ANALYSIS OF EBCOT DECODING ALGORITHM AND ITS VLSI IMPLEMENTATION FOR JPEG 2000

Hong-Hui Chen, Chung-Jr Lian, Te-Hao Chang, and Liang-Gee Chen

DSP/IC Design Lab, Department of Electrical Engineering, and
Graduate Institute of Electronics Engineering, National Taiwan University
e-mail: { sernc, cjlian, thchang, lgchen } @video.ee.ntu.edu.tw

ABSTRACT

Embedded Block Coding with Optimized Truncation (EBCOT) is the entropy coding algorithm adopted by the new still image compression standard JPEG 2000. It is composed of a multi-pass fractional bit-plane context scanning along with an arithmetic coding procedure. GPP (general purpose processor) or DSP fails to accelerate this kind of bit-level operation, which is proven to occupy most of the computational time of the JPEG 2000 system. In this paper, two new accelerating schemes are proposed and applied to our prototyping design which turns out to be powerful enough to fulfill the demand of computational requirement of the most advanced digital still camera.

1. EBCOT DECODING ALGORITHM

EBCOT[1][2] decoding algorithm is one of the two key parts of the JPEG 2000[3] decoder. These two key parts are the transform part (inverse discrete wavelet transform) and entropy decoding part (EBCOT decoding algorithm). Figure 1 shows the block diagram of JPEG 2000 decoding system.

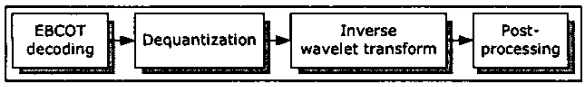

Figure 1. Block diagram of JPEG 2000 decoding system

The goal of this paper is to conduct a full analysis of the EBCOT decoding part and then presents a prototyping VLSI implementation of it. The EBCOT decoding algorithm is essentially a context-based arithmetic decoding algorithm, like that used in MPEG-4 shape coding, MPEG-4 texture coding, and JBIG/JBIG2 coding. This kind of algorithm is basically composed of two basic blocks, which are context formation and arithmetic decoder. The arithmetic decoder adopted in JPEG 2000 is the well-known MQ-coder[3][4] and we focus our discussion on the context formation part. EBCOT decoding algorithm acquires its context from the bit-plane scan with a stripe-by-stripe fashion and each bit-plane contributes three scan passes. These three scan passes are pass 1 (also called significant propagation pass), pass 2 (also called magnitude refinement pass), and pass 3 (also called clean-up pass). Figure 2 shows the basic bit-plane scan flow of the context formation operation for a code block. Note a stripe is formed by 4 rows of the bit samples within that bit-plane and typically a code block used in JPEG 2000 system is 64×64. Each bit sample is scanned/decoded at one of three passes. There are state variables kept for each bit sample and these state variables

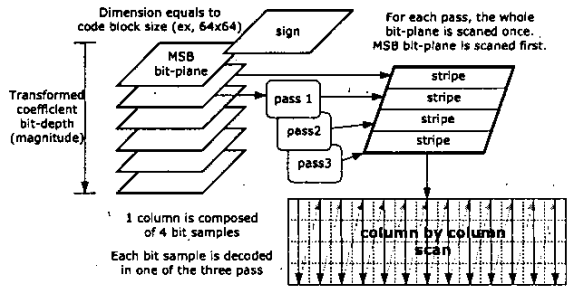

Figure 2. Bit-plane reconstruction scanning flow

Table 1. Criteria for determining the pass bit sample belongs to

	Criteria description
Pass 1	Any of the 8-connected neighborhood (Figure 3) has "Significant" state set and the bit sample itself has not become significant. After scanned, the "Coded" state will be set true.
Pass 2	Bit samples that have become significant at higher bit-plane. After scanned, the "Coded" state will be set true.
Pass 3	The bit samples that are not scanned at pass 1 or pass 2. That is, those samples with their "Coded" states remain false. After pass 3, all "Coded" states should be set false.

(note : "become significant" means that the first time non-zero bit sample of the coefficient's magnitude is found and the "Significant" state which is initially set false will be set true.)

decide which pass the bit sample should be scanned/decoded. These state variables are "Significant", "Coded", "Refined", and "Sign", and each state could be viewed as a one-bit flag. Table 1 lists the criteria for determining which pass a bit sample belongs to. At pass 1, the "Zero coding" primitive is used to determine the context of the bit sample according to the "Significant" states of the 8-connected neighborhood. In Figure 3, the 8-connected neighborhood may cross the stripe boundary. At pass 2, the context for the bit-sample is decided via "Magnitude refine coding" primitive which generates the context according to the "Significant" states of the neighborhood and the "Refined" states of its own. Pass 3 uses "Zero coding" primitive and also the "Run-length coding" primitive, which exams 4 bit samples (a column) together to see if all the bits have non-significant neighborhood and none of them are scanned at pass 1 or pass 2. If true, the run-length context is generated. However, if any of the sample in that column is not zero, the position of this

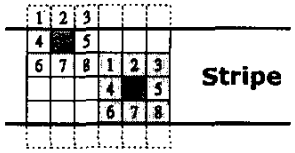

Figure 3. 8-connected neighborhood

Table 2. JPEG 2000 run-time profile

Operation	Runtime percentage (%)	Renormalized (%)
Context formation	34.6	50.5
Arithmetic decoder	7.5	10.9
Inverse quantization	2.7	3.9
IDWT	23.8	34.7
Total	68.5	100

(Environment: Windows 2000, 256M RAM, PIII 866, Microsoft Visual C++ 6.0 Target file : lena_97_L3.jp2 (encoded from 256x256 test image with 3-level DWT, compression ration=2.4. Note some computation is occupied by IO or memory manipulation)

bit should be sent through specific context and the rest bits are coded by "Zero coding" primitive. The sign of the coefficient is decoded immediately after the coefficient becomes significant using "Sign coding" primitive, which emits the context according to the "Significant" and "Sign" states of the 8-connected neighborhood. This sign decoding operation can happen at pass 1 or pass 3.

To sum up, all the coding primitives check states of the neighborhood and the decoded bit sample itself with a way like table looking-up method to determine the corresponding context for arithmetic decoder. For a more detailed description or the context tables used in JPEG 2000, one can refer to [3]. Finally, if any decision is generated from the arithmetic decoder which is provided with the context and bitstream of the decoding code block, the value of the decision will then be converted to bits of the magnitude or sign for the coefficient, or just a judgment for "Run-length coding" all depending on the coding primitive used. And these decisions will affect the state variables of the decoded coefficient that may force the coefficient to be decoded at different pass in the decoding procedure at the next bit-plane.

2. PROPOSED SKIPPING-BASED DECODING FLOW AND SKIPPING PRIMITIVES

According to the JPEG 2000 VM software profile result as shown in Table 2, the EBCOT decoding algorithm occupies most of the computational time of the JPEG 2000 decoding system. Referring to our previous work [5], it is apparently that if the architecture is not carefully designed, that is if we check all the bit samples at every location for each scan pass, clock bubbles will be generated. We list several conditions that result in clock bubbles in Table 3. Theoretically speaking, assume the bit-depth of the coefficient magnitude is "Bdep", a 3-pass scan algorithm for a code block with size NxN will cost 3xNxNxBdep clocks to complete the decoding procedure while the real decoded samples are N xNxBdep. Hence, there are 2xNxNxBdep waste clocks and the portion of extra clocks is so high as to 67 %. Our previous work [5] introduces the idea of skipping scan method with two major skipping primitives as Sample Skipping (SS) and Group -of-column Skipping (GOCS) to reduce the clock bubbles. To achieve these two skipping primitives, a proper memory arrangement is also proposed. In this paper, we propose two other

Table 3. Conditions that result in clock bubbles

Coding pass	Conditions
Pass 1	The coefficient has become significant and the bit sample should be decoded at pass 2.
	The coefficient has not become significant yet, but there is not any coefficient in its 8-connected neighborhood becomes significant. So no significant propagation for this coefficient and the bit sample should be decoded at pass 3.
Pass 2	The coefficient has not become significant in the previous bit-plane, so the bit-sample should not be decoded/refined at pass 2.
Pass 3	The bit-sample with its "Coded" states set true.

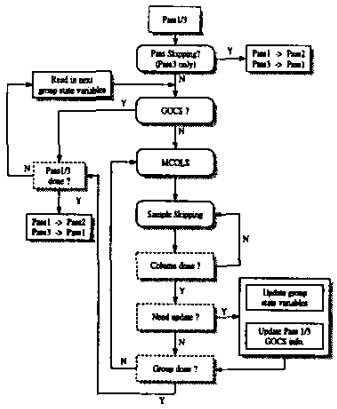

Figure. 4(a) Scan flow of pass 1 and pass 3

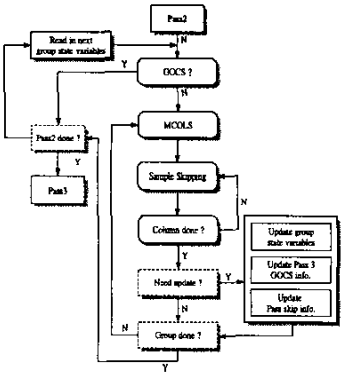

Figure. 4(b) Scan flow of pass 2

skipping primitives called Multiple Column Skipping (MCOLS) and Pass Skipping (PxS). Latter it can be shown that these two primitives can be implemented with a simple modification of the memory organization and with a very little hardware cost. The skipping ability or bubble reduction performance is even better than our previous work [5]. Before the detailed analysis of each skipping primitive, a complete skipping scan flow of each pass is depicted in Figure 4. The following is the brief description for each skipping primitive.

2.1 Sample Skipping

SS primitive is applied to each column, like the column

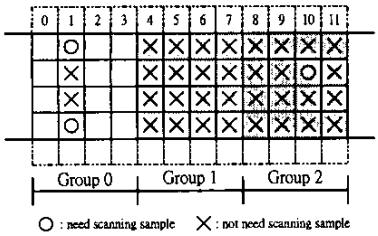

Figure 5. Skipping primitive illustrating example

numbered 1 shown in Figure 5. The bit samples need to be scanned are the first and the fourth. With a direct scan method while scanning the second and the third sample, no valid context will be generated because these two samples do not belong to this scan pass and a total scan clock counts for this column is 4. If SS is applied, the sample that contributes no valid context will be skipped, and only 2 clocks are spent for the scan of column 1.

2.2 Group of Column Skipping

As shown in Figure 5, there may be several consecutive columns that do not need to be scanned. Hence, if we pre-collect the information about the skipping information for the group and record it in memory, we can check the group skipping flag (1 bit per group) first to skip the whole group. This primitive need an additional memory to record all the group skipping flags.

2.3 Multiple Column Skipping

The problem of the GOCS is that it is not flexible enough. Look at the group 2 within Figure 5. The GOCS will fail to skip this group even that this group only has one bit sample to be scanned. With a simple modification of the memory organization, which is to enlarge the vision of the state variables, we can skip multiple columns by checking more state variables. For example, if the memory can output the state variables 4 columns per read operation, then the information will be enough to determine whether multiple columns can be skipped or not. If MCOLS with a vision of state variables up to 4 columns is applied, the group 1 can still be skipped (but no need of the GOCS flag from memory) and group 2 will cost only 1 clock (assume SS is applied) to complete the scan of the group while GOCS primitive fails and only SS works that will cost 4 clocks to complete the same scan procedure.

2.4 Pass Skipping

Every time after pass 1 or pass 3 has completed its scan procedure, there is a chance that all the "Significant" state variables of the whole coefficients have been set true. Then the following scan pass will all be the pass 2 only. The whole pass 1 and pass 3 can just be skipped. Only 1 bit flag is needed to complete this check. However, this situation almost never happens because there always exists coefficient equal to zero and never becomes significant and needs to be decoded at pass 1. As for pass 3, the "Significant" and "Coded" states can be used to decide whether the whole pass 3 is skipped and this flag should be evaluated at pass 2 scan procedure. Table 4 shows our evaluation of the Pass Skipping primitive and it shows that under lossless compression, many pass 3 scanning operations can just be skipped because they do not contribute any valid context.

Table 4. Evaluation of Pass Skipping primitive

	Lossless compression		Compression ratio=8 (1bpp)	
Test image	Total passes	Pass 3 skip counts	Total passes	Pass 3 skip counts
Lena	301	7	124	0
Shuttle	352	17	124	0

(Test image: gray level, 256x256 with 2-level DWT, results in 16 code blocks)

Table 5. Evaluation of combinations of skipping primitives

	Lena lossless	Lena CR=8	Shuttle lossless	Shuttle CR=8
GOCS_4	951430	331353	1104643	329050
GOCS_8	958301	333849	1111638	331059
GOCS_16	971629	339519	1123989	334193
SS+GOCS_4+PxS	481701	160635	556901	164172
SS+GOCS_8+PxS	483905	161106	560424	164559
SS+GOCS_16+PxS	486891	162186	563864	165153
MCOLS_4	761568	278152	887071	272326
SS	521360	175285	609467	179985
SS+MCOLS_4+PxS	441391	137490	514905	142240
SS+MCOLS_4+GOCS_8+PxS	436040	135106	509085	139653
SS+MCOLS_4+GOCS_16+PxS	433999	134184	506763	138508
Ideal	378726	109524	448647	114653

(Note: The postfix of GOCS means the number of columns grouped together and that of MCOLS means the memory output vision of columns) (Unit: clock counts)

Table 5 lists our evaluation of the different combinations of the skipping primitives with the model written by C programming language. The numbers are the total cycle counts needed to complete performing the EBCOT decoding part of the test image. The ideal cycle counts are equal to the total context counts based on the assumption that arithmetic decoder can consume 1 context per cycle and this is usually true for the sequential coding algorithm like the arithmetic decoding algorithm. For applying a single skipping primitive only, SS owns the best ability to reduce the clock bubbles, and the next are MCOLS and GOCS. For applying multiple skipping primitives, the SS + MCOLS + PxS + "long-term GOCS" performs best. The "long-term" phase means the number of column grouped together must be larger than the vision of the new memory organization for the MCOLS. We only list MCOLS_4 because from the evaluation of GOCS, the GOCS_4 performs best among all other grouping scheme. It is worth noticing that PxS will only be useful for the lossless compression for these two patterns according to the evaluation result of Table 4.

3. VLSI IMPLEMENTATION AND EXPERIMENTAL RESULT

From the evaluation result of Table 5, we select the combination, SS+MCOLS_4, applied to our prototyping JPEG 2000 block decoder design. The reasons are listed below:

1. The SS+MCOLS_4 has already possessed a very good performance.
2. If GOCS is applied, there must be an additional memory that will increase the die size and the extra gain is small.
3. PxS is skipped because the decoder will normally decoding the bistreams at the compression ratio about 5~15 or higher. PxS tends to happen at compression ratio less than 3.

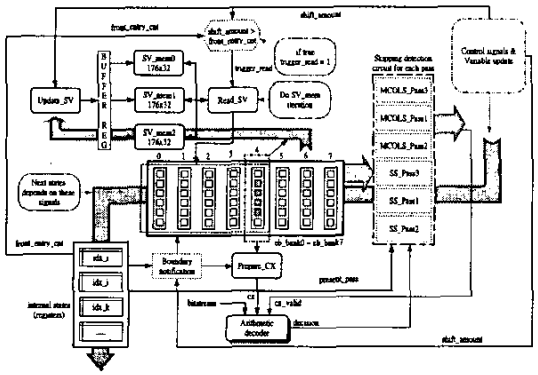

Figure 6. Proposed architecture (MCOLS+SS)

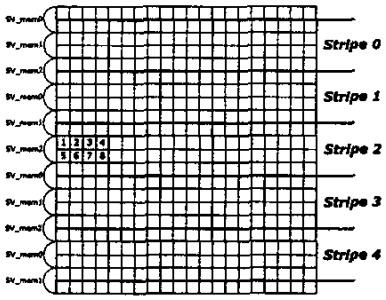

Figure 7. State memory mapping for the bit-plane

The proposed architecture is depicted in Figure 6. Three state variable memories are used to accomplish the memory iteration as proposed by our previous work [5] and memory IO capacity is increased to 32 bits. The organization of the memory can be illustrated with Figure 7. Every time when new state variables must be read out, 8 state variable groups are outputted. Each group is composed of 4 bits as "Significant", "Sign", "Refined", and "Coded" states. Hence, the IO bit-width must be 8×4=32 bits for each state variable memory. These state variables are latched into the register banks eb_bank0 – eb_bank3. In this prototyping design, we only concentrate our context determining PEs (table look-up mechanism for each position at eb_bank4 according to the state variables of the 8-connected neighborhood) at eb_bank4 only. That is, if the situation like group 2 of Figure 5 is encountered, the eb_bank must first shift two columns and make the bit sample which needs to be decoded be aligned at eb_bank4. The shift-two-column operation introduces 1 clock bubble, so the performance suffers from a little degradation. However, this simplification will reduce the hardware cost drastically and the performance turns out to be still good enough to fulfill our specification. The skipping detection circuit detects the amount of skipping through MCOLS at column domain first, then through SS at sample domain within a column. It is inevitable that there are invalid contexts and skipping circuit will notify the arithmetic decoder if the context is valid or not. If the context is valid, the arithmetic decoder will also output a valid decision, which turns to affect the skipping amount of eb_bank and the next states of internal registers. After enough columns have been scanned, the updating operation is triggered with the updating content according to eb_bank4 – eb_bank7. The experimental result of this prototyping design is shown in Table 6. In Table 6, the bubble rate means the clock counts that contribute no valid context divide

Table 6. Experimental result with Verilog model

Test image	Ideal clock	Spend clock	Bubble rate	Residue rate
Lena	378726	435113	14.89%	35.29%
Shuttle	448647	516022	15.02%	35.79%

(Lossless compression, image size: 256x256, 3-level DWT)

by the ideal clock counts. The column titled "Residue" is the clock counts spent divided by non-optimal direct 3-pass scan which is the evaluating figure used in our previous work [5]. According to [5], the encoder design using SS and GOCS skipping primitives together results in a residue rate about 40% while the decoder design presented in this paper outperforms it about 5% (from 40% to 35%). Our design is modeled by Verilog hardware description language and synthesized with a 0.35 μm cell library for TSMC 1P4M process technology. The target working frequency is 33MHz and total gate counts are about 22,000. As a simple evaluation, we assume the bitstream is compressed with the ratio equal to 5 (the compression ratio of finest quality in general digital still image camera) and Mpix is the capable million pixel counts of the input stream our design can deal with. From Equation (1) it can be calculated that our design owns the ability to handle the image about 5.73 million pixels just in 1 second.

$$\frac{Mpix \times Bit\ Depth \times Residue\ Pass \times Component}{Working\ Frequency \times Compression\ Ratio} = Time \quad (1)$$

$$\frac{Mpix \times 9 \times (3 \times 35.54\%) \times 3(RGB)}{33(MHz) \times 5} = 1\ second,\ Mpix = 5.73$$

(Average bit depth is 9, and 35.54% is the average value from Table 6)

4. CONCLUSION

Skipping based algorithm with all its useful skipping primitives is an efficient scheme to reduce the scan bubble while applied to the implementation of EBCOT coding algorithm which is a core entropy coding algorithm adopted in JPEG 2000. In this paper, two new skipping primitives MCOLS and PxS are introduced. A prototyping architecture implemented with SS and MCOLS skipping primitives is proposed and the skipping ability is better than previous work [5]. The proposed architecture could handle up to 5.73 million pixel color images in one second at 33MHz, thus can fulfill the demand of decoding power of most advanced digital still camera while JPEG 2000 is embedded as the image compression kernel.

REFERENCES

[1] D. Taubman, "EBCOT: Embedded Block Coding with Optimized Truncation," ISO/IEC JTC1/SC29/WG1 N1020R.

[2] D. Taubman, "High Performance Scalable Image Compression With EBCOT," *IEEE Trans. Image Processing*, vol. 9, Page(s): 1158 –1170, Jul. 2000.

[3] JPEG 2000 Part I: Final Draft International Standard (ISO/IEC FDIS15444-1, ISO/IEC JTC1/SC29/WG1 N1855, Aug. 2000.

[4] M.J. Slattery and J.L. Mitchell, "The Qx-coder," *IBM Journal of Research and Development*, vol.42, No. 6. 1998.

[5] K.-F. Chen, C.-J. Lian, H.-H. Chen and L.-G. Chen, "Analysis and Architecture Design of EBCOT for JPEG 2000," *IEEE International Symposium on Circuits and Systems*, vol. 2, Page(s): 765–768, 2001.