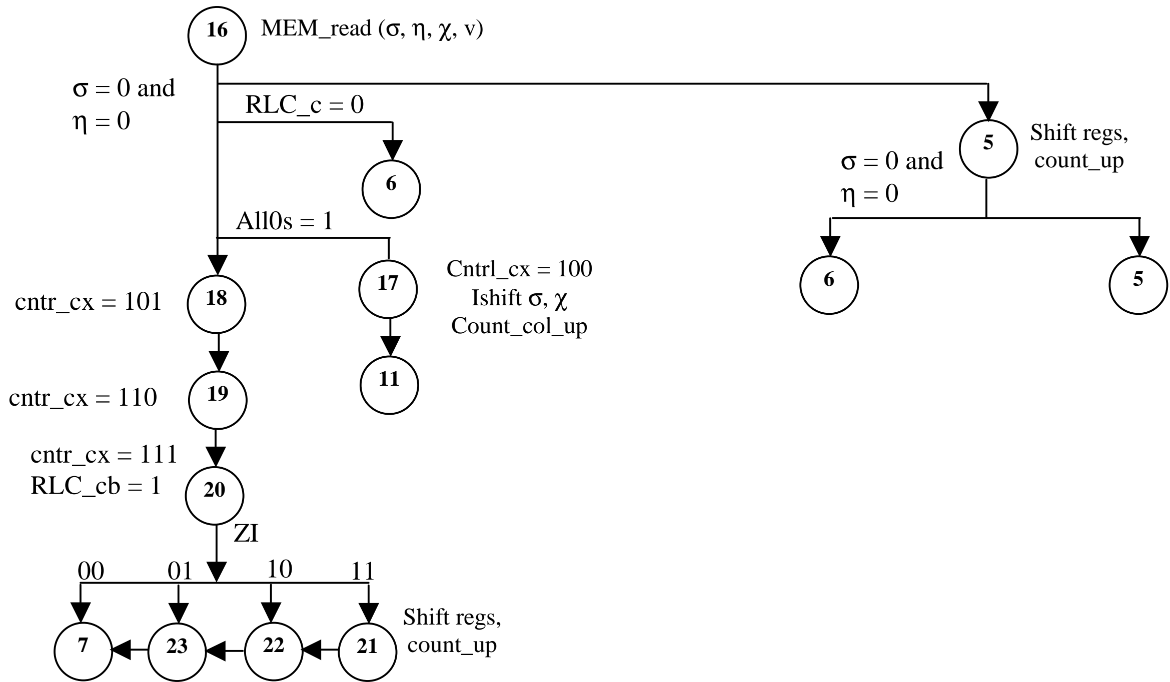

Fig.11. RLC primitive

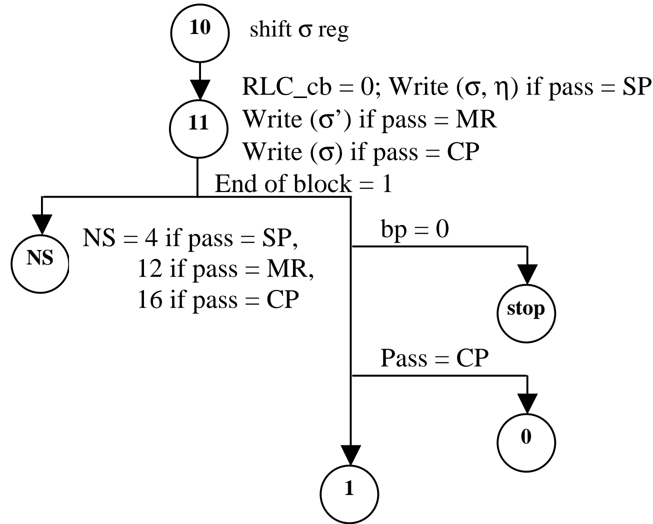

Fig. 12. Termination phase

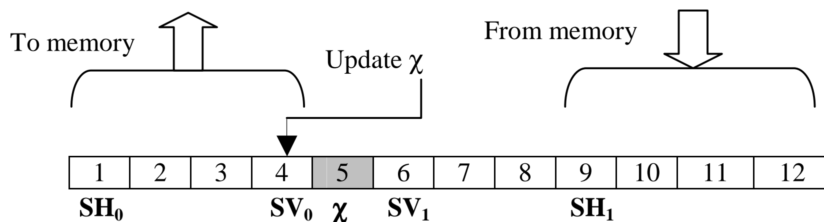

Fig.13. χ register for decoder

4.5 EBCOT Decoder

The architecture for the decoder remains almost the same as the encoder except for small changes to υ and χ memories and registers as shown in Fig.13. For instance, data from υ and χ MEMs is written out. Also, while in the encoder, both the bits of Zero Index are known before RLC is started, in the decoder, the bits are obtained one at a time. The resulting state machine is slightly different though the number of states required still remains the same.

5. MQ CODER

In JPEG2000, the binary arithmetic coding is to be performed using MQ coder [2]. This MQ coder is a multiplication free coder and is a derivative of the Q coder [8]. The architecture for the MQ coder is control-intensive like the EBCOT coder. The statistics are maintained using two look-up tables. The input context forms the index to the first table which contains the index to the second table as its data. The second table contains normalized probability values. The coding is performed based on this probability value, the input data symbol and a symbol called the *Most Probable Symbol* associated with each allowed context. The MQ coder encoder and decoder are not symmetric [2]. As a result, the control state machine for the encoder requires 32 states where as the decoder requires 25 states. This is because of a process called "bit stuffing", used to prevent a carry generated by the addition process from propagating thorough the code already sent out, and a process called "flush" used to terminate the coding properly so that decoder has enough information to decode all the bits, being present only in the encoder [2]. The datapath of both the architectures, which consists of a 16 bit adder and few specialized registers, is the same.

6. CONCLUSION

In this paper, we have briefly described a system level architecture for JPEG2000 and have focused on the hardware implementation of the bit plane coder. This was necessary since the encoder is more complex and requires significantly higher number of computations compared to the entropy encoder in current JPEG. Because of the inherent bit-wise processing of the entropy encoder in JPEG2000, memory traffic is substantial in software implementations. In this paper, we have proposed an efficient hardware implementation of the bit plane coding algorithms that requires fewer memory accesses. We have shown how lookup tables can be mapped to logic gates and how memory accesses during the computation of state bits can be reduced by careful design. We have developed and synthesized the VHDL models for the BPC and BAC coder pair. The preliminary gate counts (in 2 input NAND gate equivalents) not including the memories are – EBCOT encoder/decoder ~ 2000 gates, MQ encoder ~ 3200 gates and MQ decoder ~ 2500 gates. So a pair of BPC and BAC coders would approximately require 6000 gates. Thus an architecture with three sets of BPC-BAC coder pairs supports the high computational rate with minimal increase in hardware.

REFERENCES

1. J. L. Mitchell and W. B. Pennebaker, "JPEG still image data compression standard," Van Nostrand Reinhold, 1993.
2. JPEG2000 Final Committee Draft (FCD) (http://www.jpeg.org/JPEG2000.htm).
3. D. Taubman, " High performance scalable image compression with EBCOT", IEEE Transactions on Image Processing, vol.9, 1158-1170, July 2000.
4. ISO/IEC JTC 1/SC 29/WG 1 WG1N1878, JPEG 2000 Verification Model 8.5 (Technical description), September 13, 2000.
5. K. Andra, C. Chakrabarti and T. Acharya, "A VLSI architecture for lifting based wavelet transform", SiPS 2000, 70 –79.
6. K. Andra, C. Chakrabarti and T. Acharya, "An efficient implementation of a set of lifting based wavelet filters", ICASSP 2001,
7. I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting schemes", The Journal of Fourier Analysis and Applications, vol. 4, 247-269, 1998.
8. J. L. Mitchell and W. B. Pennebaker, "Software implementations of the Q-coder," IBM J. of Res. Develop, vol. 32, No. 6, pp. 753- 774, Nov. 1988.