EFFICIENT ENCODER DESIGN FOR JPEG2000 EBCOT CONTEXT FORMATION

Chi-Chin Chang[1], Sau-Gee Chen[2] and Jui-Chiu Chiang[3]

[1]VIA Technologies, Inc.
Tappei, Taiwan
DouglasChang@via.com.tw

[2]Department of Electronic Engineering,
National Chiao Tung University,
Hsinchu, Taiwan
sgchen@cc.nctu.edu.tw

[3]Department of Electrical Engineering
National Chung Cheng University,
Chia-Yi, Taiwan
rachel@ee.cc.edu.edu.tw

ABSTRACT

JPEG2000 is the state-of-the-art compression standard. Its high performance is achieved by using EBCOT algorithm. In this paper, we present two methods to improve the computation efficiency, hardware utilization and area reduction for pass-parallel context formations (CF) of EBCOT. The first one called Sample-Parallel Pass-Type Decision (SPPD) method improves the performance in deciding the pass types of all four samples in the same column, while the second one called Column-Based Pass-Parallel Coding (CBPC) method codes all four samples in the same column concurrently. Simulation results based on TSMC CMOS 0.15μm process indicate that the CF architecture based on the proposed techniques reduces 13.83% of the encoding time, 18.28% of the hardware cost, and increase 34.78% of the hardware utilization, compared to the original pass-parallel CF.

1. INTRODUCTION

The block diagram of JPEG2000 encoder [1] is shown in Figure 1. In the encoding process, the source image is first processed by discrete wavelet transform followed by scalar quantization. Then the quantized coefficients are partitioned into code blocks which will be encoded by EBCOT algorithm [2-3] which exhibits high compression efficiency. EBCOT is a two-tiered architecture; Tier-1 is a context-based adaptive arithmetic coder, which is composed of a context formation (CF) engine and a binary arithmetic coder (BAC). Tier-2 is responsible for rate-distortion optimization and bit-stream formation. When realizing CF, generally it takes 5 memory blocks to encode each bit plane in three passes, where every sample of the bit plane will be coded in one of these passes without any overlapping with the other two passes. Since a context-decision pair (CX,D) is generated sample by sample and pass by pass, conventional CF architectures take a long time to encode samples for a code block. Consequently, numerous techniques [4-9] are devoted to developing efficient techniques for CF implementations.

The sample-skipping architecture [5] exploits the characteristics of CF algorithm and also the sample number coded in three coding passes in different bit plane to decide whether some samples can be skipped. It involves three strategies to accelerate CF computation, including pixel skipping (PS), magnitude refinement parallelization (MRP), and group-of-columns skipping (GOCS) Thus, the number of required clock cycles can be reduced up to 60%.

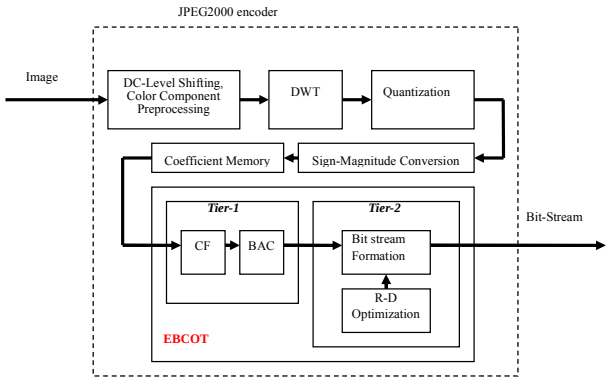

Figure 1. The block diagram of JPEG2000 encoder

The overhead introduced by GOCS is an additional memory block. To reduce the memory requirement, a memory-saving architecture [6] defines some new state variables so as to reduce the required on-chip memory by 20% approximately compared with that of [5].

The technique proposed in [10-11], is an improved high-speed design from the original CF algorithm. It is based on a new algorithm for deciding the coding pass of all the four samples in a column serially from the first to the fourth sample, only within one single clock cycle. But it results in a long critical path in deciding the coding pass for the fourth sample.

The pass-parallel architecture in [8-9] merges the involved three coding passes into a single pass, in order to improve the overall system performance and reduce memory requirement at the same time. Doing so, there is no need to generate context-decision, sample by sample and pass by pass. Moreover, context-decisions of all three coding passes for all the four samples in a column can be generated in one single clock cycle using column-based pass-parallel CF algorithm. As reported in [8], the overall system performance in terms of computation time can be improved by more than 25%. However, this scheme has two drawbacks: high hardware requirement and long critical path.

In this paper, an improved pass-parallel architecture is proposed, which can reduce the hardware requirement and ameliorate the computation performance. The organization of this paper is as follows. Section 2 reviews the CF scheme in EBCOT algorithm and gives a brief description of the pass-parallel CF technique. The proposed pass-parallel CF is in-

troduced in Section 3. The performance analysis and comparisons are presented in Section 4. Then, we will make a brief conclusion in the end.

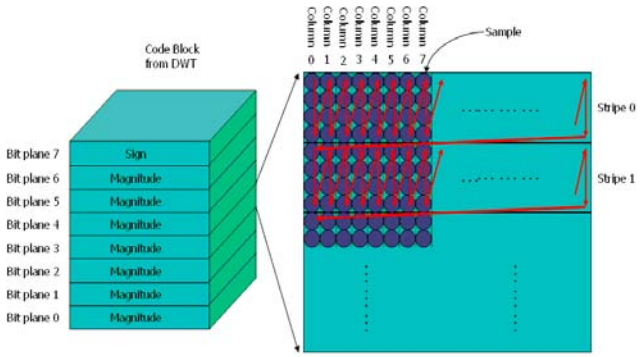

Figure 2. Code block, bit plane, stripe, column, and sample of EBCOT

2. CONTEXT FORMATION IN EBCOT ALGORITHM

In the encoding process, after the DWT and quantization operations, subband coefficients are partitioned into code blocks, where each block is a two-dimensional array which consists of integer wavelet coefficients with or without quantization, typically 64x64 or 32x32 in dimension. Then EBCOT algorithm [2] will be performed on these code blocks. The coefficients of each code block have to be expressed in sign-magnitude representations and divided into one sign bit plane and several magnitude bit planes. A bit plane is composed of many stripes; a stripe is composed of many columns, and a column is composed of four samples, as shown in Figure 2.

As shown in Figure 1, EBCOT consists of two tiers, where Tier-1 is a context-based adaptive arithmetic coder composed of a context formation (CF) engine and a binary arithmetic coder (BAC) and Tier-2 is in charge of rate-distortion optimization and bit-stream formation. First, each code block is coded by EBCOT Tier-1 CF module. CF generates context labels (CX) and decisions (D) for the binary arithmetic coder BAC to produce bit streams. After all code blocks are encoded, EBCOT Tier-2 picks up important bits according to the rate-distortion information from RD Optimization block to form the final output bit streams.

Each coding pass of a code block is scanned bit plane by bit plane, from the most significant bit plane (MSB) with at least a non-zero element to the least significant bit plane (LSB). In every bit plane, the scanning order is stripe by stripe from top to bottom. In every stripe, the scanning order is column by column from left to right. In every column, the scanning order is sample by sample from top to bottom. Each sample is coded by its context decision (CX, D) pair and sent to BAC. The (CX, D) pair of each sample is decided by the following primitivies:

1) Five coding states: sign value (χ), magnitude value (ν), significance state (σ), refinement state (ζ), and already-coded state (η)
2) Four coding primitives: zero coding (ZC), sign coding (SC), magnitude refinement coding (MRC), and run-length coding (RLC).
3) Three coding passes: significance propagation pass (SPP or Pass 1), magnitude refinement pass (MRP or Pass 2), and cleanup pass (CUP or Pass 3).

For more details for context-decision using these primitives, please refer to [3].

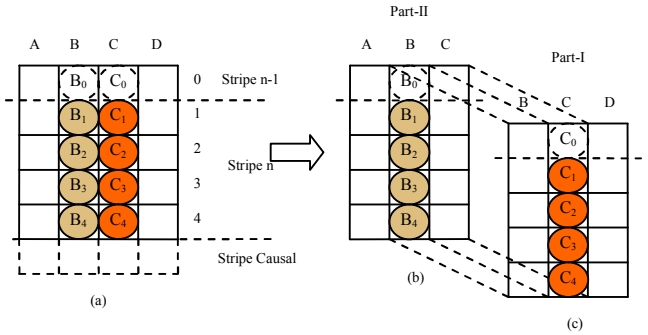

Figure 3. The pass-parallel coding windows of [8]. (a) The whole context window (b) Part-II window (c) Part-I window

2.1 Pass-parallel Context Formation

According to the CF flow [2-3], samples in a bit plane are coded pass by pass. However, each sample is coded by only one of the three coding passes; it implies that most of computation time is wasted. If the pass type of a sample can be determined and the sample can be coded simultaneously, the overall compression performance can be improved. Thus, [8] proposed a technique to process the three coding passes within the same bit-plane in parallel. Besides, a column-based operation [4] is adopted where the four samples in a column is coded one by one using the context formation engine. In addition, a "stripe casual" mode defined in JPEG 2000 [1] is used to eliminate the dependence between consecutive stripes.

To deal with wrong pass type decisions due to the dependence among these three coding passes, the context window employed in [8] consists of two parts: one for Pass 1 and Pass 2 decisions while the other for Pass 3 which is delayed by one stripe, as shown in Figure 3. Furthermore, they use 4 different state variables S_1, S_3, V_0, X_0 instead of those state variables described in the original EBCOT algorithm [3]. That is, two significance variables, significance state 1 (S_1) and significance state 3 (S_3) are introduced to replace three original significance state variable (σ), refinement state variable (ζ), and coded state variable (η), while variables V_0 and X_0 represent the magnitude and sign values, respectively. In this method, the required memory for a 64×64 code block is 4K bits less than that of a conventional design.

However, it has several disadvantages:

1) Huge coding operation requirement: context-decision pairs are decided in both parts of context window. It takes totally 23 coding operations (i.e., 11 coding operations for samples B_1 to B_4, 12 for samples C_1 to C_4).
2) High storage-element (SE) requirement: due to parallel processing of pass-type decision, it totally takes 68 SEs to store state variables S_1, S_3, V_0, and X_0.
3) Low pass type decision performance: it takes a long time to decide the pass type of sample C_4, because the signifi-

cant state and pass type of sample C_3 should be decided in advance. Thus it results in a long critical path.

3. PROPOSED PASS-PARALLEL ARCHITECTURE

In this paper we propose two methods to improve the coding performance of the pass-parallel CF while reducing the hardware requirement at the same time. The first one is called **Sample-Parallel Pass Type Decision (SPPD)** method which is composed of two novel checking and decision algorithms, namely, *Pass Type Checking* and *Fast Pass Type Decision* algorithms. The Pass Type Checking algorithm decides "false" pass types of all four consecutive samples in parallel. The Fast Pass Type Decision algorithm corrects these "false" pass types to the "true" ones by carefully analyzing the relationship between adjacent samples. The second one is called **Column-Based Pass-Parallel Coding (CBPC)** method. By delaying all coding passes by one column, i.e., to column B, 8 coding operations for context decisions generation are saved, thus the power consumption is also reduced.

3.1 Sample-Parallel Pass Type Decision

Consider the context window shown in Figure 3. In order to simultaneously decide the "false" pass type for each sample in column C in parallel, five types of state variables are used in Part-I window. The first three are classified as S_1, S_3 and V_0 which record the significance states for the samples coded in Pass 1, in Pass 3, and the magnitude bit of each sample in Pass 2, respectively. The remaining two state variables denoted as β_0, β_1 records the pass type state of each sample, where β_0, β_1 are defined below:

$$[\beta_1,\beta_0]=\begin{cases}[00], \text{ if sample belongs to Pass 3}\\ [01], \text{ if sample belongs to Pass 1}\\ [10], \text{ if sample belongs to Pass 2, not the first time}\\ [11], \text{ if sample belongs to Pass 2, first time}\end{cases}$$

Then the corresponding equations of pass type, significant states S_1 and S_3 for each sample can be derived from the state variables β_0, β_1 and V_0, as demonstrated in equations (1-2). Since samples belong to Pass 2 are already significant in the previous bit plane, it is not necessary to define the significance state for such samples.

State variables β_0 and β_1 are not only able to replace S_1 and S_3 of a sample, but also carry the pass type and MRC context information. Thus, with these state variables, there is no need to code samples in both Part-I and Part-II context windows. That means the same computation performance as that of [8] can be achieved, but only one suite of hardware is needed. Thus the power consumption is also reduced.

$$S_1(B_n)=\beta_1(B_n) \mid \beta_0(B_n) \,\&\, V_0(B_n) \quad (1)$$

$$S_3(B_n)=\beta_1(B_n) \mid \sim\beta_0(B_n) \,\&\, V_0(B_n) \quad (2)$$

$$S_1(B_n)\mid S_3(B_n)=\beta_1(B_n) \mid V_0(B_n) \quad (3)$$

After introducing β_0, β_1 and illustrating their relationship with the original state variables used in [8], the proposed SPPD algorithm can be actualized by the following three steps:

Table 1. False pass-type decision for sample C_{n+1}

Column B	Column C	Column D	row
$S_1(B_n)=1$ (eq. B_n)	$S_1(C_n)=1$ (eq. C_n)	$S_1(D_n)=1$ (eq. D_n)	n
$S_1(B_{n+1})=1$ (eq. B_{n+1})	$S_1(C_{n+1}) \mid S_3(C_{n+1})=0$ (eq. C_{n+1})	$S_1(D_{n+1}) \mid S_3(D_{n+1})=1$ (eq. D_{n+1})	n+1
$S_1(B_{n+2})=1$ (eq. B_{n+2})	$S_1(C_{n+2}) \mid S_3(C_{n+2})=1$ (eq. C_{n+2})	$S_1(D_{n+2}) \mid S_3(D_{n+2})=1$ (eq. D_{n+2})	n+2

3.1.1 Pass-Checking Algorithm

For all the 4 samples in column C, the equations listed in Table 1 will be used to decide their pass types simultaneously. The obtained pass types at this step are denoted as "false" pass type (FPT(C_1) to FPT(C_4)). Note that the equations in row *n*+2 will be set to zero for deciding pass type of C_4 due to stripe causal mode. Besides, FPT with significance state that denoted as FPTS(C_n) can be derived by FPT(C_n) & $V_0(C_n)$.

3.1.2 Pass type correction: case 1

For sample C_1, the pass type determined by FPT is correct. However, for samples C_2 to C_4, FPT values assigned by step 1 need some modifications. Let's take samples C_1 and C_2 for example: Suppose sample C_1 is decided as Pass 1 and significant while sample C_2 is decided as Pass 3 concurrently, using the equations in Table 1. Since sample C_1 belongs to Pass 1 and is significant, the pass type of sample C_2 should be changed from Pass 3 to Pass 1.

3.1.3 Pass type correction: case 2

If sample C_n is decided as Pass 1 and significant while sample C_{n+1} is decided as Pass 3 and significant, then we have to correct the pass type of sample C_{n+1} by changing the pass type from Pass 3 and significant to Pass 1 and significant, where n can be 1, 2, or 3.

Once the pass types of all the four samples in column C are decided, they are stored in the state variables β_0 and β_1. In the next clock cycle, these state variables will be used in Part-II window to perform coding operations. Note that samples in column B in the next clock cycle are identical to samples in column C in the current clock cycle.

3.2 Column-Based Pass-Parallel Coding

All the coding operations for all the four consecutive samples in the same column are performed in Part-II window. There are six types of state variables located in this window, which are classified as S_1, S_3, V_0, X_0, β_0 and β_1. Figure 4 shows the state variables used in Part-II window and how they are composed.

These state variables are used for coding operations, including ZC, SC, MRC, RLC, UC (uniform coding) operations as mentioned in Section 2. Samples in column B of Part-II window are coded concurrently by appropriate coding operations according to the pass types stored in the state variables β_0 and β_1. For example, if a sample is decided as Pass 1 and significant, i.e., $\sim\beta_1(B_n)$ & $\beta_0(B_n)$ & $V_0(B_n)=1$, ZC and SC are applied to encode this sample.

The pass type and significance state of a sample can be obtained from the following logic equations of β_0 and β_1 and V_0. Finally, appropriate coding operations can be employed.

$$\begin{cases} \text{pass 1}: \sim \beta_1(B_n) \,\&\, \beta_0(B_n) \\ \text{pass 1 and significant}: \sim \beta_1(B_n) \,\&\, \beta_0(B_n) \,\&\, V_0(B_n) \\ \text{pass 3}: \sim \beta_1(B_n) \,\&\, \sim \beta_0(B_n) \\ \text{pass 3 and significant}: \sim \beta_1(B_n) \,\&\, \sim \beta_0(B_n) \,\&\, V_0(B_n) \\ \text{pass 2}: \beta_1(B_n) \\ \text{pass 2 and first time}: \beta_1(B_n) \,\&\, \beta_0(B_n) \end{cases}$$

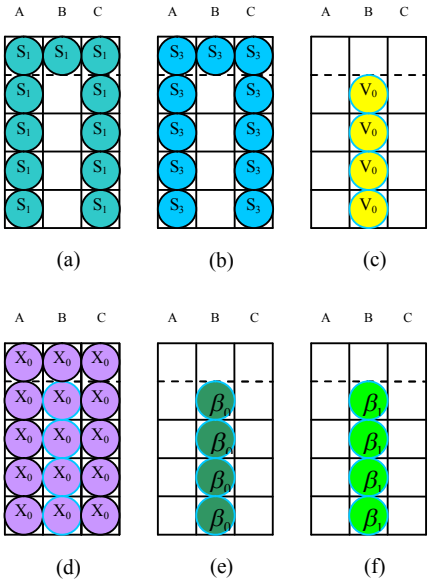

Figure 4. State variables used in Part-II window of the new "CBPC" algorithm (a)S_1, (b)S_3, (c)V_0, (d)X_0, (e) β_0, (f) β_1

4. PERFORMANCE ANALYSIS

With the proposed two new techniques, we are able to improve the computation performance, while reduce the hardware requirement, and enhance the hardware utilization of the original pass-parallel architecture, as verified below.

4.1 Memory Requirement

Since the proposed methods code all samples in Part-II window, the storage elements (SEs) that temporarily store state variables X_0 for column D in Part-I window can be saved. It totally takes 63 SEs, which is 5 SEs less than the original architecture [8]. According to the cell-library data sheet [12], at least 181.44 μm^2 of area size is saved.

4.2 Hardware Area

Since all samples in a column, no matter what pass type they belong to, are coded in parallel in Part-II window, there is no need to code samples in Part-I window. Thus only 15 coding operations are required. It saves 4 ZC and 4 SC operations in comparison to the original pass-parallel architecture. Table 2 shows the coding cells required for the original and proposed pass-parallel architectures, while Table 3 shows the area size and the power consumption of zero-coding and sign-coding cells, based on TSMC 0.15µm 1p7m process. As illustrated in Table 4, about 4055.12 μm^2 of area size is saved by applying the proposed architecture. Compared to the original CF module, it shows that about 18.28% of area size is reduced.

Table 2. Number of coding cells for samples C_1 to C_4 and B_1 to B_4 of the original and proposed pass-parallel architectures

Sample	Original Pass-Parallel		Total	Proposed Pass-Parallel		Total
C_1	ZC, SC, MRC		12	None		0
C_2	ZC, SC, MRC			None		
C_3	ZC, SC, MRC			None		
C_4	ZC, SC, MRC			None		
B_1	ZC, SC	RLC UC(1) *2	11	ZC, SC, MRC	RLC UC(1) *2	15
B_2	ZC, SC			ZC, SC, MRC		
B_3	ZC, SC			ZC, SC, MRC		
B_4	ZC, SC			ZC, SC, MRC		
(1) Uniform-Coding if all samples in a column belong to Pass 3						

Table 3. Area and power consumption analysis of zero-coding and sign-coding cells

	Area Size (μm^2)	Power Consumption (Watt)
ZC cell	711.37	3.667e-04
SC cell	302.41	1.826e-04
Cell Library: TSMC 0.15µm 1p7m standard cell library Circuit Compiler: Synopsys Design Compiler Version W-2004.12-SP3 for linux 72, medium effort, not ungrouped. Power Compiler: Synopsys PrimePower Version W-2004.12-SP2 for linux, Vector_Free_Power_Analysis average_power mode, clock period is 3ns		

Table 4. Area and power consumption analysis of original and proposed context formation architectures

	Area Size (μm^2)(1)	Area Size (μm^2)(2)	Power Consumption (Watt)(1)
Original CF	22182.12	217749.08	1.904e-03
Proposed CF	18127.31	213694.27	1.873e-03
(1) Area estimation without state variable memories. Cell Library: TSMC 0.15µm 1p7m standard cell library Circuit Compiler: Synopsys Design Compiler Version W-2004.12-SP3 for linux 72, medium effort, not ungrouped. Power Compiler: Synopsys PrimePower Version W-2004.12-SP2 for linux, Vector_Free_Power_Analysis average_power mode, clock period is 3ns (2) Area estimation with state variable memories (Four 1W1R1024x4 memories to store state variables ɣ, χ, σ1, and σ3 of a code block. 48891.72 μm^2 for each one, compiled from Virage® ts15d2p11rfsb07 Rev: 3.4.3 (build REL-3-4-3-2003-06-23))			

4.3 Hardware Utilization

From the previous discussion, both the original pass-parallel CF and the proposed CF code all 4 consecutive samples in one clock cycle, but there are some differences between them. First, the original CF codes samples in both Part-I and Part-II windows, but the proposed CF codes samples in Part-II window only. The former design takes 23 coding cells to code samples, and the latter one takes only 15 coding cells. As a result, the hardware (coding operations) utilization rate of the proposed CF is higher than the original CF (by about 34.78% improvement).

4.4 Execution Time

Both the original architecture and the proposed architecture decide pass types of all 4 consecutive samples in a column within one clock cycle. From the previous discussion, the original architecture takes a longer time to decide the pass type of sample C_4, but for the proposed method, the pass type of each sample is decided individually and concurrently, then a pass type adjustment algorithm called *Fast Pass Type Decision* is adopted. Therefore, the pass type decision path of

sample C_4 in the proposed architecture is shorter than the original one. From this result we know that the operation clock cycle of the proposed architecture is shorter than the original one, and thus a higher operation frequency can be achieved. The computation time of the proposed CF with and without applying the SPPD algorithm is illustrated in Table 5. The computation performance is improved by about 13.83%. Furthermore, Table 6 shows that the proposed methods outperform the other CF designs with a shortest critical path in deciding the pass type of the samples.

Table 5. Computation time of proposed CF with and without SPPD

	Fastest CLK (ns)	Maximum Frequency (MHz)
CF w/o SPPD	3.54	282.49
CF w/ SPPD	3.11	321.54
Cell Library: TSMC 0.15μm 1p7m standard cell library Environment: WCCOM, Process variable = 1.3, Voltage = 1.08V, Temperature=70ºC, worst_case_tree Circuit Compiler: Synopsys Design Compiler Version W-2004.12-SP3 for linux 72, medium effort, not ungrouped		

Table 6. Performance comparison in pass type decisions of various CF designs

Architecture	Proposed Pass-Parallel	Sample-Parallel [10], [11]	Pass-Parallel [9]	Original Pass-Parallel
Critical Path Delay (ns)	1.00	1.73	1.10	1.59

Note:
1. The area of coding cells of the Sample-Parallel architecture [10-11] is the same as the proposed Pass-Parallel architecture, but one extra code-state memory (η) is needed in the former.
2. The Sample-Parallel architecture [10-11] is sample-parallel, but not pass-parallel; the proposed architecture is both sample-parallel and pass-parallel, thus overall coding performance is improved.

5. CONCLUSION

In this paper, we present our research work on the design of the context formation module of EBCOT Tier-1 in JPEG2000. Since EBCOT Tier-1 coder exhibits high coding efficiency with high computational complexity cost, we propose an efficient architecture for its realization with small area and high speed by two techniques: Sample-paralleled pass type decision (SPPD) and column-based pass-parallel coding (CBPC) techniques. SPPD can reduce the processing time by 13.83%, while CBPC can save about 18.28% of hardware area, compared with the original pass-parallel method.

REFERENCES

[1] ISO/IEC JTC 1/SC 29/WG 1 N1684, JPEG2000 Part Ⅰ Final Committee Draft Version 1.0, March 2000.

[2] D. Taubman, "High Performance Scalable Image Compression with EBCOT," in *Proc. IEEE Int. cof. Image Processing*, Kobe, Japan, 1999, vol. 3, pp. 344-348.

[3] D. Taubman, "High Performance Scalable Image Compression with EBCOT," *IEEE Trans. on Image Processing*, vol. 9, issue 7, pp.1158 –1170, July 2000.

[4] K.-F. Chen, C.-J. Lian, H.-H. Chen and L. Chen, "Analysis and Architecture Design of EBCOT for JPEG 2000," in *Proc. IEEE Int. Symp. on Circuits and Systems*, vol. 2, pp. 765 –768, 2001.

[5] 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," in *Proc. IEEE Int. Symp. on Circuits and Systems*, vol. 13, pp. 219 -230, 2003.

[6] Y.-T. Hsiao, H.-D. Lin, K.-B. Lee and C.-W. Jen, "High-speed Memory-saving Architecture for the Embedded Block Coding in JPEG2000," in *Proc. IEEE Int. Symp. on Circuits and Systems*, pp. 122 -136, 2002.

[7] Y. Li, R. E. Aly, M. A. Bayoumi and S. A. Mashali, "Parallel High-Speed Architecture for EBCOT in JPEG2000," in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing,* vol. 2, pp.481-484, 2003.

[8] J.-S. Chiang, Y.-S. Lin and C.-Y. Hsieh, "Efficient Pass-Parallel Architecture for EBCOT in JPEG2000," in *Proc. IEEE Int. Symp. on Circuits and Systems*, vol. 1, pp. 773 -776, 2002.

[9] J.-S. Chiang, C.-H. Chang, Y.-S. Lin, C.-Y. Hsieh, and C.-H. Hsia, "High-Speed EBCOT with dual context-modeling coding architecture for JPEG2000," in *Proc. IEEE Int. Symp. on Circuits and Systems*, vol. 3, pp. 865 -868, 2004.

[10] A. K. Gupta, D. Taubman and S. Nooshabadi, "High Speed VLSI Architecture for Bit Plane Encoder of JPEG2000," in *Proc. IEEE Int. Symp. on Circuits and Systems*, vol. 2, pp. 233 -236, 2004.

[11] A. K. Gupta, D. Taubman and S. Nooshabadi, "Optimal 2 sub-bank memory architecture for Bit Plane Coder of JPEG2000," in *Proc. IEEE Int. Symp. on Circuits and Systems*, pp. 4373 -4376, 2005

[12] Artisan, TSMC 0.15um CL015LV process 1.2-Volt SAGE-X^{TM} V2.0 Standard Cell-Library Databook.