

VLSI Algorithms and Architectures for JPEG2000

Tinku Acharya

Abstract

In this paper, we presented VLSI algorithms and architectures for JPEG2000. JPEG2000 is the new standard for image compression. We briefly described the core algorithms for JPEG2000 standard and its several features desirable in many interactive multimedia applications. The core compression algorithm has three major components – *discrete wavelet transform* (DWT), *fractional bit plane coding* (BPC), and *context adaptive binary arithmetic coding*. DWT and BPC are very computationally, as well as memory expensive operations. As a result, special purpose VLSI implementations of these algorithms are desirable for many devices to compress large size images in real time. Traditionally, the DWT is realized by convolution based finite impulse response (FIR) filtering techniques. However, Lifting based implementation of DWT is found to be computationally efficient and suitable for VLSI implementations. The basic principle behind the lifting based scheme is to decompose the finite impulse response filters in wavelet transform into a finite sequence of simple filtering steps. Lifting based DWT implementation have been recommended in JPEG2000 standard. Consequently, this has become an area of active research and several architectures have been proposed in recent years. In this paper, we reviewed some of the key lifting architectures suitable for VLSI implementation. The *embedded block coding with optimized truncation* (EBCOT) algorithm has been adopted for computation of BPC in JPEG2000. This algorithm is complex and inefficient to implement in a general purpose machine. We have described a special purpose architecture suitable for VLSI implementation of EBCOT algorithm. We also reviewed some elegant architectures in the literature which exploit the underlying data and computational parallelism inherent in the EBCOT algorithms. We also presented a top-level systems architecture for VLSI implementation of the JPEG2000 standard.

Keywords - Architecture, Discrete Wavelet Transform, JPEG2000, EBCOT, Lifting, VLSI.

1 Introduction

JPEG2000 is the new and current still image compression standard that has been developed under the auspices of the International Organization for Standardization (ISO) [1, 2]. JPEG2000 standard provides many desirable functionalities suitable for interactive multimedia applications. Some of the salient features are as follows.

- *Superior low bit-rate:* JPEG2000 standard offers superior visual quality at very low bit-rates (below 0.1 bit/pixel) compared to the baseline JPEG. For equivalent visual quality JPEG2000 achieves more compression than JPEG.
- *Continuous tone and bi-level image compression:* JPEG2000 standard can be both for continuous-tone (grayscale and color) and bi-level images.
- *Large dynamic range:* JPEG2000 standard allows images with various dynamic range (1 to 38 bits) of pixels for each color component.
- *Large images:* Image size can be as large as $(2^{32} - 1) \times (2^{32} - 1)$ and the maximum number of components in an image can be 2^{14} .
- *Lossless and lossy compression:* The single unified framework provides both lossless and lossy mode of image compression.
- *Progressive transmission:* We can organize the code-stream in a progressive manner in terms of *pixel accuracy* (i.e., visual quality or SNR) of images that allows reconstruction of images with increasing pixel accuracy as more and more compressed bits are received and decoded. The code-stream can also be organized as progressive in resolution such that the higher-resolution images are generated as more compressed data are received and decoded.
- *Region of interest (ROI) coding:* User may desire certain parts of an image that are of greater importance to be encoded with higher fidelity compared to the rest of the image.
- *Random access and compressed domain processing:* By randomly extracting the code-blocks from the compressed bitstream, it is possible to manipulate certain areas of the image. Some of the examples of

compressed-domain processing could be cropping, flipping, rotation, translation, scaling, feature extraction, etc.

- *Robustness to bit-errors (error resiliency)*: Robustness to bit-errors is highly desirable for transmission of images over noisy communications channels. The JPEG2000 standard facilitates this by coding small size independent code-blocks and including resynchronization markers in the syntax of the compressed bitstream. There are also provisions to detect and correct errors within each code-block.
- *Sequential buildup capability*: The JPEG2000-compliant system can be designed to encode an image from top to bottom in a single sequential pass without the need to buffer an entire image, and hence is suitable for low-memory on-chip VLSI implementation.
- *Metadata*: The extended file syntax format allows inclusion of meta-data information to describe the data into the compressed bitstream.

Organization of the rest of the paper is as follows. In the next section, we briefly review the JPEG2000 standard, describe the principles and underlying algorithms to define the standard, and description of different parts of the standard. In section 3, we present a top-level systems architecture for VLSI implementation of the core coding system of JPEG2000 standard. In section 4, we present the lifting implementation of the discrete wavelet transform and review number of key architectures for VLSI implementation proposed in the literature. A VLSI architecture for EBCOT to implement the fractional bit plane coding is presented in section 5. In section 5, we also reviewed some of the efficient and elegant architectures for EBCOT proposed in the literature. We conclude this paper in section 6.

2 JPEG2000 Standard

JPEG2000 standard has a number of parts as follows:

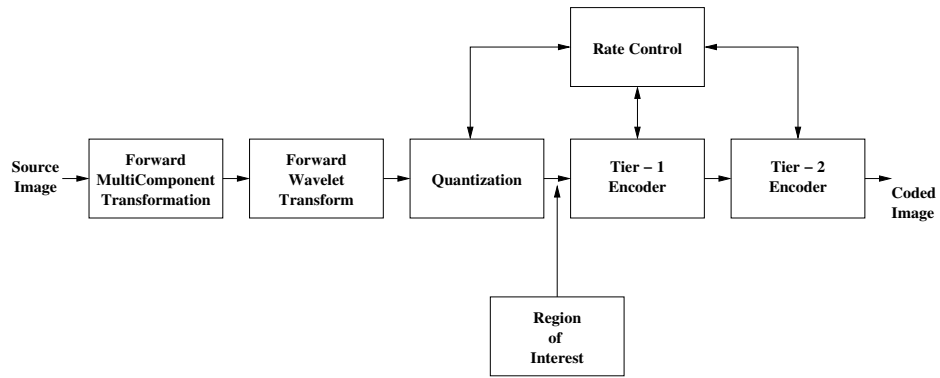
- Part 1—This is the *Core Coding System* which specifies the basic features and code-stream syntax for JPEG2000. [2]
- Part 2— This describes the *extensions* to the core coding system [3].
- Part 3—Motion JPEG2000 [4] specifies a file format (MJ2) that contains an image sequence encoded with the JPEG2000 core coding algorithm for motion video.

- Part 4—Conformance Testing [5] specifies compliance-testing procedures for encoding/decoding using Part 1 of JPEG2000.
- Part 5—In *Reference Software* [6] part, two software source packages are provided for the purpose of testing and validation for JPEG2000 systems.
- Part 6—*Compound Image File Format* [7] specifies another file format (JPM) for the purpose of storing compound images.
- *Part 7*—This part has been abandoned.
- Part 8—*Secure JPEG2000* (JPSEC) deals with security aspects for JPEG2000 applications such as encryption, watermarking, etc.
- Part 9—Interactivity Tools, APIs and Protocols (JPIP). This part defines an interactive network protocol, and it specifies tools for efficient exchange of JPEG2000 images and related metadata.
- Part 10—*3D and Floating Point Data* (JP3D) part is developed with the concern of three-dimensional data such as 3D medical image reconstruction, as an example.
- Part 11—*Wireless* (JPWL) part is developed for wireless multimedia applications and deals with error protection, detection, and correction.
- Part 12—ISO Base Media File Format has a common text with ISO/IEC 14496-12 for MPEG-4.

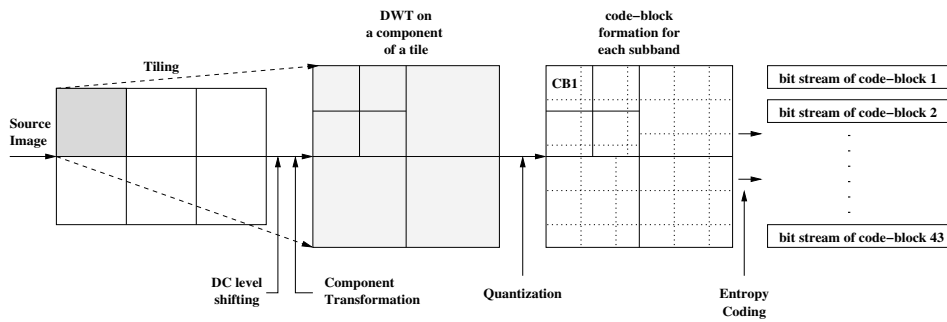
In this paper, we mainly focused on the JPEG2000 Part 1 standard for core coding system [2]. The core coding system can be divided into three phases. We call them (1) image preprocessing, (2) compression, and (3) compressed bitstream formation. The computational block diagram of the functionalities of the compression system is shown in Figure 1(a). The data flow of the compression system is shown in Figure 1(b).

2.1 Image Preprocessing

The image preprocessing phase consists of three major functions (optional): first *tiling*, then *DC level shifting*, followed by the *multicomponent transformation*.



(a)



(b)

Figure 1: (a) Block diagram of the JPEG2000 encoder algorithm; (b) dataflow.

2.1.1 Tiling

If the input source is very large, it can be partitioned into a number of rectangular nonoverlapping blocks. Each of these blocks is called a *tile*. The tiles can be of arbitrary size up to the original image. The tiles are compressed independently. For VLSI implementation, it requires large on-chip memory to buffer large tiles mainly for DWT computation. The tile size 256×256 or 512×512 is found to be a typical choice for VLSI implementation based on the cost, area, and power consideration.

2.1.2 DC Level Shifting

The purpose of DC level shifting is to ensure that the input image samples are approximately centered around zero. DC level shifting is performed on image samples that are represented by unsigned integers only. All samples $I_i(x, y)$ in the i^{th} component of the image (or tile) are level shifted by subtracting the same quantity $2^{S_{siz}^i - 1}$ to produce the DC level shifted sample $I'_i(x, y)$ as follows,

$$I'_i(x, y) \leftarrow I_i(x, y) - 2^{S_{siz}^i - 1}$$

where S_{siz}^i is the precision of image samples signaled in the SIZ (image and tile size) marker segment in compressed bitstream.

2.1.3 Multicomponent Transformations

The multicomponent transform reduces the correlations (if any) amongst the components in a multicomponent image. The standard defines an optional multicomponent transformation in the first three components only. These first three components can be interpreted as three color planes (R, G, B) for ease of understanding. That's why they are often called multicomponent color transformation as well. In general, each component can have different bit-depth (precision of each pixel in a component) and different dimension. However, the condition of application of multicomponent transform is that the first three components should have identical bit-depth and identical dimension as well. There are two different transformations: (1) reversible color transform (RCT), and (2) irreversible color transform (ICT).

For lossless compression of an image, the *reversible color transform* (RCT) is essential because the pixels can be exactly reconstructed by the inverse RCT. The forward and inverse RCT functions are

$$\begin{pmatrix} Y_r \\ U_r \\ V_r \end{pmatrix} = \begin{pmatrix} \lfloor \frac{R+2G+B}{4} \rfloor \\ B - G \\ R - G \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} G \\ R \\ B \end{pmatrix} = \begin{pmatrix} Y_r - \lfloor \frac{U_r+V_r}{4} \rfloor \\ V_r + G \\ U_r + G \end{pmatrix}$$

The *Irreversible Color Transformation* (ICT) is used in lossy compression. ICT is the same as the luminance–chrominance color transformation used in baseline JPEG. Y is the luminance component of the image representing intensity of the pixels (light) and Cb and Cr are the two chrominance components representing the color information in each pixel. The transformation matrix for forward and inverse ICT are

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299000 & 0.587000 & 0.114000 \\ -0.168736 & -0.331264 & 0.500000 \\ 0.500000 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

and

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.00000 & 0.0 & 1.402000 \\ 1.00000 & -0.344136 & -0.714136 \\ 1.00000 & 1.772000 & 0.0 \end{bmatrix} \times \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix}$$

2.2 Compression

After the preprocessing phase, the tiles are independently compressed. The *compression* process is mainly divided into three sequential steps: (1) *Discrete Wavelet Transform* (DWT), (2) *Quantization*, and (3) *Tier-1 coding* (Entropy Encoding). As shown in Figure 1(b), each component (tile) is independently analyzed by a suitable discrete wavelet transform (DWT). In part I of the standard, two different filter banks have been recommended. The (5, 3) filter is recommended for lossless compression, whereas (9, 7) filter is recommended for lossy compression. The DWT essentially decomposes each component into a number of subbands in different resolution levels. Each subband is then independently quantized by a quantization parameter, in case of lossy compression. The quantized subbands are then divided into a number of smaller *code-blocks* of equal size, except for the code-blocks at the boundary of each subband. Typical size of the code-blocks is usually 32×32 or 64×64 for better memory handling and is very suitable for VLSI implementation with on-chip memory in the encoder architecture. Each code-block is then entropy encoded independently to produce compressed bitstreams as shown in the dataflow diagram in Figure 1(b).

2.2.1 Discrete Wavelet Transforms

Discrete Wavelet Transform (DWT) has been effectively used in signal and image processing applications ever since Mallat [8] proposed the multiresolution representation of signals based on wavelet decomposition.

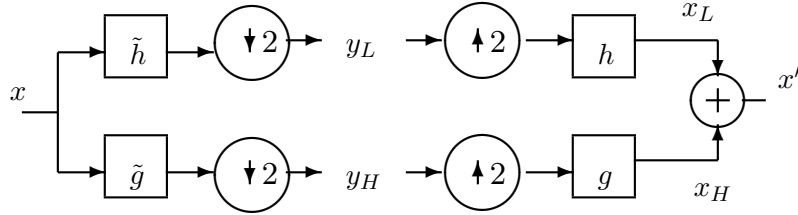


Figure 2: Signal analysis and reconstruction in 1D DWT.

DWT has traditionally been implemented by convolution or FIR filter bank structures. In forward DWT, the input signal (x) is filtered separately by a low-pass filter (\tilde{h}) and a high-pass filter (\tilde{g}). The two output streams are then sub-sampled by simply dropping the alternate output samples in each stream to produce the low-pass (y_L) and high-pass (y_H) subband outputs as shown in Figure 2. The two filters (\tilde{h}, \tilde{g}) form the *analysis* filter bank. The original signal can be reconstructed by a *synthesis* filter bank (h, g) starting from y_L and y_H as shown in Figure 2. Given a discrete signal $x(n)$, the output signals $y_L(n)$ and $y_H(n)$ in Figure 2 can be computed as follows:

$$y_L(n) = \sum_{i=0}^{\tau_L-1} \tilde{h}(i)x(2n-i), \quad y_H(n) = \sum_{i=0}^{\tau_H-1} \tilde{g}(i)x(2n-i) \quad (1)$$

where τ_L and τ_H are the lengths of the low-pass (\tilde{h}) and high-pass (\tilde{g}) filters respectively. During the inverse transform computation, both y_L and y_H are first up-sampled by inserting zeros in between two samples and then filtered by low-pass (h) and high-pass (g) filters respectively. Then they are added together to obtain the reconstructed signal (x') as shown in Figure 2.

For multiresolution wavelet decomposition, the low-pass subband (y_L) is further decomposed in a similar fashion in order to get the second-level of decomposition, and the process repeated. The inverse process follows similar multi-level synthesis filtering in order to reconstruct the signal. A two level DWT decomposition and its reconstruction have been shown in

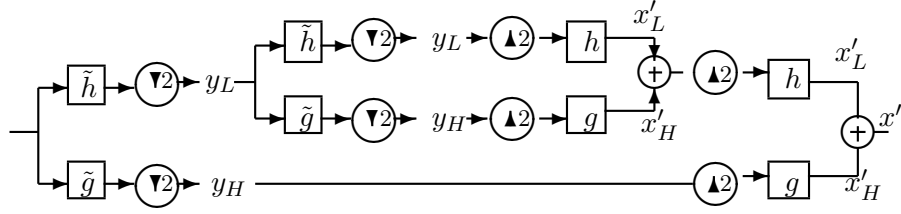
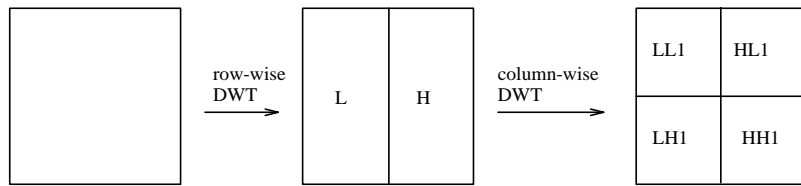
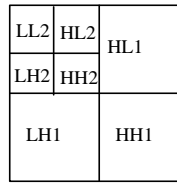


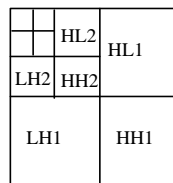
Figure 3: Signal analysis and reconstruction in two-level 1D DWT.



(a) First level of decomposition



(b) Second level of decomposition



(c) Third level of decomposition

Figure 4: Three levels of decomposition in 2D DWT.

Figure 3, as an example. Since two dimensional wavelet filters are separable functions, 2D DWT can be obtained by first applying the 1D DWT row-wise (to produce L and H subbands in each row) and then column-wise as shown in Figure 4(a). In the first level of decomposition, four subbands LL1, LH1, HL1 and HH1 are obtained. Repeating the same in the LL1 subband, it produces LL2, LH2, HL2 and HH2 and so on, as shown in Figure 4(c).

2.2.2 Quantization

After the DWT, all the subbands are quantized in lossy compression mode in order to reduce the precision of the subbands to aid in achieving compression. Quantization is not performed in case of lossless encoding. In Part 1 of the

standard, the quantization is performed by uniform scalar quantization with dead-zone about the origin. In dead-zone scalar quantizer with step-size Δ_b , the width of the dead-zone (i.e., the central quantization bin around the origin) is $2\Delta_b$ as shown in Figure 5. The standard supports separate quantization step sizes for each subband. The quantization step size (Δ_b) for a subband (b) is calculated based on the dynamic range of the subband values. The formula of uniform scalar quantization with a dead-zone is

$$q_b(i, j) = \text{sign}(y_b(i, j)) \left\lfloor \frac{|y_b(i, j)|}{\Delta_b} \right\rfloor, \quad (2)$$

where $y_b(i, j)$ is a DWT coefficient in subband b and Δ_b is the quantization step size for the subband b . All the resulting quantized DWT coefficients $q_b(i, j)$ are signed integers.

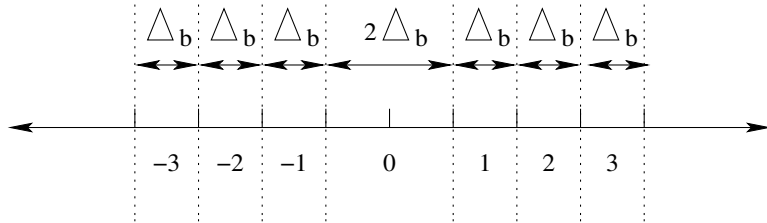


Figure 5: Dead-zone quantization about the origin.

All the computations up to the quantization step are carried out in two's complement form. After the quantization, the quantized DWT coefficients are converted into sign-magnitude representation prior to entropy coding because of the inherent characteristics of the entropy encoding process.

2.2.3 Tier-1 coding

The quantized wavelet coefficients in each code-block are independently entropy encoded to generate the compressed bit-stream by *Tier-1* coding as shown in Figure 1(b). If the precision of the elements in the code-block is p , then the code-block is decomposed into p bit-planes and they are encoded from the most significant bit-plane to the least significant bit-plane sequentially. Each bit-plane is scanned in a particular scan pattern as shown in Figure 6. The scan pattern can be divided into sections (or stripes), each with four consecutive rows starting from the first row of a code-block. If the total number of rows of a code-block is not a multiple of 4, all the sections will have four consecutive rows except the very last section. The scan

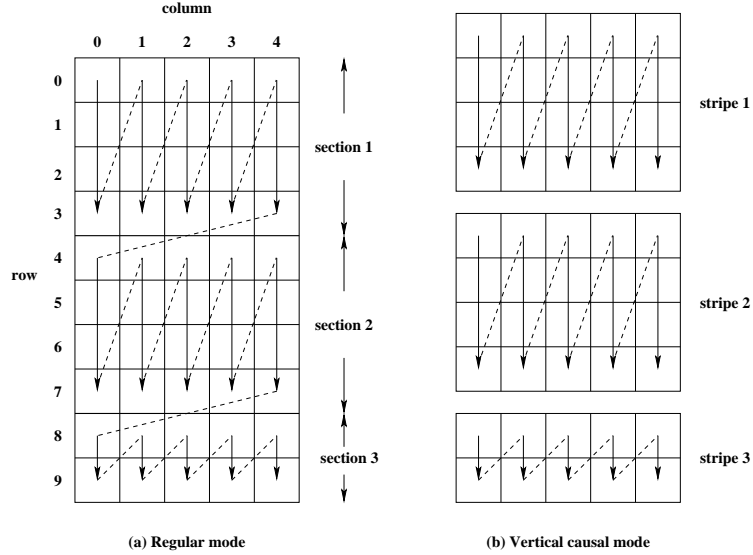


Figure 6: Scan pattern of each bit-plane in a code-block: (a) regular mode; (b) vertical causal mode.

starts from the first section and down to the last section until all elements of a code-block are encoded or decoded. Each section is scanned from the first row of the first column. The next location to be scanned will be the next row on the same column. After a column in the section is completely coded, start scanning at the first row of the next column in the same section. Continue the coding process until all columns in a section are coded. This same process is then applied to the next adjacent section until all of them are coded.

Each bit in a bit-plane is encoded by a *fractional bit-plane coding* (BPC) technique to generate intermediate data in the form of a *context* and a binary *decision* value for each bit position. In JPEG2000 the *embedded block coding with optimized truncation* (EBCOT) algorithm [9] has been adopted for the BPC. The binary decision values generated by the EBCOT are encoded using a context adaptive *binary arithmetic coder* (BAC), called the MQ-coder [10]. The *context* information generated by EBCOT is used to select the estimated probability value from a lookup table and this probability value is used by the MQ-coder to adjust the intervals and generate the compressed codes. JPEG2000 standard uses a predefined lookup table with

47 entries for only 19 possible different contexts for each bit type depending on the coding passes. This facilitates rapid probability adaptation in the MQ-coder and produces compact bitstreams.

EBCOT encodes each bit in only one of the following three coding passes in order.

- *significant propagation pass* (SPP): During SPP, a bit is coded if its location is not significant, but at least one of its eight-connected neighbors is significant. By significant, we mean that the bit is most significant bit of the corresponding sample in the code-block.
- *magnitude refinement pass* (MRP): All the bits that have not been coded in SPP and became significant in a previous bit-plane are coded in this pass.
- *cleanup pass* (CUP): All the bits that have not been coded in either SPP or MRP are coded in this pass. CUP also performs a form of run-length coding to efficiently code a string of zeros.

In EBCOT, the coding passes perform one or more of the four possible coding operations to generate the context (cx) and decision (d) information to be subsequently entropy encoded by the MQ-coder. The four coding operations are *zero coding* (ZC), *sign coding* (SC), *magnitude refinement coding* (MRC), and *run-length coding* (RLC). The coding operations require to use three different state variables σ , σ' , and η . These state variables are represented in the form of three two-dimensional binary arrays and each array has the exactly the same dimension of a code-block. In addition to these state variables, the EBCOT requires two more arrays of size of a code-block. They are *sign array* (χ) and *magnitude array* (v). The sign array is used to store the sign bits of the elements of the code-block. The magnitude array stores the unsigned integers of the code-block elements. The principles of the coding operations and the state variables are explained in great detail in [1, 2, 9].

2.3 Bit-stream formation

After the compressed bits for each code-block are generated by Tier-1 coding, the *Tier-2* coding engine efficiently represents the layer and block summary information for each code-block. A *layer* consists of consecutive bit-plane coding passes from each code-block in a tile, including all the subbands of

all the components in the tile. The block summary information consists of length of compressed code words of the code-block, the most significant magnitude bit-plane at which any sample in the code-block is nonzero, as well as the truncation point between the bitstream layers, among others. The decoder receives this information in an encoded manner in the form of two tag trees. This encoding helps to represent this information in a very compact form without incurring too much overhead in the final compressed file. The encoding process is popularly known as *Tag Tree coding* [1, 2].

2.3.1 Rate Control

Rate control is a process by which the bit-rates (sometimes called coding rates) are allocated in each code-block in each subband in order to achieve the overall target encoding bit-rate for the whole image while minimizing the distortion (errors) introduced in the reconstructed image due to quantization and truncation of codes to achieve the desired code rate [9, 11].

The JPEG2000 encoder generates a number of independent bitstreams by encoding the code-blocks. Accordingly a rate-distortion optimization algorithm generates the truncation points for these bitstreams in an optimal way in order to minimize the distortion according to a target bit rate. There is another inefficient, but simple way to control bit-rate by choosing the quantization step size. The bigger the step size, the lower the rate will be.

The bit-rate control is open issue for the JPEG2000 standard and it is up to the prerogative of the developers how they want to accomplish this. From the hardware implementation perspective, the rate-distortion algorithm requires a microcontroller to compute the breakpoints using a rate-distortion optimization technique and supply these breakpoints to the entropy encoding engine for formation of the compress bitstream.

3 A VLSI Architecture for JPEG2000 Encoder

There are only few papers published in the literature for VLSI implementation of a complete systems architecture of JPEG200 [12, 13]. Here we present the overview of a global architecture for VLSI implementation of JPEG2000 encoder proposed by *Andra, Chakrabarti, and Acharya* [12] is shown in Figure 7. Key building blocks of this architecture are as follows.

- **DWT computation engine:** The DWT computation engine in Figure 7 computes the multilevel wavelet decomposition of the input im-

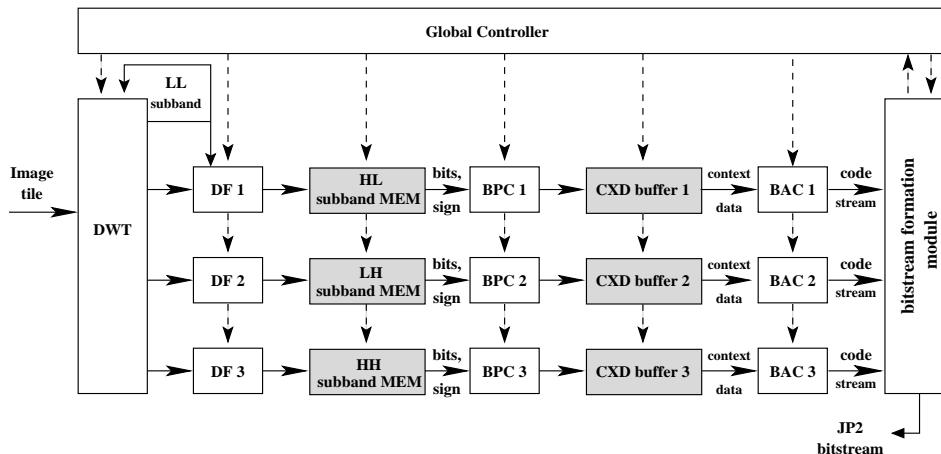


Figure 7: A top-level architecture for the JPEG2000 encoder.

age tile. In each level of decomposition, the DWT engine produces four subbands (LL, HL, LH, and HH). The subband LL is input back to the DWT engine for the next level of decomposition. The other three subbands are input to the subsequent phases of the architecture for entropy encoding.

In the context of the JPEG2000 standard, the lifting-based architecture is more suitable compared to the convolution-based architecture because of the inherent features of lifting-based implementation of DWT, such as reduced number of computations both for (9, 7) and (5, 3) filter banks, in-place computation leading to reduced memory requirements, scope of parallel processing, etc. We describe the details of lifting implementation of DWT and its VLSI architectures in Section 4.

- **Data format (DF) engine:** The *data format* engines DF1, DF2, and DF3 first quantize the DWT coefficients in HL, LH, and HH subbands respectively by corresponding quantization parameters supplied by a *global controller*. For lossless compression, the quantization parameter is 1. Each quantized sample (integer) is represented in a 16-bit word. After quantization, each DF engine converts the two's complement representation of each quantized sample to sign-magnitude representation required by the EBCOT algorithm [9] executed by the BPC modules.

The DF engine then decomposes the subband into a number of code-blocks and stores them in a special local *subband memory* (MEM) as shown in Figure 7. The DF engine also determines the most significant bit-plane of each code-block. The most significant bit-plane is the first bit-plane that contains at least one 1 bit.

- **Subband memory (MEM) module:** A special *subband memory* (MEM) module [12] has been used in this architecture such that the memory structure can handle word-in-bit-out format combined with the stripe structure in order to efficiently handle the input data by the BPC coder. Structure of the subband memory MEM is depicted in Figure 8. There are 32×8 rows and each row is made of 64 bits. Each $b_{R,C}^p$ represents the bit value in row R and column C in the p^{th} bit-plane of the code-block. In this structure, each column (stripe) of the scan pattern is grouped together and each row consists of 16 such columns. Hence 32 columns (each four-row stripe) are stored in two consecutive rows in SM. In this fashion, a 32×32 size code-block is represented by 128 rows each with 64 bits in the subband memory module. The concept can be extended to any size of code-block. A code-block of size $M \times N$ can be represented by $\frac{M}{4} \times N$ rows each with 64 bits.

$b_{0,0}^0$	$b_{1,0}^0$	$b_{2,0}^0$	$b_{3,0}^0$	$b_{0,0}^1$	$b_{1,0}^1$	$b_{2,0}^1$	$b_{3,0}^1$	$b_{3,0}^{15}$
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
$b_{0,31}^0$	$b_{1,31}^0$	$b_{2,31}^0$	$b_{3,31}^0$	$b_{0,31}^1$	$b_{1,31}^1$	$b_{2,31}^1$	$b_{3,31}^1$	$b_{3,31}^{15}$
$b_{4,0}^0$	$b_{5,0}^0$	$b_{6,0}^0$	$b_{7,0}^0$	$b_{4,0}^1$	$b_{5,0}^1$	$b_{6,0}^1$	$b_{7,0}^1$	$b_{7,0}^{15}$
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
$b_{4,31}^0$	$b_{5,31}^0$	$b_{6,31}^0$	$b_{7,31}^0$	$b_{4,31}^1$	$b_{5,31}^1$	$b_{6,31}^1$	$b_{7,31}^1$	$b_{7,31}^{15}$
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
$b_{28,0}^0$	$b_{29,0}^0$	$b_{30,0}^0$	$b_{31,0}^0$	$b_{28,0}^1$	$b_{29,0}^1$	$b_{30,0}^1$	$b_{31,0}^1$	$b_{31,0}^{15}$
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
$b_{28,31}^0$	$b_{29,31}^0$	$b_{30,31}^0$	$b_{31,31}^0$	$b_{28,31}^1$	$b_{29,31}^1$	$b_{30,31}^1$	$b_{31,31}^1$	$b_{31,31}^{15}$

Figure 8: Structure of the subband memory for a 32×32 code-block; $b_{R,C}^p$ represents the bit at location (R, C) in the p^{th} bit-plane of the code-block.

- **Bit-plane coder (BPC) engine:** Each BPC engine essentially exe-

cutes the EBCOT algorithm [9]. The input to the BPC engine is read from the *subband memory* (MEM) module, which consists of both the sign bit and the bit-planes of the code-block. Output of the BPC engine is a sequence of 5-bit context and 1-bit data pair (cx, d) temporarily stored in an internal buffer (CXD) before they are encoded by the QM-coder (BAC) in the subsequent pipeline stage of the architecture. In this JPEG2000 encoder architecture, three BPC engines have been used to encode the code-blocks from the three subbands HL, LH, and HH as shown in Figure 7. However, multiple BPC engines can be used to process multiple code-blocks from each subband to exploit the data parallelism because the encodings of the code-blocks are independent from each other and hence speed up the architecture at the cost of additional hardware resources.

- **Context and Data (CXD) buffer:** The CXD buffer is a FIFO with a read port and a write port so that the BPC module can write the 6-bit data (5 bits for context and 1 bit for data) in the FIFO and BAC can read the 6-bit data from the FIFO in parallel. The size of the FIFO should be large enough in order to manage the speed difference between the BPC and BAC in the pipeline. The control circuit (global controller) manages the pointers of the FIFO for both the read and write operations.
- **Binary arithmetic coding (BAC) engine:** Each BAC engine executes the MQ-encoding algorithm [10] to encode the data bit of the context-data pair (cx, d) read from the CXD FIFO.
- **Global controller:** The global controller generates all the necessary control signals to enable all the modules in the architecture including loading the image tiles, reading and writing the subband memory, controlling the pointers to access the CXD FIFO, generating the control signals for the BPC and BAC encoders, and loading the Q-table for the BAC module. It is also used to handle the rate-control and generation of the final code-stream and the header of the compressed file. The Tag Tree coding and organization of the compressed code can be done by either a local microcontroller available in the chip or the host processor controlling the chip.

The DWT and EBCOT operations are computationally very expensive as well as memory intensive. The EBCOT operations is a control intensive

operations as well. In the following sections, we review efficient VLSI algorithms and architecture for implementation of DWT and EBCOT engine only. We avoid MQ-coder architectures in this paper because of limited space constraint.

4 Lifting Implementation of DWT and VLSI Architectures

For the filter bank in Figure 2, the conditions for perfect reconstruction of a signal [14, 15] are given by

$$\begin{aligned} h(z)\tilde{h}(z^{-1}) + g(z)\tilde{g}(z^{-1}) &= 2, \\ h(z)\tilde{h}(-z^{-1}) + g(z)\tilde{g}(-z^{-1}) &= 0 \end{aligned} \quad (3)$$

where $h(z)$ is the Z -transform of the FIR filter h , as an example, and we can define *polyphase matrices* as

$$\tilde{P}(z) = \begin{bmatrix} \tilde{h}_e(z) & \tilde{h}_o(z) \\ \tilde{g}_e(z) & \tilde{g}_o(z) \end{bmatrix}, \quad P(z) = \begin{bmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{bmatrix} \quad (4)$$

where h_e contains the even coefficients and h_o contains the odd coefficients of the FIR filter h . Often $P(z)$ is called the *dual* of $\tilde{P}(z)$. The wavelet transform in terms of the polyphase matrix can be expressed as

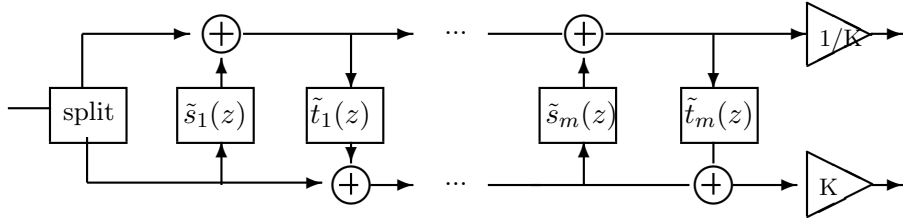
$$\begin{bmatrix} y_L(z) \\ y_H(z) \end{bmatrix} = \tilde{P}(z) \begin{bmatrix} x_e(z) \\ z^{-1}x_o(z) \end{bmatrix}, \quad \begin{bmatrix} x_e(z) \\ z^{-1}x_o(z) \end{bmatrix} = P(z) \begin{bmatrix} y_L(z) \\ y_H(z) \end{bmatrix}$$

for the forward DWT and inverse DWT respectively. It has been shown in [14, 15] that if (\tilde{h}, \tilde{g}) is a complementary filter pair, we can apply the Euclidean algorithm to factorize $\tilde{P}(z)$ into a finite sequence of alternating upper and lower triangular matrices as follows:

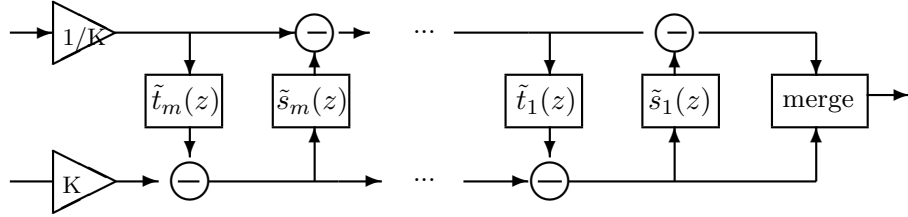
$$\tilde{P}(z) = \left\{ \prod_{i=1}^m \begin{bmatrix} 1 & \tilde{s}_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \tilde{t}_i(z) & 1 \end{bmatrix} \right\} \begin{bmatrix} K & 0 \\ 0 & \frac{1}{K} \end{bmatrix} \quad (5)$$

where K is a constant, $\tilde{s}_i(z)$ and $\tilde{t}_i(z)$ (for $1 \leq i \leq m$) are Laurent polynomials of lower orders. Computation of the upper and lower triangular matrices are known as *primal lifting* and *dual lifting* respectively [14, 15]. Often these two basic lifting steps are called *update* and *predict* as well.

Hence the lifting based forward wavelet transform essentially is to first apply the *lazy* wavelet on the input stream (split into even and odd samples), then alternately execute *primal* and *dual* lifting steps, and finally *scale* the two output streams by $\frac{1}{K}$ and K respectively, to produce low-pass and high-pass subbands, as shown in Figure 9(a). The inverse DWT can be derived by traversing above steps in the reverse direction as shown in Figure 9(b).



(a) Forward transform



(b) Inverse transform

Figure 9: Lifting based forward and inverse DWT.

4.1 Lifting of wavelet filters for JPEG2000

The core coding system of JPEG2000 standard recommended two wavelet filters. The (5, 3) filter bank is recommended for lossless compression, whereas the (9, 7) filter bank has been recommended for lossy compression [1, 2].

Consider the (5,3) wavelet filter $\tilde{h} = (-\frac{1}{8}, \frac{1}{4}, \frac{3}{4}, \frac{1}{4}, -\frac{1}{8})$ and $\tilde{g} = (-\frac{1}{2}, 1, -\frac{1}{2})$. The polyphase matrix of this filter bank is

$$\tilde{P}(z) = \begin{bmatrix} \tilde{h}_e(z) & \tilde{h}_o(z) \\ \tilde{g}_e(z) & \tilde{g}_o(z) \end{bmatrix} = \begin{bmatrix} -\frac{1}{8}z^{-1} + \frac{3}{4} - \frac{1}{8}z & \frac{1}{4} + \frac{1}{4}z \\ -\frac{1}{2}z^{-1} - \frac{1}{2} & 1 \end{bmatrix}$$

which can be factorized as

$$\tilde{P}(z) = \begin{bmatrix} 1 & \frac{1}{4}(1+z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\frac{1}{2}(1+z^{-1}) & 1 \end{bmatrix}$$

If the samples are numbered starting from 0, the even terms of the output stream form the lowpass subband and the odd terms form the highpass subband. Hence, we can interpret the above matrices in the time domain as $y_{2i+1} = b(x_{2i} + x_{2i+2}) + x_{2i+1}$ and $y_{2i} = a(y_{2i+1} + y_{2i+3}) + x_{2i}$, where $a = -\frac{1}{2}$ and $b = \frac{1}{4}$, $0 \leq i \leq \frac{N}{2}$.

For the (9, 7) wavelet filter, the polyphase can be factorized [15] as

$$\tilde{P}(z) = \begin{bmatrix} 1 & a(1+z^{-1}) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ b(1+z) & 1 \end{bmatrix} \begin{bmatrix} 1 & c(1+z^{-1}) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ d(1+z) & 1 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & \frac{1}{K} \end{bmatrix}$$

where $a = -1.586134342$, $b = -0.05298011854$, $c = 0.8829110762$, $d = -0.4435068522$, $K = 1.149604398$.

Hence, the forward transform for (5, 3) and (9,7) filters can be represented as $Y_{(5,3)} = XM_1M_2$ and $Y_{(9,7)} = XM_1M_2M_3M_4$ respectively, where

$$M_1 = \begin{bmatrix} 1 & a & 0 & . & . & . & . & . & . \\ 0 & 1 & 0 & 0 & . & . & . & . & . \\ 0 & a & 1 & a & 0 & . & . & . & . \\ . & 0 & 0 & 1 & 0 & 0 & . & . & . \\ . & . & 0 & a & 1 & a & 0 & . & . \\ . & . & . & 0 & 0 & 1 & 0 & 0 & . \\ . & . & . & . & 0 & a & 1 & a & 0 \\ . & . & . & . & . & 0 & 0 & 1 & 0 \\ 0 & . & . & . & . & . & 0 & a & 1 \end{bmatrix}, \quad M_2 = \begin{bmatrix} 1 & 0 & 0 & . & . & . & . & . & . \\ 0 & 1 & b & 0 & . & . & . & . & . \\ 0 & 0 & 1 & 0 & 0 & . & . & . & . \\ . & 0 & b & 1 & b & 0 & . & . & . \\ . & . & 0 & 0 & 1 & 0 & 0 & . & . \\ . & . & . & 0 & b & 1 & b & 0 & . \\ . & . & . & . & 0 & 0 & 1 & 0 & 0 \\ . & . & . & . & . & 0 & b & 1 & 0 \\ 0 & . & . & . & . & . & 0 & 0 & 1 \end{bmatrix}$$

$$M_3 = \begin{bmatrix} 1 & c & 0 & . & . & . & . & . & . \\ 0 & 1 & 0 & 0 & . & . & . & . & . \\ 0 & c & 1 & c & 0 & . & . & . & . \\ . & 0 & 0 & 1 & 0 & 0 & . & . & . \\ . & . & 0 & c & 1 & c & 0 & . & . \\ . & . & . & 0 & 0 & 1 & 0 & 0 & . \\ . & . & . & . & 0 & c & 1 & c & 0 \\ . & . & . & . & . & 0 & 0 & 1 & 0 \\ 0 & . & . & . & . & . & 0 & c & 1 \end{bmatrix}, \quad M_4 = \begin{bmatrix} 1 & 0 & 0 & . & . & . & . & . & . \\ 0 & 1 & d & 0 & . & . & . & . & . \\ 0 & 0 & 1 & 0 & 0 & . & . & . & . \\ . & 0 & d & 1 & d & 0 & . & . & . \\ . & . & 0 & 0 & 1 & 0 & 0 & . & . \\ . & . & . & 0 & d & 1 & d & 0 & . \\ . & . & . & . & 0 & 0 & 1 & 0 & 0 \\ . & . & . & . & . & 0 & d & 1 & 0 \\ 0 & . & . & . & . & . & 0 & 0 & 1 \end{bmatrix}$$

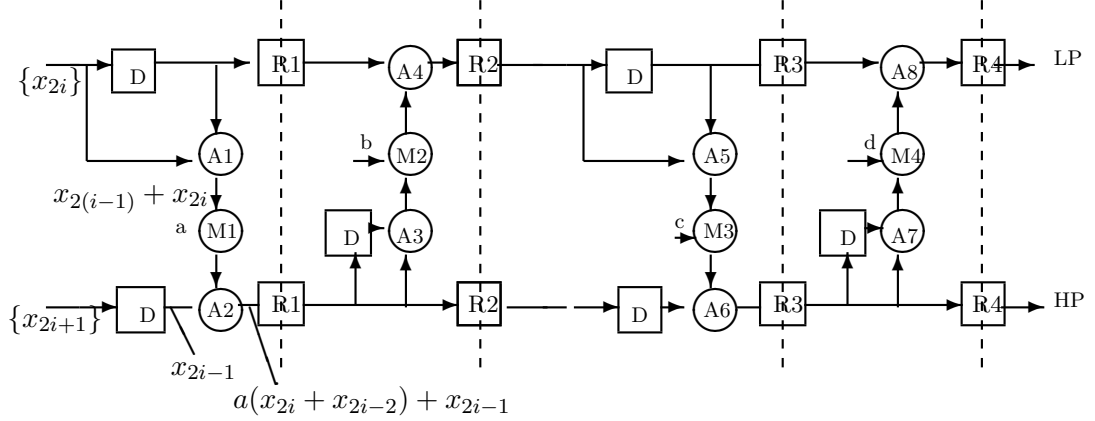


Figure 11: The direct mapped architecture in [17].

pipeline registers (R). There are two input lines to the architecture: one that inputs even samples $\{x_{2i}\}$, and the other one that inputs odd samples $\{x_{2i+1}\}$. There are four pipeline stages in the architecture. In the first pipeline stage, adder A1 computes $x_{2i} + x_{2i-2}$ and adder A2 computes $a(x_{2i} + x_{2i-2}) + x_{2i-1}$. The output of A2 corresponds to the intermediate results generated in the first stage of Figure 10. The output of adder A4 in the second pipeline stage corresponds to the intermediate results generated in the second stage of Figure 10. Continuing in this fashion, adder A6 in the third pipeline stage produces the high-pass output samples, and adder A8 in the fourth pipeline stage produces the low-pass output samples. For lifting schemes that require only 2 lifting steps, such as the (5, 3) filter, the last two pipeline stages need to be bypassed causing the hardware utilization to be only 50% or less. Also, for a single read port memory, the odd and even samples are read serially in alternate clock cycles and buffered. This slows down the overall pipelined architecture by 50% as well.

4.3 Folded Architecture

The pipelined architecture in Figure 11 can be further improved by carefully folding the last two pipeline stages into the first two stages [19] as shown in Figure 12. The architecture consists of two pipeline stages, with three pipeline registers R1, R2 and R3. In (9, 7) type filtering operation, inter-

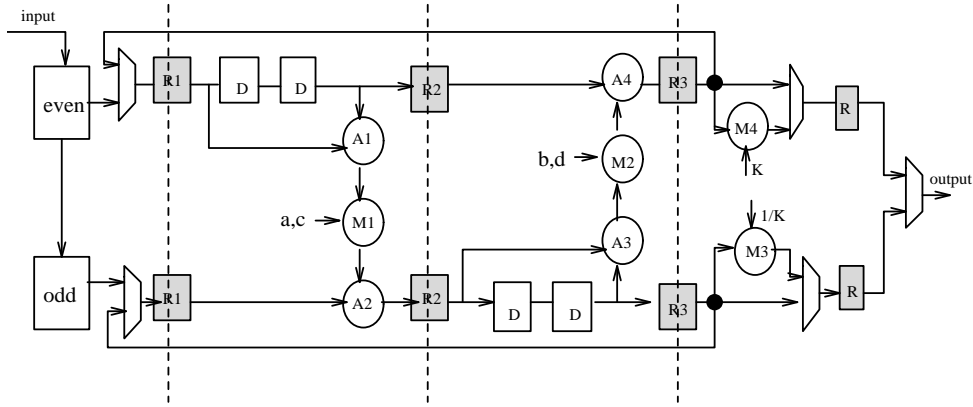


Figure 12: The folded architecture in [19].

mediate data (R3) generated after the first two lifting steps (phase 1) are folded back to R1 for computation of the last two lifting steps (phase 2). The architecture can be reconfigured so that computation of the two phases can be interleaved by selection of appropriate data by the multiplexers. As a result, two delay registers (D) are needed in each lifting step in order to properly schedule the data in each phase. Based on the phase of interleaved computation, the coefficient for multiplier M1 is either a or c , and similarly the coefficient for multiplier M2 is either b or d . The hardware utilization of this architecture is always 100%. Note that for the (5, 3) type filter operation, folding is not required.

4.4 Flipping Architecture

While conventional lifting-based architectures require fewer arithmetic operations, they sometimes have long critical paths. For instance, the critical path of the lifting-based architecture for the (9,7) filter is $4T_m + 8T_a$ while that of the convolution implementation is $T_m + 4T_a$. One way of improving this is by pipelining which results in a significant increase in the number of registers. For instance, to pipeline the lifting-based (9,7) filter such that the critical path is $T_m + 2T_a$, 6 additional registers are required.

The timing accumulation problem can be eliminated by removing the multiplications along the critical path [20]. This is done by scaling the remaining paths by the inverse of the multiplier coefficients. Figure 13(a)-(c) describes how scaling at each level can reduce the multiplications in the

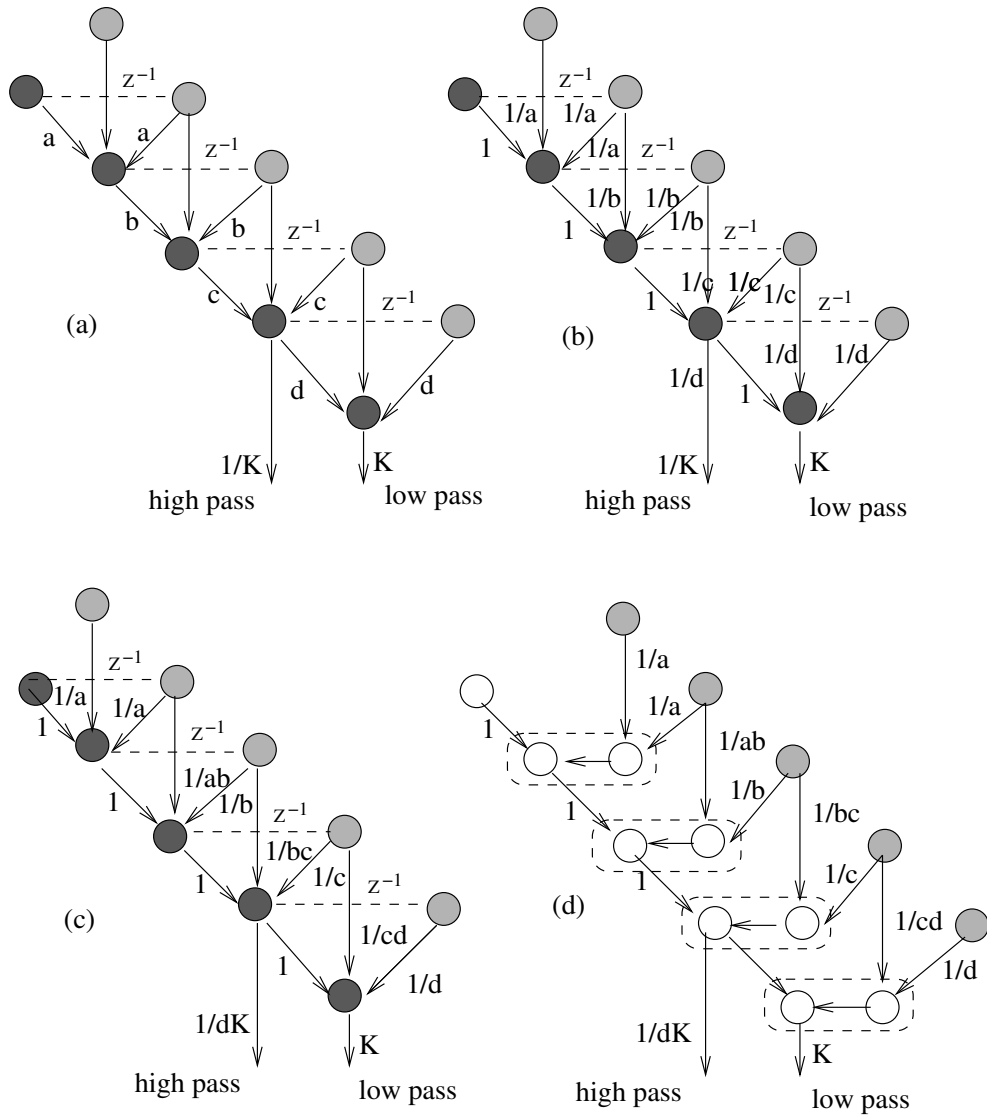


Figure 13: A Flipping Architecture [20]. (a) Original architecture, (b)-(c) Scaling the coefficients to reduce the number of multiplications, (d) Splitting the three-input addition nodes to two-input nodes.

critical path. Figure 13(d) further splits the three input addition nodes into two 2-input adders. The critical path is now $T_m + 5T_a$. The minimum critical path of T_m can be achieved by 5 pipelining stages using 11 pipelining registers (not shown in the figure). Detailed hardware analysis of lossy (9,7), integer (9,7) and (6,10) filters have been included in [20]. Further more, since the flipping transformation changes the round-off noise considerably, techniques to address precision and noise problems have also been addressed in [20].

4.5 Recursive Architecture

Most of the traditional DWT architectures compute the i th level of decomposition upon completion of the $(i - 1)$ th level of decomposition. However, in multiresolution DWT, the number of samples to be processed in each level is always half of the size in the previous level. Thus it is possible to process multiple levels of decomposition simultaneously. This is the basic principle of a *recursive architecture* that was first proposed for a convolution based DWT in [21] and applied for lifting based DWT in [22, 23]. Here computations in higher levels of decomposition is initiated as soon as enough intermediate data in low-frequency subband is available for computation. The proposed architecture for a 3-level decomposition of an input signal using Daubechies-4 DWT is shown in Figure 14.

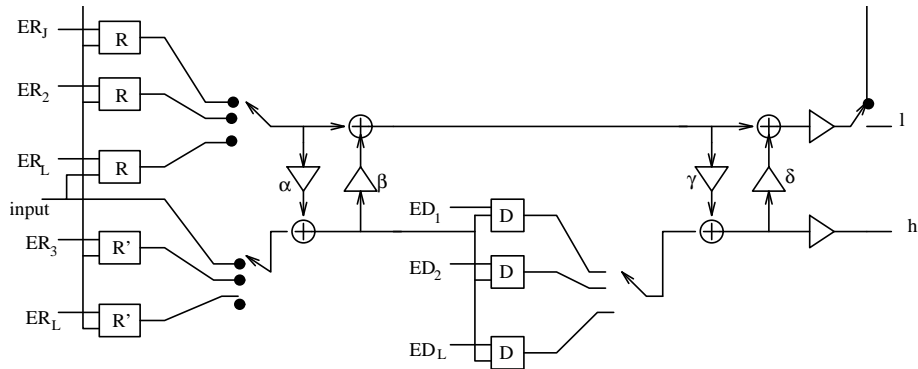


Figure 14: The Recursive Architecture in [22].

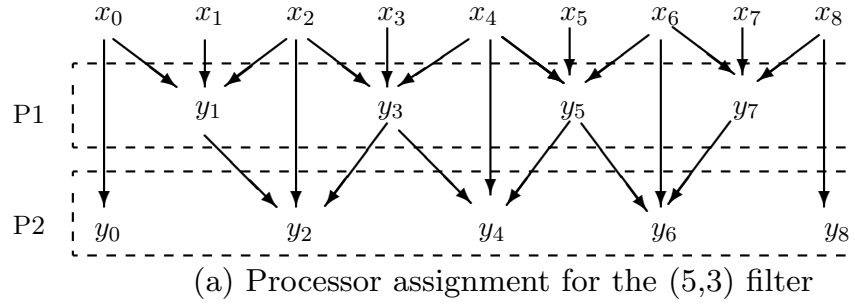
The basic circuit elements used in this architecture are delay elements, multipliers and MAC units which are in turn designed using a multiplier, an adder and two shifters. The multiplexors M1 and M2 select the even and

odd samples of the input data as needed by the lifting scheme. S1, S2 and S3 are the control signals for data flow of the architecture. The select signal (S1) of the multiplexors is set to 0 for the first level of computation and is set to 1 during the second or third level computation. The switches S2 and S2 select the input data for the second and third level of computation. The multiplexor M3 selects the delayed samples for each level of decomposition based on the clocked signals shown in Figure 14.

4.6 A Programmable Generalized Architecture

The architecture proposed by *Andra, Chakrabarti, and Acharya* [24] is an example of a highly programmable architecture that can support a large set of filters. These include filters (5,3), (9,7), C(13,7), S(13,7), (2,6), (2,10), and (6,10). Since the data dependencies in the filter computations can be represented by at most four stages, the architecture consists of four processors, where each processor is assigned computations of one stage. Figure 15(a) describes the assignment of computation to two processors, P1 and P2, for the (5,3) filter which can be represented by two stages.

The processor architecture consists of adders, multipliers and shifters that are interconnected in a manner that would support the computational structure of the specific filter. Figure 16 describes the processor architectures for the (5,3) filter and the (9,7) filter. While the (5,3) filter architecture consists of two adders, and a shifter, the (9,7) filter architecture consists of two adders and a multiplier. Figure 15(b) describes part of the schedule for the (5,3) filter. The schedules are generated by mapping the data dependency graph onto the resource-constrained architecture. It is assumed that the delays of each adder, shifter and the multiplier are 1, 1 and 4 time units respectively. For example, Adder1 of P1 adds the elements (x_0, x_2) in the 2^{nd} cycle and stores the sum in register $R1$. The shifter reads this sum in the next cycle (3^{rd} cycle), carries out the required number of shifts (one right shift as $a = -0.5$) and stores the data in register Rs . The second adder (Adder2) reads the value in Rs and subtracts the element x_1 to generate y_1 in the next cycle. To process $N = 9$ data, the P1 processor takes four cycles. Adder 1 in P2 processor starts computation in the sixth cycle. The gaps in the schedules for P1 and P2 are required to store the zeroth element of each row.



Cycle	Processor 1 (P1)			Processor 2 (P2)		
	Adder1	Shifter	Adder2	Adder1	Shifter	Adder2
1	–	–	–	–	–	–
2	$x_0 + x_2$	–	–	–	–	–
3	$x_2 + x_4$	R1	–	–	–	–
4	$x_4 + x_6$	R1	Rs- $x_1=y_1$	–	–	–
5	$x_6 + x_8$	R1	Rs- $x_3=y_3$	–	–	–
6		R1	Rs- $x_5=y_5$	y_1, y_3	–	–
7		–	Rs- $x_7=y_7$	y_3, y_5	R1	y_0
8				y_5, y_7	R1	Rs+ x_2
9					R1	Rs+ x_4
10						Rs+ x_6

(b) Partial schedule for the (5,3) filter implementation

Figure 15: Processor assignment and partial schedule for the (5,3) filter implementation in the Generalized architecture in [24].

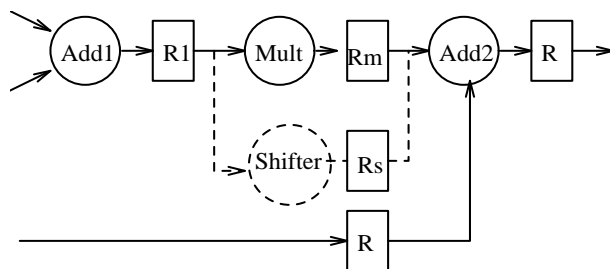
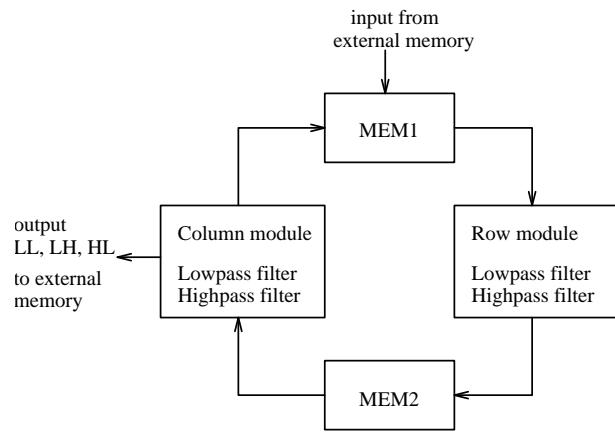


Figure 16: Processor architecture for the (5,3) and (9,7) filters in [24]

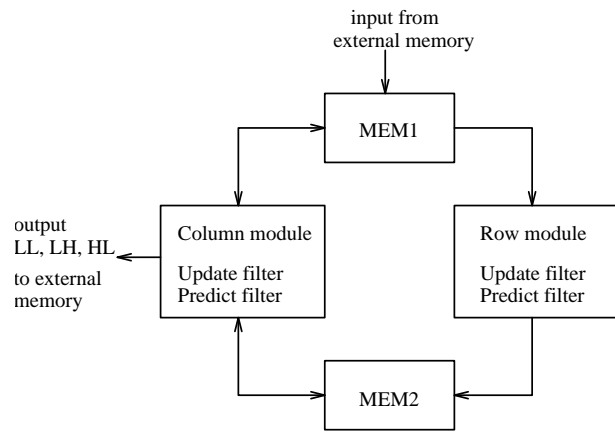
4.7 Two Dimensional DWT Architecture

Generally, 2D wavelet filters are separable functions. A straight-forward approach for 2D implementation is to first apply the 1D DWT row-wise (to produce L and H subbands) and then column-wise to produce four subbands LL, LH, HL and HH as shown in Figure 4 in each level of decomposition. Obviously, the processor utilization is a concern in direct implementation of this approach because it requires all the rows be filtered before the column-wise filtering can begin and thus it requires a size of memory buffer of the order of the image size. The alternative approach is to begin the column-processing as soon as sufficient number of rows have been filtered. The column-wise processing is now performed on these available lines to produce wavelet coefficients row-wise. The overview of the two-dimensional architecture for convolution based DWT is shown in Figure 17(a). The row module reads the data from memory MEM1, performs DWT along the rows and writes the data into memory MEM2. The column module reads the data from MEM2, performs DWT along the columns and writes ‘LL’ data to MEM1 and ‘LH’, ‘HL’, ‘HH’ data to external memory.

A similar approach can be implemented for the lifting scheme as well. The basic idea of lifting based approach for DWT implementation is to replace the parallel low-pass and high-pass filtering of traditional approach by a sequence of alternating smaller filters. The computations in each filter can be partitioned into prediction (dual lifting) and update (primal lifting) stages as shown in Figure 17(b). Here the row module reads the data from MEM1, performs the DWT along the rows (‘H’ and ‘L’) and writes the data into MEM2. The prediction filter of the column module reads the data from MEM2, performs column-wise DWT along alternate rows (‘HH’ and ‘LH’) and writes the data into MEM2; the update filter of the column



(a) Convolution-based architecture



(b) Lifting-based architecture

Figure 17: Overview of convolution and lifting-based 2D DWT Architecture.

module reads the data from MEM2, performs column-wise DWT along the remaining rows, and writes the ‘LL’ data into MEM1 for higher octave computations and ‘HL’ data to external memory. Note that this is a generic architectural flow and is the backbone of the existing 2D architectures.

The architecture proposed by *Andra, Chakrabarti, and Acharya* [24] can compute a large set of filters for both the 2D forward and inverse transforms. It supports two classes of architectures based on whether lifting is implemented by one or two lifting steps. The M2 architecture corresponds to implementation using one lifting step or two factorization matrices, and the M4 architecture corresponds to implementation by two lifting steps or four factorization matrices.

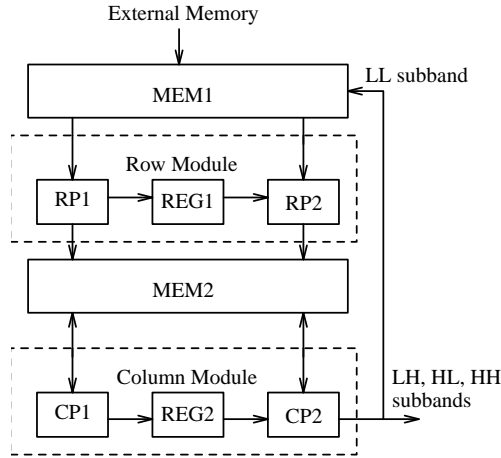


Figure 18: Block diagram of the generalized M2 architecture in [24].

The dataflow of the M2 architecture is similar to that in Figure 17(b). A block diagram of the M2 architecture is shown in Figure 18. It consists of the row and column computation modules and two memory units, MEM1 and MEM2. The row module consists of two processors RP1 and RP2 along with a register file REG1, and the column module consists of two processors CP1 and CP2 along with a register file REG2. All the four processors RP1, RP2, CP1, CP2 in the proposed architecture consists of 2 adders, 1 multiplier and 1 shifter as shown in Figure 16. For the M2 architecture, RP1 and CP1 are predict filters and RP2 and CP2 are update filters.

Figure 19 illustrates the data access pattern for the (5,3) filter with $N=5$. RP1 calculates the *high-pass* (odd) elements along the rows, y_{01}, y_{03}, \dots , while

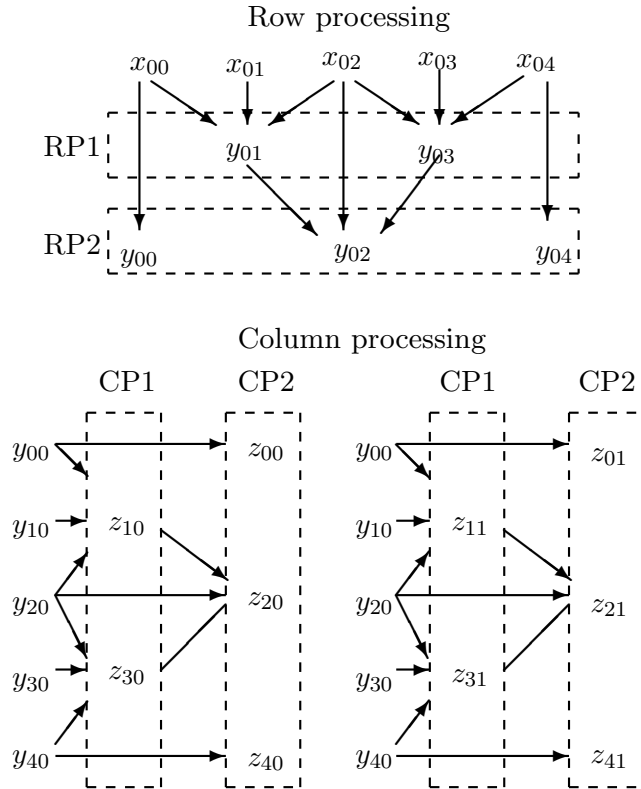


Figure 19: Data access patterns for the row and column modules for the (5,3) filter with $N=5$ in the 2D DWT architecture in [24].

RP2 calculates the *low-pass* (even) elements along the rows, $y_{00}, y_{02}, y_{04}, \dots$. CP1 calculates the *high-pass* and *low-pass* elements $z_{10}, z_{11}, \dots; z_{30}, z_{31}, \dots$ along odd rows and CP2 calculates *high-pass* and *low-pass* elements $z_{00}, z_{01}, \dots; z_{20}, z_{21}, \dots; z_{40}, z_{41}, \dots$ along the even rows. Note that CP1 and CP2 start computations as soon as the required elements are generated by RP1 and RP2.

The memory modules, MEM1 and MEM2, are both dual port with one read and one write port, and support two simultaneous accesses per cycle. MEM1 consists of two banks and MEM2 consists of four banks. The multi-bank structure increases the memory bandwidth and helps support highly

pipelined operation. Details of the memory organization and size, register file, and schedule for the overall architecture with specific details for each constituent filter have been included in [24].

The dataflow of the M4 architecture is quite different. Since this is a generalized architecture with the hardware in the row and column modules fixed, the computations span two passes. In the first pass, the row-wise computations are performed using both the modules. Module 1 reads the data from MEM1, executes the first two matrix multiplications, and writes the result into MEM2. Module 2 executes the next two matrix multiplications and writes the result into MEM1. In the second pass, the transform is computed along the columns. Once again, Module 1 executes the first two matrix multiplications and Module 2 executes the next two matrix multiplications.

5 VLSI Architectures for EBCOT

The block diagram of the VLSI architecture for the EBCOT encoder developed by *Andra, Chakrabarti, and Acharya* [12, 25, 26] is shown in Figure 20. Following are the key building blocks of this architecture.

1. Three separate blocks of combinational logic circuits generate the context and data pairs (cx, d) for the *zero coding* (ZC), *magnitude refinement coding* (MRC), and *sign coding* (SC) operations. The logic circuits for implementation of these operations and also the run-length coding (RLC) operation have been described in great detail in [2].
2. Five different special-purpose shift registers (σ -reg, η -reg, σ' -reg, ν -reg, and χ -reg) process the state variables σ , η , σ' , ν , and χ as described in the EBCOT algorithm. The sizes of shift registers to hold σ , η , σ' , ν , and χ state variables are 15, 8, 8, 4, and 12 bits respectively, as shown in Figure 20.
3. The local memory modules σ MEM, η MEM, and σ' MEM, each of size 32×4 bits, are used to load and store the three state variables σ , η , and σ' respectively. These three memory modules are updated in every pass by the corresponding σ , η , σ' registers. The magnitude (ν) and sign (χ) bits are read from the subband memory (SM) bit-plane by bit-plane. All the memories have a single read and write port as shown in Figure 20.

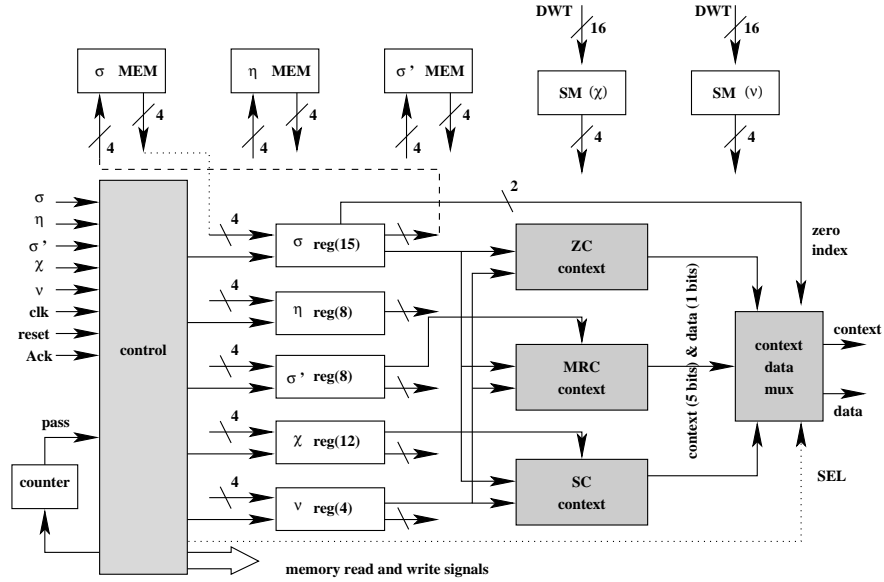


Figure 20: VLSI architecture for the EBCOT encoder.

4. The *context* and *data* multiplexer selects the right context (cx) and the corresponding data bit (d) generated by the ZC, MRC, SC, and RLC logic circuits. The possible context values from the RLC logic output are either 17 or 18 as explained in the standard. The data bit (d) is chosen from the v , sign data χ , hard coded RLC data bits (0,1), or the zero-index ZI(MSB, LSB) bits (00–11, because the run-length could be 0, 1, 2, or 3). The multiplexer (mux) is controlled by a 3-bit control signal ($cntrl_{cx}$). Based on the particular coding pass (CUP, SPP, MRP) executed, the controller generates the control signals $cntrl_{cx}$. The contexts and data for different $cntrl_{cx}$ are shown in Figure 21.
5. The *controller* is basically a state-machine that generates control signals to control the *shift registers* (σ -reg, η -reg, σ' -reg, v -reg, and χ -reg) and the *context* and *data* multiplexer (mux). The controller generates the read and write signals for the local

The computation blocks and registers in this architecture are controlled by a state machine. Combinational logic of the computation blocks, design

$cntrl_{cx}$	Context	Data
000	-	-
001	ZC	v
010	SC	sign bit
011	MC	v
100	17	0
101	17	1
110	18	ZI[MSB]
111	18	ZI[LSB]

Figure 21: Output of the *Context* and *Data* mux for different $cntrl_{cx}$.

of the registers and the control unit have been dealt in great length in [1]. A VLSI architecture for MQ-coder implementation has also been presented in great detail in [1].

5.1 Pass-Parallel Architecture for EBCOT

Chiang, Lin, and Hsieh [27] proposed a novel architecture for implementation of EBCOT in which the three coding passes of bit-plane coding process are merged into a single pass in order to improve overall systems performance. In this architecture, the authors proposed an efficient *pass-parallel context modeling* scheme in order to reduce the number of memory-access and clock-cycle requirements to implement EBCOT in hardware. The pass-parallel context modeling scheme processes the three coding passes of the same bit-plane in parallel by modifying the original bit-plane coding algorithm in a clever way by introducing two significance state variables σ_0 and σ_1 rather than single significance state variable σ in the original EBCOT algorithm.

In this algorithm, the coding operation in cleanup pass (CUP) is delayed by one stripe from the other two coding passes: significance propagation pass (SPP) and magnitude refinement pass (MRP). In the pass-parallel coding process, one of the two significance state variables $\sigma_0[m, n]$ and $\sigma_1[m, n]$ at location $[m, n]$ is toggled to 1 while this sample becomes significant in SPP and CUP respectively. Both the significant state variables are set to 1 immediately after the first magnitude refinement coding (MRC) is applied for this sample. Also the task of the magnitude refinement state variable σ'

is replaced by XOR (Exclusive OR) operation of the σ_0 and σ_1

$$\sigma'[m, n] = \sigma_0[m, n] \oplus \sigma_1[m, n] \quad (6)$$

where \oplus is the XOR operation. As a result, the on-chip memory requirement doesn't increase because of introduction of two significance state variables. The correct significant states of the samples within the context window are computed as follows.

- For samples belonging to SPP, the significance state of a visited sample at location $[m, n]$ is equal to $\sigma_0[m, n]$ and the significance state of the sample that has not been visited is

$$\sigma_{not} = \sigma_0[m, n] \vee \sigma_1[m, n] \quad (7)$$

where \vee is the binary OR operation.

- For samples belonging to the magnitude refinement pass (MRP), the significance state of the visited sample at location $[m, n]$ is equal to $\sigma_0[m, n]$. The significance state of the sample at location $[m, n]$ that has not been visited is determined by

$$\sigma_{not} = \sigma_0[m, n] \vee \sigma_1[m, n] \vee v^p[m, n] \quad (8)$$

where $v^p[m, n]$ is the magnitude of the data bit at location $[m, n]$ in p^{th} bit-plane of the code-block to be encoded. This is possible because a sample at location $[m, n]$ becomes significant if and only if its magnitude bit $v^p[m, n]$ is 1 for the first time.

- For samples belonging to the cleanup pass (CUP), the significance states of all neighbors are determined by Eq. 7.

Because of the pass-parallel context modeling, one MQ-coder module can be used in the JPEG2000 encoder architecture instead of three by switching the right context and data bits into the MQ-coder module [27]. Based on this pass-parallel coding of the bit-planes and single-pass switching arithmetic encoder, overall systems performance of a JPEG2000 architecture in terms of computation time can be improved by more than 25% [27].

5.2 Memory-Saving Architecture for EBCOT

Hsiao, Lin, Lee, and Jen [28] proposed an efficient high-speed memory savings architecture for implementation of the embedded bit-plane coding to improve the overall systems performance of a JPEG2000 encoder. In this architecture, mainly three speedup strategies have been applied in order to accelerate the context formation module in the EBCOT engine. They are pixel skipping, magnitude refinement parallelization, and group-of-columns skipping. Based on these speedup strategies, the on-chip memory for implementation of EBCOT can be reduced by approximately 20% [28].

The *renormalization* step in the MQ-coder has been enhanced in the code-string register to improve the clock rate of the MQ-coder implementation in [28]. Adopting these speedup strategies for both bit-plane coding and the MQ-coder, overall systems performance of the JPEG2000 encoder can be enhanced by reducing the clock cycles and memory requirements.

5.3 Enhanced EBCOT Architecture by Skipping

Lian, Chen, Chen, and Chen [29] recently proposed a very efficient hardware architecture for implementation of the EBCOT algorithm. Because of the characteristics of the fractional bit-plane coding by the EBCOT algorithm, distribution of the number of bits coded in three coding passes in EBCOT vary greatly from bit-plane to bit-plane. In the most significant bit-plane of the coding blocks all the samples are insignificant and only the *cleanup pass* (CUP) is executed. In the lower significant bit-planes, more and more bits are processed by the *magnitude refinement pass* (MRP), whereas the number of bits processed by CUP keeps on decreasing. The number of bits encoded by the *significant propagation pass* (SPP) increases at the beginning in first few bit-planes and then it decreases in lower significant bit-planes because the number of samples became significant in the previous bit-planes keeps on growing and the samples are encoded by the MRP in the following bit-planes. This skewed nature of distribution of the number of samples encoded in each coding pass has been exploited to develop an efficient architecture. As a result, experimentally it has been shown that sometimes it even reduces the number of clock cycles by almost 60% compared to the straightforward implementation of the EBCOT architecture.

In this architecture, the clocking requirements are reduced by applying two speedup techniques called *simple-skipping* (SS) and *group-of-columns skipping* (GOCS) to generate the context information. The basic principle

of both the methods is to skip the samples not belonging to a particular coding pass in order to avoid any computation. In each coding pass, it identifies the *need-to-be-coded* (NBC) samples so that the bits can be simply skipped. In the SS method only n cycles are spent to encode the NBC bits if there are only n NBC samples in a stripe. Since each stripe contains 4 samples, $4 - n$ clock cycles can be saved by skipping $4 - n$ bits which are not NBC. If a stripe does not contain any NBC, only one clock cycle is spent to check the condition only. The GOCS method is to skip a group of columns all together, if there are no NBC samples in the group of stripes (columns). The GOCS method can be applied in both MRP and CUP. The number of NBC samples in each group are marked in the SPP. There is a special memory in the architecture called the GOC memory to indicate the status of the samples in the group of stripes. The best number of columns in a group has been found to be 8 based on experimentation to study the performance of the GOCS method. If a group of stripes contains no NBC samples, the GOC memory is marked by a bit 0, otherwise it is marked by 1. While coding in MRP and CUP, the contents of the GOCS memory are checked. If the value is 0 for the currently coding GOC, all the stripes of the group can be skipped all together. As a result, only one clock cycle is required to check this condition and results in saving 31 clock cycles because a group of 8 stripes contain 32 samples.

An extreme case of skipping is called pass skipping. If the samples become significant in an earlier bit-plane, it is possible that all samples are encoded in SPP and MRP resulting in skipping the whole bit-plane for CUP. It is also possible that all samples in the lower bit-planes belong to MRP and none of them to SPP and CUP and hence skipping both the passes for the whole bit-plane all together. Although the occurrences of these cases are very small, it significantly speeds up the bit-coding when they happen.

6 Decoder Architecture for JPEG2000

The top-level architecture for the JPEG2000 decoder is shown in Figure 22. The decoder architecture is similar to the encoder architecture shown in Figure 7 with data flow in the reverse direction.

The *bitstream parsing module* parses the compressed file to generate the code-stream. The code-stream is decoded by the three MQ-decoders (BAC1, BAC2, BAC3) in order to generate the context and data pairs corresponding to each subband. The MQ-decoder architecture is very similar to the encoder

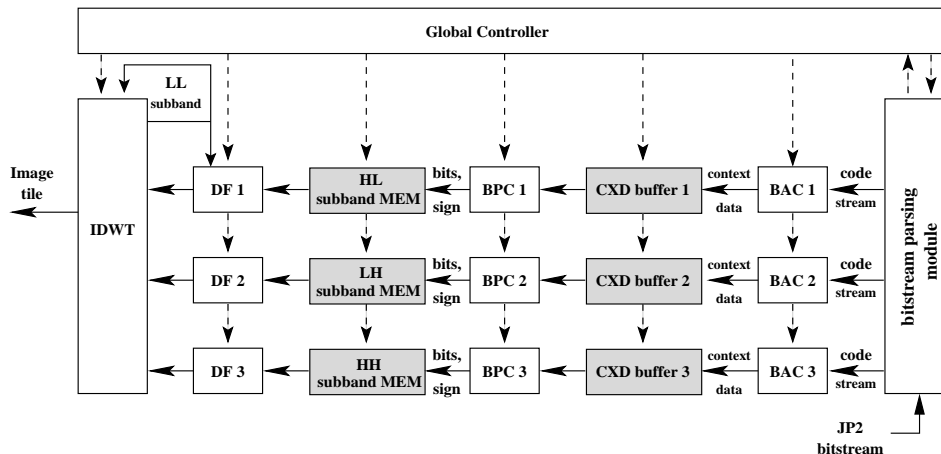


Figure 22: A top-level architecture for the JPEG2000 decoder.

architecture with few minor changes [1]. After the *context* and *data* are generated by the MQ-decoder, they are stored in the CXD buffers (CXD buffer1, CXD buffer2, and CXD buffer3). The EBCOT decoders (BPC1, BPC2, and BPC3) decode the *context* and *data* to generate the bit-planes of the code-blocks. The EBCOT decoder algorithm is essentially the same as the encoder algorithm with small and obvious changes in v and χ memories and registers. For example, in the EBCOT decoder the data from v (value) and χ (sign) are written into the subband memory instead of reading from it. We leave the details of the EBCOT architecture as an exercise for the reader. The *data format* engines (DF1, DF2 and DF3) convert these sign-magnitude values of the code-blocks into two's complement representation in order to be used by the inverse discrete wavelet transform (IDWT) architecture. The IDWT architecture generates the image tiles.

7 Conclusions

In this paper, we presented VLSI algorithms and architectures for implementation of the JPEG2000 standard for image compression. We described the algorithms for JPEG2000 standard for the core coding system. We discussed several key features of JPEG2000 standard desirable in many interactive multimedia applications. We presented a top-level systems architecture suitable for VLSI implementation of a JPEG2000 encoder. We reviewed

some of the key lifting architectures suitable for VLSI implementation of discrete wavelet transform which is a key component in JPEG2000 encoder. A special purpose architecture suitable for VLSI implementation of EBCOT algorithm has been reviewed. In addition, we also reviewed some efficient architectures in the literature which exploit the underlying data and computational parallelism inherent in the EBCOT algorithms.

References

- [1] T. Acharya and P. S. Tsai. *JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures*. John Wiley & Sons, Hoboken, New Jersey, 2004.
- [2] ISO/IEC 15444-1, “Information Technology – JPEG2000 Image Coding System – Part I: Core Coding System,” 2000.
- [3] ISO/IEC 15444-2, Final Committee Draft, “Information Technology—JPEG2000 Image Coding System—Part 2: Extensions,” 2000.
- [4] ISO/IEC 15444-3, “Information Technology—JPEG2000 Image Coding System—Part 3: Motion JPEG2000,” 2002.
- [5] ISO/IEC 15444-4, “Information Technology—JPEG2000 Image Coding System—Part 4: Conformance Testing,” 2002.
- [6] ISO/IEC 15444-5, “Information Technology—JPEG2000 Image Coding System—Part 5: Reference Software,” 2003.
- [7] ISO/IEC 15444-6, Final Committee Draft, “Information Technology—JPEG2000 Image Coding System—Part 6: Compound Image File Format,” 2001.
- [8] S. Mallat, “A theory for multiresolution signal decomposition: The Wavelet representation,” *IEEE Trans. Pattern Analysis And Machine Intelligence*, Vol. 11, no. 7, pp.674-693, July 1989.
- [9] D. S. Taubman, “High Performance Scalable Image Compression with EBCOT,” *IEEE Trans. on Image Processing*, Vol. 9, No. 7, pp. 1158-1170, July 2000.
- [10] ISO/IEC 14492-1, “Lossy/Lossless Coding of Bi-level Images,” 2000.

- [11] T. Kim, H. Kim, P.-S. Tsai, and T. Acharya, "Memory Efficient Progressive Rate-Distortion Algorithm for JPEG2000," *IEEE Transactions of Circuits and Systems for Video Technology*, Vol. 15, No. 1, pp. 181–187, January 2005.
- [12] K. Andra, C. Chakraborti, and T. Acharya, "A High Performance JPEG2000 Architecture," *IEEE Transactions of Circuits and Systems for Video Technology*, Vol. 13, No. 3, pp. 209–218, March 2003.
- [13] L. Liu, N. Chen, H. Meng, L. Zhang, Z. Wang, H. Chen, "A VLSI architecture of JPEG2000 encoder," *IEEE Journal of Solid-State Circuits*, Volume 39, Issue 11, pp. 2032- 2040, Nov. 2004.
- [14] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets," *Applied and Computational Harmonic Analysis*, Vol. 3, no. 15, pp.186-200, 1996.
- [15] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting schemes", *The J. of Fourier Analysis and Applications*, Vol. 4, 247-269, 1998.
- [16] T. Acharya and C. Chakrabarti, "A Survey on Lifting-Based Discrete Wavelet Transform Architectures," *The Journal of VLSI Signal Processing*, Vol. 42, No. 3, pp. 321-339, March 2006.
- [17] C. C. Liu, Y. H. Shiau, and J. M. Jou, "Design and implementation of a progressive image coding chip based on the lifted wavelet transform," *Proc. of the 11th VLSI Design/CAD Symposium*, Taiwan, August 2000.
- [18] J. M. Jou, Y. H. Shiau, and C. C. Liu, "Efficient VLSI Architectures for the Biorthogonal Wavelet Transform by Filter Bank and Lifting Scheme," *IEEE International Symposium on Circuits and Systems*, Sydney, Australia, pp. 529-533, May 2001.
- [19] C.J Lian, K. F. Chen, H. H. Chen, and L. G. Chen, "Lifting based discrete wavelet transform architecture for JPEG2000", *IEEE International Symposium on Circuits and Systems*, Sydney, Australia, pp. 445–448, May 2001.
- [20] C.T. Huang, P.C. Tseng, and L.G. Chen, "Flipping structure: an efficient VLSI architecture for lifting-based discrete wavelet transform," *IEEE Transactions on Signal Processing*, April 2004, pp. 1080-1089.

- [21] M. Vishwanath, "The recursive pyramid algorithm for the discrete wavelet transform," *IEEE Transactions on Signal Processing*, 42:673-676, March 1994.
- [22] H. Liao, M. K. Mandal, and B. F. Cockburn, "Novel architectures for lifting-based discrete wavelet transform", *Electronics Letters*, Vol. 38, Issue 18, pp. 1010-1012, Aug 29, 2002.
- [23] H. Liao, M. K. Mandal, and B. F. Cockburn, "Efficient Architectures for 1-D and 2-D Lifting-Based Wavelet Transform", *IEEE Transactions on Signal Processing*, Vol. 52, No 5, pp. 1315-1326, May 29, 2004.
- [24] K. Andra, C. Chakrabarti, and T. Acharya, "A VLSI Architecture for Lifting-Based Forward and Inverse Wavelet Transform," *IEEE Trans. of Signal Processing*, Vol. 50, No. 4, pp. 966-977, April 2002.
- [25] K. Andra, T. Acharya, and C. Chakraborti, "Efficient VLSI Implementation of Bit-plane Coder of JPEG2000," in *Proc. of the SPIE Intl. Symposium on Optical Science and Technology, Applications of Digital Image Processing XXIV*, Vol. 4472, pp. 246-257, San Diego, July 2001.
- [26] K. Andra, "Wavelet and Entropy Coding Accelerators for JPEG2000," Ph.D. Dissertation, Arizona State University, December 2001.
- [27] J. S. Chiang, Y. S. Lin, and C. Y. Hsieh, "Efficient Pass-Parallel for EBCOT in JPEG2000," *Proc. of the IEEE Intl. Symposium on Circuits and Systems (ISCAS 2002)*, pp. 773-776, Scottsdale, Arizona, May 2002.
- [28] Y. T. Hsiao, H. D. Lin, and C. W. Jen, "High-Speed Memory Saving Architecture for the Embedded Block Coding in JPEG2000," *Proc. of the IEEE Intl. Symposium on Circuits and Systems (ISCAS 2002)*, pp. 133-136, Scottsdale, Arizona, May 2002.
- [29] C. J. Lian, K. F. Chen, H. H. Chen, and L. G. Chen, "Analysis and Architecture Design of Block-coding Engine for EBCOT in JPEG 2000," *IEEE Transactions of Circuits and Systems for Video Technology*, Vol. 13, No. 3, pp. 219-230, March 2003.

Author's Biography: Dr. Tinku Acharya is the Chief Technology Officer and co-founder of Avisere Inc., Arizona, USA, and the Managing Director of its Indian subsidiary. He is also an Adjunct Professor in the Department of Electrical Engineering, Arizona State University, USA. Dr. Acharya received his B.Sc. (Honors) in Physics, B.Tech and M.Tech in Computer Science from the University of Calcutta, India and his Ph.D. in Computer Science from the University of Central Florida, Orlando, USA.

Currently, Dr. Acharya directs the technology team worldwide to develop core Intellectual Properties and products for Intelligent Video Analytics and its Computer Vision platform in Avisere Inc. Before Avisere, he served in Intel Corporation (1996–2002) to lead several R&D teams in numerous projects to develop algorithms and architectures in image and video processing, multimedia computing, PC-based digital camera, high-performance reprographics architecture for color photo-copiers, biometrics, multimedia architecture for 3G cellular mobile telephony, analysis of next-generation microprocessor architecture, etc. Before Intel, he was a consulting engineer at AT&T Bell Laboratories (1995–1996), a research faculty in the Institute of Systems Research, Institute of Advanced Computer Studies, University of Maryland at College Park (1994–1995), and held visiting faculty positions at Indian Institute of Technology (IIT), Kharagpur. He also served in National Informatics Center, Planning Commission, Government of India (1988–1990). He collaborated in research and development with Palo Alto Research Center (PARC) in Xerox Corporation, Eastman Kodak Corporation, and many other institutions worldwide.

Dr. Acharya is inventor of more than 100 US patents and 14 European patents in the areas of electronic imaging, data compression, multimedia computing, biometrics, color image processing, computer vision, and their VLSI architectures and algorithms. He contributed to over 70 technical papers published in international journals and conferences. He is author of four books - (i) *Image Processing: Principles and Applications* (Wiley, New Jersey, 2005), (ii) *JPEG2000 Standard for Image Compression: Concepts, Algorithms, and VLSI Architectures* (Wiley, 2004), (iii) *Information Technology: Principles and Applications* (Prentice-Hall India, New Delhi, 2004), and (iv) *Data Mining: Multimedia, Soft Computing and Bioinformatics* (Wiley, 2003). His pioneering works won him international acclamation. He has been awarded the Most Prolific Inventor in Intel Corporation Worldwide in 1999 and 2001, and Intel Corporation Arizona site for five consecutive years (1997–2001).

Dr. Acharya is a Fellow of Indian National Academy of Engineering (FNAE), Life Fellow of the Institution of Electronics and Telecommunication Engineers (FIETE), and Senior Member of IEEE. He served on the U.S. National Body of JPEG2000 standards committee (1998–2002). His current research interests are in computer vision for enterprise applications, biometrics, multimedia computing, multimedia data mining, and VLSI architectures and algorithms.