VLSI Architecture of Arithmetic Coder Used in SPIHT

Kai Liu, Evgeniy Belyaev, and Jie Guo

***Abstract*—A high-throughput memory-efficient arithmetic coder architecture for the set partitioning in hierarchical trees (SPIHT) image compression is proposed based on a simple context model in this paper. The architecture benefits from various optimizations performed at different levels of arithmetic coding from higher algorithm abstraction to lower circuits' implementations. First, the complex context model used by software is mitigated by designing a simple context model, which just uses the brother nodes' states in the coding zerotree of SPIHT to form context symbols for the arithmetic coding. The simple context model results in a regular access pattern during reading the wavelet transform coefficients, which is convenient to the hardware implementation, but at a cost of slight performance loss. Second, in order to avoid rescanning the wavelet transform coefficients, a breadth first search SPIHT without lists algorithm is used instead of SPIHT with lists algorithm. Especially, the coding bit-planes of each zero tree are processed in parallel. Third, an out-of-order execution mechanism for different types of context is proposed that can allocate the context symbol to the idle arithmetic coding core with a different order that of the input. For the balance of the input rate of the wavelet coefficients, eight arithmetic coders are replicated in the compression system. And in one arithmetic coder, there exists four cores to process different contexts. Fourth, several dedicated circuits are designed to further improve the throughput of the architecture. The common bit detection (CBD) circuit is used for unrolling the renormalization stage of the arithmetic coding. The carry look-ahead adder (CLA) and fast multiplier-divider are also employed to shorten the critical path in the architecture. Moreover, an adaptive clock switch mechanism can stop some invalid bit-planes' clock for the power saving purpose according to the input images. Experimental results demonstrate that the proposed architecture attains a throughput of 902.464 Mb/s at its maximum and achieves savings of 20.08% in power consumption over full bit-planes coding scheme based on field-programmable gate arrays (FPGAs).**

***Index Terms*—Arithmetic coding, common bit detection (CBD) circuit, context model, out-of-order execution, set partitioning in hierarchical trees (SPIHT), VLSI arithmetic coder architecture.**

Manuscript received May 14, 2010; revised September 14, 2010; accepted January 19, 2011. Date of publication February 24, 2011; date of current version March 12, 2012. This work was supported in part by the National Natural Science Foundation of China under Grant 60802076, by the Fundamental Research Funds for the Central Universities under Grant JY10000903003, and by the Open Research Funds of State Key Laboratory for novel software technology under Grant KFKT2010B28. This work was done in part at the State Key Laboratory for Novel Software Technology, Nanjing University, China.

K. Liu is with the Computer School, Xidian University, Shaanxi 710071, China (e-mail: kailiu@mail.xidian.edu.cn).

E. Belyaev is with Saint-Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences, Saint-Petersburg 199178, Russian Federation (e-mail: e_beliaev@mail.ru).

J. Guo is with the Communication School, Xidian University, Xi'an, Shaanxi 710071, China (e-mail: jguo@mail.xidian.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2011.2109068

I. Introduction

As arithmetic coding (AC) [1], [2] method can obtain optimal performance for its ability to generate codes with fractional bits, it is widely used by various image compression algorithms, such as the QM in JPEG [3], the MQ in JPEG2000 [4]–[6], and the context-based adaptive binary arithmetic coder (CABAC) [7], [8] in H.264. Especially, the set partitioning in hierarchical trees (SPIHT) [9] uses an AC method to improve its peak signal-to-noise ratio (PSNR) about 0.5 dB. Although the theory and program code of AC are mature, the complicated internal operations of AC limit its application for some real time fields, such as satellite image and high speed camera image compressions. In order to achieve performance gains, high speed architecture of AC in compression scenarios must be designed to meet the throughput requirement.

Thus both industrial and academic research groups have put their efforts to AC hardware architectures for various image compression systems. However, there are two main challenges in hardware architecture design for high speed applications. One is data dependencies in AC which require the result of iteration i before next run can commence during the adaptive model update and internal loops. The other one is that AC requires increasingly greater precision as more data arrive. In order to deal with such difficulties, several architectures are proposed in the past years.

Wiseman [10] proposed a systolic hardware architecture for a quasi AC which is a simple version of AC. In [10], the architecture uses a pipeline processing to compute each stage of AC, which eliminates an internal high frequency clock and utilizes a fast lookup table for state transitions. Although the architecture can improve the speed of the internal operations, such as the probability interval update and cumulative calculations, it cannot offer supports for multi-contexts AC processing in image compression fields. For the QM coder in JPEG, Andra's [11] gave a new architecture which decreases operations for the more probable symbol (MPS) and used a non-overlap window style for the speedup purpose. In [11], the probability interval partition is accelerated by exchange of the less probable symbol (LPS) interval with the MPS interval. Thus the amount of operations is reduced by 60%–70% compared with other coders. Another highlight of [11] is the non-overlap window that is applied to the continuous MPS in order to simplify renormalization operation. Due to simple operations in Andra's coder, the performance is slightly lowered by 1%–3%.

In SPIHT algorithm aspect, many researchers proposed various modifications to improve performance of SPIHT. Some algorithms aim for better PSNR values, which do not concern

1063-8210/$26.00 © 2011 IEEE

about hardware issues. Kassim [12] introduced a method for selecting an optimal wavelet packet transform (WPT) basis for SPIHT, which efficiently compacts the high-frequency subband energy into as few trees as possible and avoids parental conflicts. Their proposed SPIHT-WPT coder achieved improved coding gains for highly textured images. Ansari [13] proposed a context based SPIHT (CSPIHT) method, which used a segmentation and interactive method for selecting the contextual region of interest mask to achieve a better performance results in medical images. In order to reduce memory and speed up SPIHT software, Akter [14] used one list to store the coordinates of wavelet coefficients instead of three lists of SPIHT and merged the sorting pass with the refinement pass together as one scan pass. On the other side, Wheeler [15] proposed a modified SPIHT algorithm which does not use lists. Because of no insert and search operations for list, the speed of algorithm can be improved greatly.

In SPIHT implementation area, corresponding architectures are mainly designed for SPIHT algorithm without AC. Memory bands for storing the wavelet coefficients are used in SPIHT coder [16], [17]. But for some large width images, for example, satellite images, it is difficult to integrate many memories on board in the memory bands architectures. Fry and Hauck [18] realized a configurable SPIHT coder with field programmable gate array (FPGA) devices, which can reach a throughput of 244 Mpixels/s. Huang [19] gave his SPIHT implementation architecture which modified the coefficient scanning process of SPIHT and used a 1-D addressing method for the wavelet coefficients. The throughput of [19] can be 30 frames per second with CIF images. Ritter [20] proposed an SPIHT coder with reduced access to random memory. Pan [21] proposed a listless modified SPIHT which reduced memory in hardware architecture. Unfortunately, all of these architectures mentioned above did not give a detailed AC part description in their coders. However for some critical applications, an AC part in SPIHT is nontrivial in order to further improve the performance. Therefore it is necessary to design a high speed AC architecture for SPIHT.

As far as architecture is concerned, there are three stages during implementation. The first step for designing an AC in SPIHT is to set a context model suitable for hardware processing. One of context model is designed in the QccPack-SPIHT by Fowler [22]. In the architecture, a simple context model based on the QccPackSPIHT software is designed, which just exploits the relationship of nodes in one zerotree and establishes four types of context for current position value, current position sign, descendant set (D set) and grant descendant set (L set). The second step is to remove the internal loops of AC and arrange different modules for hardware. The last step is to connect all modules by different paths to build one AC. The main contributions of this architecture can be summarized as follows.

1) A simple context state model which is based on the neighbor pixels' significant states is designed for hardware implementation.
   In order to achieve high speed architecture, we adopt a fixed breadth first search scan order for SPIHT coding instead of variable scan order to avoid rescanning the wavelet coefficients. Based on this scanning order, we design a simple context model which just uses the brother nodes' states in the coding tree for the fast processing purpose. The degradation of performance in PSNR values compared with the QccPackSPIHT context model is slight through the experimental results.
2) According to the context model, different context symbols formed by SPIHT algorithm are processed in parallel by the arithmetic coder for the speedup purpose.
   In order to improve the throughput of our arithmetic coder, we utilize two methods to remove the bottlenecks in the whole image coding process. One method is a bit-plane parallel scheme for all wavelet coefficient bit-planes, which changes the process order of bit-plane from sequential to parallel manner. The other important way is an out of order mechanism for the execution in the arithmetic coder. By the out of order execution for multiple contexts, the coding speed can be accelerated greatly. Thus the architecture is able to consume multiple input symbols in one clock cycle.
3) In order to reduce memory size, an internal memory array is used for the cumulative probability values. Carry look-ahead adder (CLA) circuits are employed for the update of probability variables.
   Since the architecture uses an FPGA platform as prototype, plenty of flip-flops in the device can be used for the memory array instead of external RAM structure. Therefore the access time can be improved significantly for the internal memory array. At the same time, the critical path is shortened by the advanced calculation circuit structure, such as the CLA and fast multiplier-divider.
4) For power efficient design, a dedicated adaptive power management module is used to stop clocks for the invalid bit-plane, which contains no information about the wavelet coefficients. And the memory access pattern is also compacted for power saving purpose.

According to SPIHT, the number of bit-planes used for representing the wavelet coefficients varies with different images. The rich content images may use more bits to represent the wavelet coefficients. On the other hand, the poor content images may require fewer bits to represent these coefficients. Then, we can stop some bit-plane coders by cutting its input clock according to the maximal coefficient in wavelet domain adaptively. We also analyze the pattern of memory access and optimize the memory behavior in the architecture to reduce power consumption.

The rest of this paper is organized as follows. Section II provides an overview of AC and SPIHT algorithms. The challenges for SPIHT arithmetic coding architecture are also shown in that section. In Section III, after the bit-plane parallel SPIHT with breadth first search is described, the proposed context symbol model based on the wavelet coefficient's magnitude and sign characteristic is described in detail for architecture design. The architecture of arithmetic coder and the whole SPIHT coding structure are given in Section IV. The experimental results would be given in Section V. Section VI provides a brief summary.

II. Background and Motivation

A. Principle of Arithmetic Coding

Let L be a set of symbols. Every symbol $i \in L$, gets a probability $P_i \in [0,1]$ when $\sum_{i\in L} P_i = 1$. Codeword for symbols sequence $X^M = \{X_1, X_2, \ldots X_M\}$ is represented as $\lceil -\log_2 p(X^M) + 1 \rceil$ bits of number, $\sigma(X^M) = q(X^M) + p(X^M)/2$, where $p(X^M)$ and $q(X^M)$ are the probability and the cumulative probability of sequence X^M accordingly. In practice, integer implementation of arithmetic coder is based on three v-size registers: *low*, *high*, and *range* [23].

Let $\mathrm{cum\_freq}[i]$ be the cumulative counts for the symbol i, i.e., $\mathrm{cum\_freq}[i] = \sum_{j\le i} P_j$. Then the interval for the symbol i is $[\mathrm{cum\_freq}[i-1], \mathrm{cum\_freq}[i]]$. If the current probability interval is $[\mathrm{low}, \mathrm{high})$, then the update can be done by the following formula:

$$\begin{cases} \mathrm{range} = \mathrm{high} - \mathrm{low} + 1 \\ \mathrm{high} = \mathrm{low} + \mathrm{range} \times \frac{\mathrm{cum\_freq}[i-1]}{\mathrm{cum\_freq}[0]} \\ \mathrm{low} = \mathrm{low} + \mathrm{range} \times \frac{\mathrm{cum\_freq}[i]}{\mathrm{cum\_freq}[0]}. \end{cases} \tag{1}$$

The precision of registers *low*, *high*, and *range* grows with M increment. For decreasing coding latency and avoiding registers underflowing, the normalization procedure is used as follows.

1) If high $<$ HALF, where HALF $= 0.5 \cdot 2^v$, then a "0" bit is written into output bitstream.
2) If low $\ge$ HALF, then a "1" bit is written into output bitstream.
3) Otherwise, output bit is not defined. In this case, a *bit_to_follows* counter is increased. Then if condition 1 is satisfied then a "0" bit and *bit_to_follows* ones are written into output bitstream. If condition 2 is satisfied then a "1" bit and *bit_to_follows* zeros are written into output bitstream.

After the above conditions, the registers *low*, *high* are scaled to avoid underflowing. The corresponding codes are shown in Fig. 7.

Basically, AC will shorten the length of the coding interval continuously as new symbols arrive. If the input symbol's probability is high, the shrink of the coding interval will be slow. Otherwise, if there are some rare symbols in the coder, the speed of shrink will be fast. Thus, the coding interval will be large at the end of coding for high probability symbols which consume fewer bits for final codes than those of low frequency symbols.

In practical applications, conditional probabilities of symbols have better performance than non-conditional probabilities do. Then the context-based AC is widely used in the various fields. The context means conditions for current symbol. As far as the image coding is concerned, the context refers to neighbor pixels' states. After the transform stage in compression, the coefficients have the property of energy compaction. Then different coefficients form different context windows using a preset model. The different contexts will be sent to independent coding parts for updating the interval and emitting the code bits.

B. SPIHT Image Compression

SPIHT with lists algorithm uses three different lists to store significant information of wavelet coefficients for image coding purpose. Three lists are list of insignificant sets (LIS), list of insignificant pixels (LIP), and list of significant pixels (LSP). At first, SPIHT combines nodes of a coefficient tree in wavelet domain and its successor nodes into one set which is denoted as insignificant. With traveling each tree node, sets in the LIS are partitioned into four different subsets which are tested for significant state. The function used for testing set significant state is defined by the following formula:

$$S_n(\tau) = \begin{cases} 1, & \max\limits_{(i,j)\in\tau} \{|c_{i,j}|\} \ge 2^n \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Note: $C_{i,j}$ is the coefficient value for (i,j) position in the wavelet domain. The τ stands for the set of coefficients and $S_n(\tau)$ is used for significant state of set τ at bit-plane n.

If the magnitudes of nodes in the set are less than some predefined threshold, i.e., the set is insignificant, a bit will be emitted for the entire set. Because of similarity of coefficients in a zerotree, the strategy used in the partition procedure can be very efficient for coding transform information. That is why SPIHT can use fewer bits to code coefficients of one image after wavelet transform.

During the partition processing, AC consumes coding symbols with its contexts and switches to the different probability interval. After updating the probability interval, AC outputs the final coding bits. The key factor of AC performance is its context scheme. If the context model is simple, the corresponding hardware complexity is low at a cost of performance degradation. On the contrary, with the complicated context forms, the performance of AC can be improved, but the capacity of memory and throughput will be a significant bottleneck in architecture.

C. Challenges With SPIHT Arithmetic Coding Architecture

As far as hardware architecture of arithmetic coder in SPIHT is concerned, there are three main challenges for designers to solve during real-time implementation. First, a large amount of coding symbols is supplied to arithmetic coder which can be a bottleneck for high speed real-time applications. Because the scheme of SPIHT is a bit-plane based method, which codes each bit-plane from the most significant bit-plane (MSB) to the least significant bit-plane (LSB) sequentially, the quantity of context symbols for arithmetic coder will be proportional to the coded planes that are determined by the maximal wavelet coefficient. For the 9/7 wavelet filter, the precision of wavelet coefficient will be increased compared to the pixel precision after transformation stage. Therefore in order to keep speed balance between the wavelet transformation and the arithmetic coder, the throughput of arithmetic coder must match the input rate of the wavelet stage. For the design of arithmetic coder, we test some typical images with different pixel precision and compression ratio using the QccPackSPIHT software. The number of context symbols used for the arithmetic coding is shown in Tables I–III. In Tables I–III, the average context symbols per pixel and the bit-planes used for arithmetic coding are proportional to the bit rate and the bit depth of pixel. In order to achieve the balance

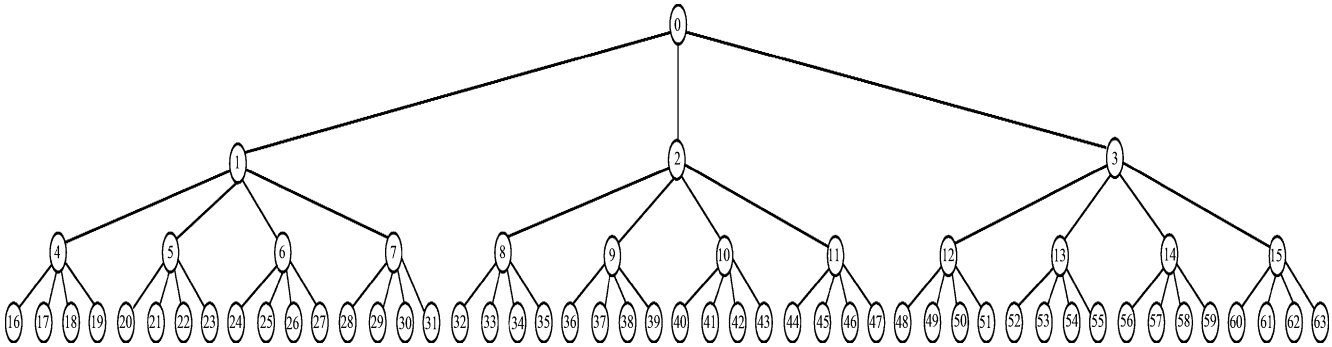

Fig. 1. Travel order of breadth first search.

TABLE I
CONTEXT NUMBER WITH PRECISION AT 8 BIT PER PIXEL AND TARGET RATE AT 4 BIT PER PIXEL

Image	Context number	Bit-planes number	Context per pixel
lena	727109	12	2.77
Barbara	719188	11	2.74
Airport	2900560	11	2.77
Pentagon	2915135	11	2.78
Bike	15288155	12	2.92
Café	15578839	11	2.97
Woman	15092128	12	2.88

TABLE II
NUMBER WITH PRECISION AT 10 BIT PER PIXEL AND TARGET RATE AT 5 BIT PER PIXEL

Image	Context number	Bit-planes number	Context per pixel
s001_10_s_s	3907231	12	3.73
s002_10_s_s	3895210	15	3.71
s003_10_s_s	4656309	15	4.44
s004_10_s_s	3848611	12	3.67
s005_10_s_s	3878811	11	3.70
s006_10_s_s	3909874	13	3.73
s007_10_s_s	3859761	13	3.68

mentioned above, there are two basic ways for the arithmetic coder, i.e., increasing the clock frequency of arithmetic coder without area overhead or multiple arithmetic coders' replication without the clock frequency increment. In the architecture, a replication method is used to alleviate the bottleneck. Second, the memory size in a single arithmetic coder can be limited for implementation. The memory used for the probability values and the cumulative probability values are main parts for the arithmetic coder. In order to simplify the architecture, we need small size of memory for these parts. In software, every probability value is represented by an aligned data type, which is not efficient as the real range for the probability used. Therefore in the architecture design, we cannot use simple array for these memory part as software does. The last challenge comes from the carry propagation problem caused by the probability update operation, which increases the critical path of the coder and reduces the speed of arithmetic coder.

III. SPIHT WITH BREADTH FIRST SEARCH AND ITS CONTEXT MODEL FOR ARITHMETIC CODING

A. Bit-Plane Parallel SPIHT With Breadth First Search

In order to prevent multiple scan of the wavelet coefficients that is difficult for the real-time hardware implementations, we uses the breadth first search (BFS) for traveling a zerotree. The SPIHT-BFS visits each coefficient only once and outputs coding information to form context symbols according to the corresponding context model for hardware.

After the SPIHT-BFS is defined, the main challenge for hardware implementation comes from the sequential processing style of bit-planes. In order for bit-plane parallel processing, all kinds of significant information for each bit-plane, i.e., the pixel significant information, the D set and the L set significant information should be achieved simultaneously. According to the significant test function, if a pixel turns to be significant at the $p+1$th bit-plane, then it will be permanently significant for the other bit-planes from the pth to the LSB bit-plane. On the other side, if the magnitude bit of a pixel at the pth bit-plane is 1, but the current state of the pixel is insignificant, then the pixel also becomes significant. Therefore a pixel's significant state at the pth bit-plane is defined by the following formula:

$$\begin{cases} \mathrm{Sig}^{p}(i,j) = \mathrm{Sig}^{p+1}(i,j) + \mathrm{Mag}^{p}(i,j) \\ \mathrm{Sig}^{\mathrm{msb}}(i,j) = 0. \end{cases} \tag{3}$$

Note: $\mathrm{Sig}^{p}(i,j)$ is the significant state for pixel at (i,j) of pth bit-plane, $\mathrm{Mag}^{p}(i,j)$ stands for the corresponding magnitude value.

After expanding formula (3), the relationship between significance of each coefficient and magnitude bit can be obtained. For the architecture design, an OR gates array can be exploited for the significant states. Then the significant information is independent on the coding planes. For the D set significant state and the L set significant state, we can also exploit some logic gates and delay unit to realize parallel processing. Then all information needed for coding a tree is ready for parallel processing.

Because the set significant state bits can only be resolved by logic gates after one tree is visited, a whole tree needs to be stored. Fig. 1 draws the travel order of zerotree by the breadth first search with three levels of wavelet transform. The pipeline can be used for the SPIHT-BFS and the latency is just one zerotree clocks.

B. Context Model for AC in SPIHT With Breadth First Search

In SPIHT-BFS, wavelet coefficients are organized into zerotrees through bands. The coding information is formed during each run of visiting from the MSB bit-plane to the LSB bit-plane. And for the tradeoff between memory constraint and coding performance, the context model should be confined to only one zerotree. In order to reduce complexity, only four

TABLE III
CONTEXT NUMBER WITH PRECISION AT 16 BIT PER PIXEL AND TARGET RATE AT 8 BIT PER PIXEL

Image	Context number	Bit-planes number	Context per pixel
s001_16_r_s	6971432	17	6.65
s002_16_r_s	6970231	16	6.65
s003_16_r_s	6977198	15	6.65
s004_16_r_s	6988715	16	6.66
s005_16_r_s	7013694	16	6.69
s006_16_r_s	6986935	16	6.66
s007_16_r_s	6999621	15	6.68

(note: wavelet filter is 9/7 lifting scheme with four levels, and images with 10 bits, 16 bits are chosen from spectral and remote sensing of real satellite images with resolution of 1024x1024.)

TABLE IV
CONTEXT DESIGN OF FOUR TYPES OF INFORMATION

Con.number	Context label	Explanations
	FC Type	
0	FC_Non_Sig	Three neighbor nodes are all insignificant.
1	FC_One_Sig	One of three neighbor nodes is significant.
2	FC_Two_Sig	Two of three neighbor nodes are significant.
3	FC_Three_Sig	Three neighbor nodes are all significant.
	FSign Type	
4	FS_Non_Sig	Three neighbor nodes are all positive.
5	FS_One_Sig	One of three neighbor nodes is negative.
6	FS_Two_Sig	Two of three neighbor nodes are negative.
7	FS_Three_Sig	Three neighbor nodes are all negative.
	FD Type	
8	FD_Non_Sig	Three neighbor nodes' descendants are all insignificant.
9	FD_One_Sig	One of three neighbor nodes' descendants is significant.
10	FD_Two_Sig	Two of three neighbor nodes' descendants are significant.
11	FD_Three_Sig	Three neighbor nodes' descendants are all significant.
	FL Type	
12	FL_Non_Sig	Three neighbor nodes' grant descendants are all insignificant.
13	FL_One_Sig	One of three neighbor nodes' grant descendants is significant.
14	FL_Two_Sig	Two of three neighbor nodes' grant descendants are significant.
15	FL_Three_Sig	Three neighbor nodes' grant descendants are all significant.

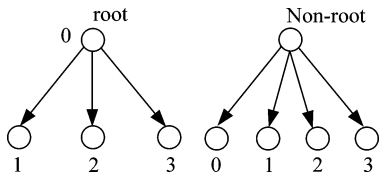

Fig. 2. Node's position in the zero tree.

types of information is used to form context model in the coding stage, i.e., the current position significant, the magnitude value, the current position sign value, the current position descendant set value and the current position grant descendant set value.

In SPIHT-BFS, each node in the zerotree can emit four types of information, i.e., FC (for the current position significant or magnitude value), FSign (for the current position sign value), FD (for the current position descendant set value), and FL (for the current position grant descendant set value). Four bits will output at most if all information are valid. If all kinds of information are invalid, there is no bit emitted in the code stream. During the context design, the context type is divided into four categories according to the four types above. For a node in a zerotree, the root node and non-root nodes can be treated by different ways. Fig. 2 shows the position of two kinds of nodes. The root node has no brother in the same level, but has three son nodes in the next level. On the other hand, a non-root node has four son nodes in the next level. For the simplicity, the label is used in Fig. 2 as the context window for both root and non-root nodes. Then the context model can be built on the state of these four nodes. When the node is visited in a zerotree, the context of current node is depended on the other three neighbor nodes' state. For each type of information, there are $2^3 = 8$ contexts. And the total context number is $4 * 8 = 32$ for the four types of information. The symbol of each context is 0 or 1, which means significant state for the FC, FD, FL, or FSign state for the current node. In order to reduce the context number, we decrease each type of information's context to 4. Then the total number of context can be reduced to 16. Table IV gives the detailed context form.

IV. ARCHITECTURE OF ARITHMETIC CODER

A. Whole SPIHT Architecture

In Fig. 3, the detailed architecture of SPIHT encoding is shown.

The original images are transformed by the *line based lifting wavelet engine* [24] at first. The transformed coefficients are written into the *wavelet coefficients buffer*. The *processor dispatcher* receives coefficients in the breadth first way from the *wavelet coefficients buffer* and allocates these coefficients to one of arithmetic coders *(ACs)* from eight processors array through the internal bus. In order to adapt a wide precision and compression ratio range, eight arithmetic coders are symmetric and

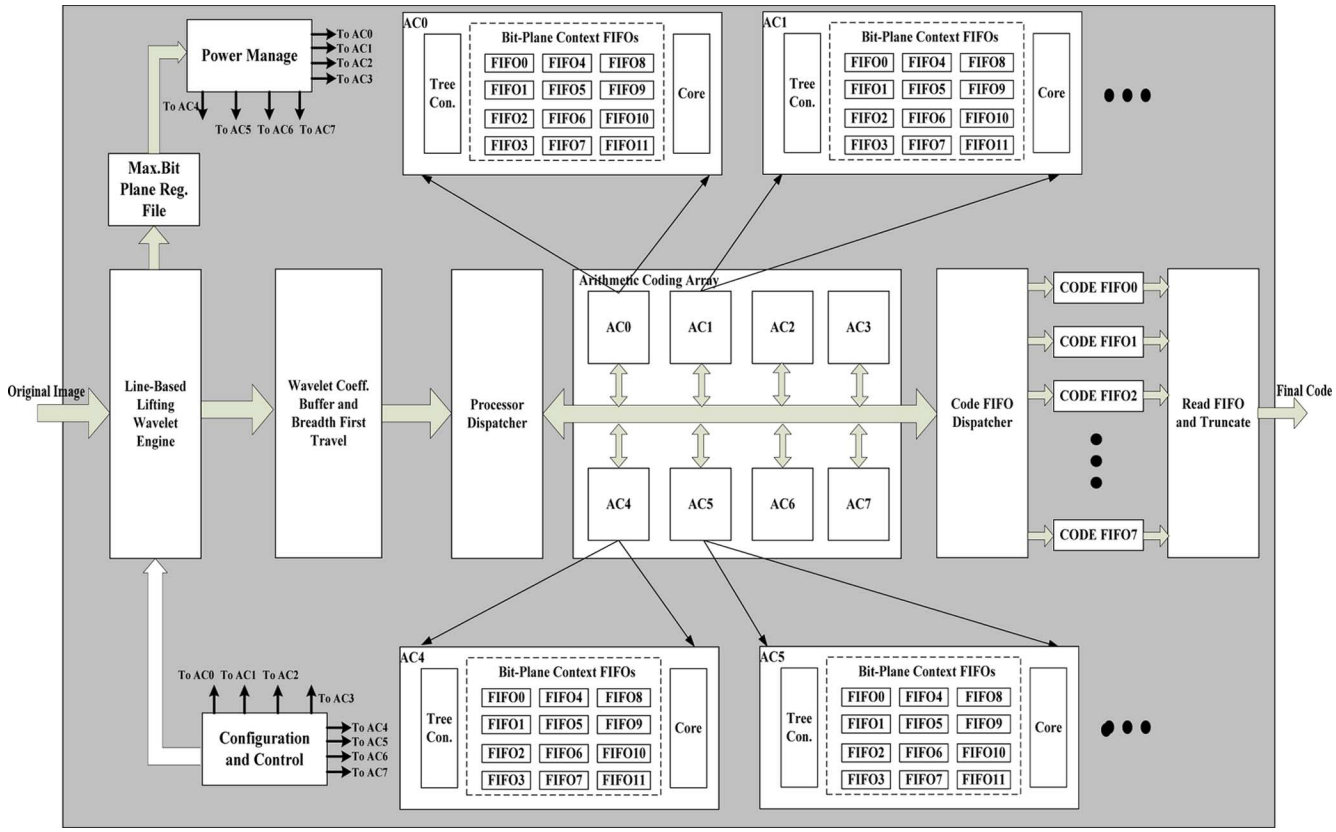

Fig. 3. Architecture of SPIHT encoding.

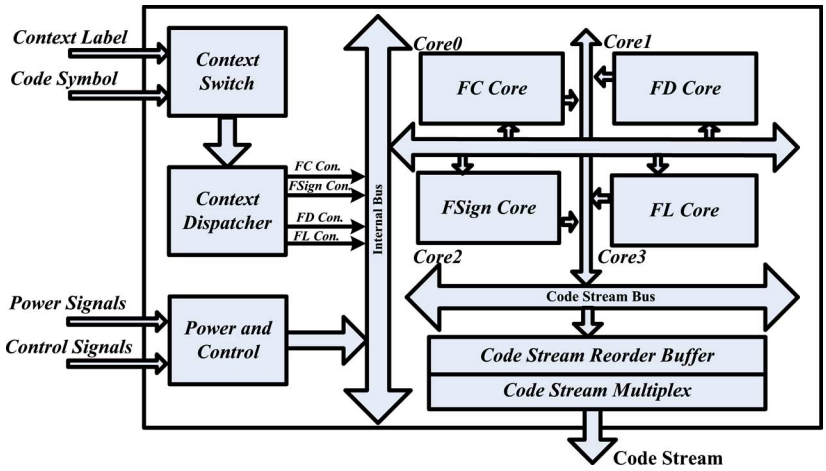

Fig. 4. Arithmetic coder's core structure.

work in parallel. The output of each arithmetic coder is sent to the internal bus and is distributed to the corresponding code FIFO by the *code FIFO dispatcher*. The *Read FIFO and Truncate* module are responsible for the final code stream formation, which reads each code FIFO from top to bottom and truncates the code stream according to the bit rate requirement. Besides the main parts in the architecture, there are some auxiliary modules in Fig. 3. The *power management part* will stop the clock input for the unused bit-planes of each arithmetic coder based on the maximal bit-plane register file for power reduction. The *configuration and control part* is responsible for the parameters setting such as image resolution, wavelet type, decomposition level and target bit rate. The control signals for the whole architecture are also asserted by this part.

From Fig. 3, the arithmetic coder in the overall architecture plays an important role during the coding process. The arithmetic coder consists of three main parts, i.e., the tree construction noted as *Tree Con*, the *bit plane context FIFOs array* and the *coding core*. The tree construction part visits the wavelet coefficients by the breadth first search order. During the reading process, the context values of each bit-plane are formed based on the context model mentioned. For speedup, all valid bit-planes are scanned in parallel. The invalid bit-planes are idle by stopping the corresponding clock. The *bit plane context FIFOs*

TABLE V
EXAMPLE OF OUT OF ORDER EXECUTION

Context dispatcher part					State of coding cores			
Cycle	#	Context	Send	Finish	FC core	FD core	FL core	FSign core
1	1	(FC0,D0)	1		1	2	3	4
	2	(FD0,D0)	2					
	3	(FL0,D0)	3					
	4	(FS0,D0)	4					
2	5	(FC1,D1)	5	1	5	2	3	4
	6	(FD1,D1)	–					
3	7	(FC2,D2)	7	5	7	6	–	8
	8	(FS1,D1)	8	4				
			6	2				
4	9	(FC3,D3)	9	7	9	10	11	8
	10	(FD2,D2)	10	6				
	11	(FL1,D1)	11					
	12	(FS2,D2)	–					
5				9	–	10	11	12
				8				
6				10	–	–	–	12
				11				
7				12	–	–	–	–

(note: FC, FD, FL, or FS denote the context label; D is the binary code symbol.)

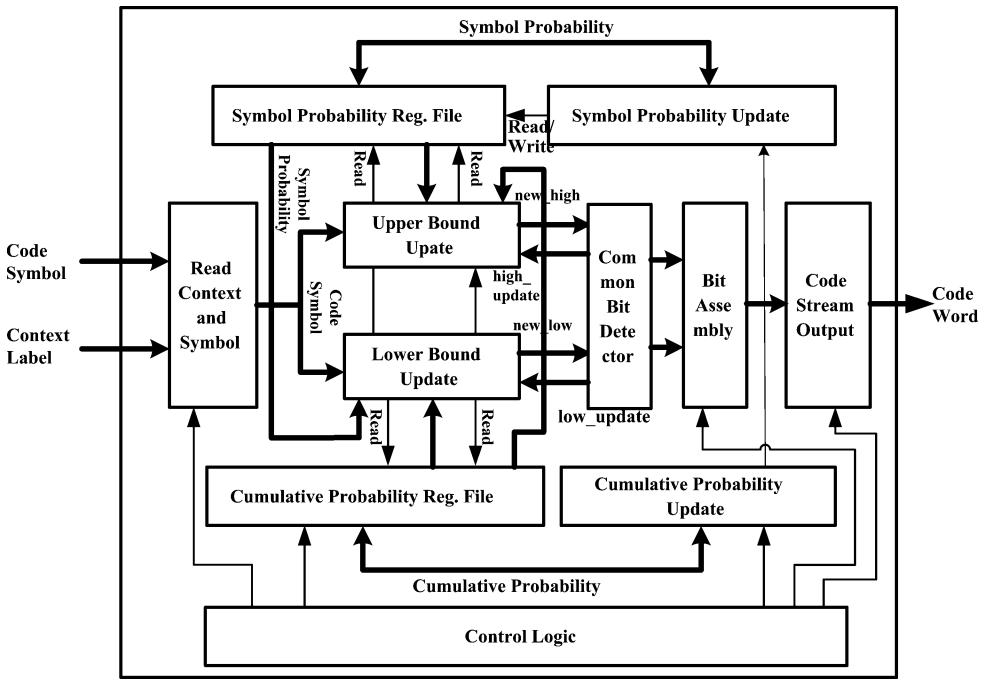

Fig. 5. Internal structure of arithmetic coder.

array includes twelve FIFOs, which store the context values of twelve bit-planes. The size of each FIFO is 256×5 bits because each code tree has 256 nodes and 16 different contexts will use 4 bits, the last bit is used for the binary context symbol. The coding core part reads these context first-input–first-outputs (FIFOs) sequentially and calculates the corresponding context to form code bytes.

B. Architecture of AC Core

The structure of the core part is illustrated in Fig. 4. The input signals can be divided into two categories, i.e., the context related and the control related. When the context label and binary code symbol arrive, the *context switch* differentiates the input context and sends the context value to the context dispatcher by different paths. The main task of the context dispatcher is to schedule the order of the input contexts, which are sent to different calculation cores. In order for speedup, the *context dispatcher* can emit the context values to each core by a disorder, which means that execution order can be different from that of input. A small buffer for context value is set in the *context dispatcher* to implement reorganizing the processing order. Table V shows an example of execution using twelve different contexts. Each of four coding cores has its state register to indicate whether the coding core can receive new context. When there is no context in the buffer,

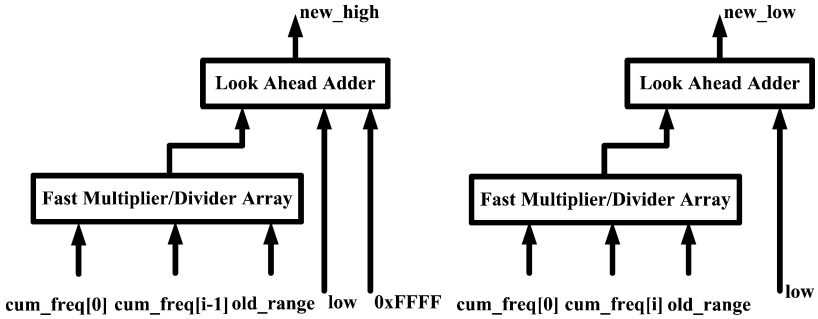

Fig. 6. Calculation units for update of the probability interval.

```
for(;;){    /*Loop to output bits. */
        if(high < Half){
                bit_plus_follow(0);  /* Output 0 if in low half.*/
        }
        else if(low >= Half){
                bit_plus_follow(1);
                low -= Half;
                high -= Half;       /* Subtract offset to top */
        }
        else if(low >= First_qtr && high < Third_qtr){ /* Output an opposite bit */
                                        /* later if in middle half.*/
                bits_to_follow +=1;
                low -= First_qtr;               /* Subtract offset to middle*/
                high -= First_qtr;
        }
        else break;                     /* otherwise exit loop */
        low = 2 * low;                  /* Scale up code range */
        high = 2 * high + 1;
}
```

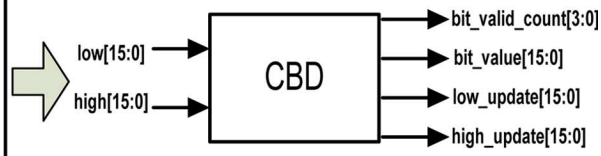

Fig. 7. Loop unrolling structure.

the state of core is set to idle. If a context symbol arrives, the state of core is set to the context label to block any new context. The *dispatcher* checks the states to find if there is an idle core. Then the *dispatcher* combines several contexts and emits them to the corresponding cores. The context and its binary symbol are emitted to the corresponding calculation cores, i.e., *FC core*, *FSign core*, *FD core*, and *FL core* through the internal bus. If the incoming context is blocked, it will be delayed in the *dispatcher* and wait for the next clock cycle to be emitted. At beginning, four cores are ready for processing the contexts. Then in the first clock cycle, four contexts are emitted simultaneously. In the second clock cycle, two new contexts arrive. As (FD_0, D_0) context pair is not finished, (FD_1, D_1) context pair is blocked. Only (FC_1, D_1) context pair can be emitted to the *FC core* because (FC_0, D_0) has been done by the *FC core*. But in the third clock cycle, the (FD_1, D_1) context pair can be emitted to the *FD core* because the *FD core* is ready to process new pair.

From Fig. 4, every code core works independently, which allows multi-contexts being calculated in parallel. Then the outputs of each core are connected with another bus. The code stream reorder buffer is used to sort the order of each code core as the execution order differs from the input order. The code stream multiplex module collects all code bits emitted by the code cores and assembles these bits into bytes. The code stream bytes are finally emitted to external by output ports. The internal structure of each code core is identical, as shown in Fig. 5.

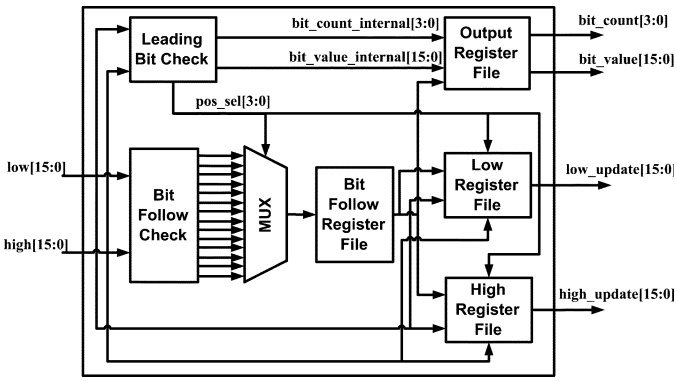

Fig. 8. CBD structure.

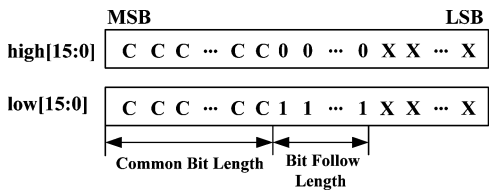

Fig. 9. Explanation of leading bit check and bit follow check.

For each *core*, the *Read Context and Symbol* part compares the context label with its internal register to judge whether the context label conforms to its own tag. Then the correct label is transmitted to *Boundary Update* part. In this part, the upper and

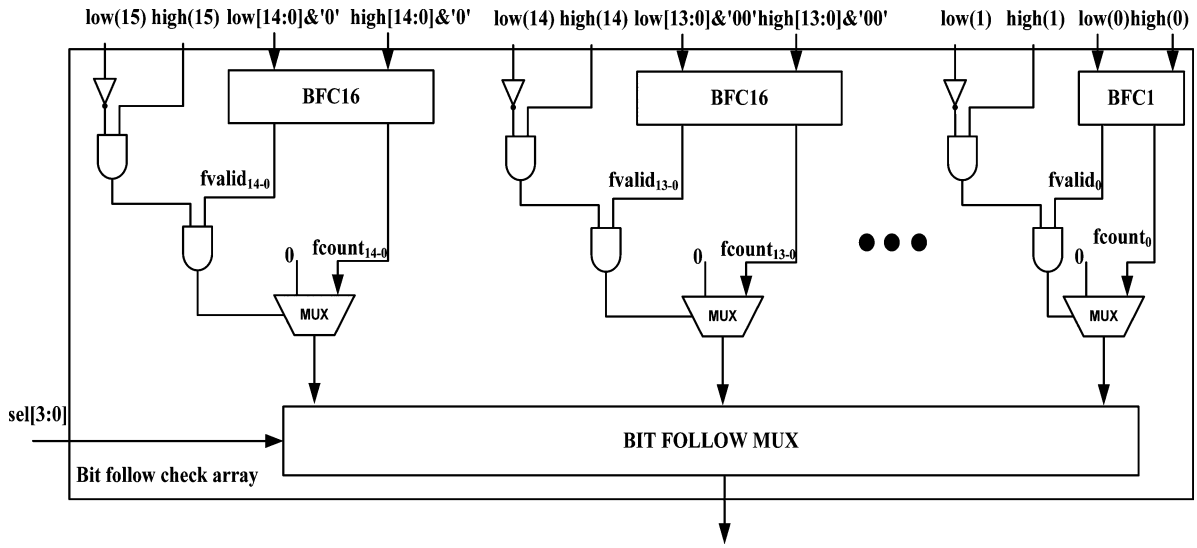

Fig. 10. Bit follow check structure.

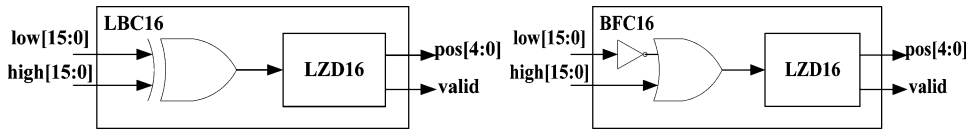

Fig. 11. LBC16 and BFC16 structures.

lower bounds are computed by two different parts, i.e., *Upper Bound Update* and *Lower Bound Update*. The *coding symbol probability register file* records the symbol probability values for the symbol i. The cumulative probabilities $\mathrm{cum\_freq}[i]$ and $\mathrm{cum\_freq}[i-1]$, which are stored in the *cumulative probability register file*, accompanied with the old bound values *high* and *low* to compute new bounds for the probability interval. The outputs of two *update parts*, i.e., *new_high* and *new_low*, are calculated by these variables based on the formula (1). Fig. 6 gives the detailed calculation units for these values. There are only one adder and one multiplier in the unit which is suitable for high speed implementation with FPGA devices. For speedup purpose, a CLA and a fast multiplier array are employed to reduce the delay of critical path. The new bound values are then registered and connected to the *common bit detector* (CBD) part which unrolls the internal loop and records the same bits from the MSB to the LSB between two registers. At last the same bits are collected to form byte-align code stream in the *Bit Assembly*. These bytes are supplied directly to the *Code Stream Output* part for emitting bytes to the external bus. The *Coding Control* part is responsible for the whole code core's running and sending the various commands and control signals.

C. CBD Structure

For AC, the code bits are generated by an internal loop, which is essential to the architecture design. In Fig. 7, a CBD module is used to unroll the internal loop.

The inputs of CBD are *low* and *high* values after calculation of formula (1). The *bit_valid_count* signals the output bit count from the MSB to the LSB of bit_value register. The *bit_value* is just a concatenation of common bits and bits_to_follow which is used for underflow. The *low_update* and *high_update* registers

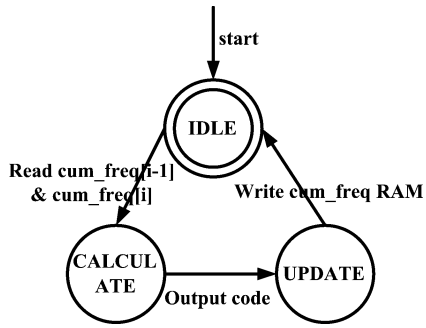

Fig. 12. State machine for coding control.

are shifted values for two bounds used for the next run. Fig. 8 illustrates the detailed structure inside the CBD module.

In CBD, the leading bit check (LBC) detects the common bits between *low* and *high* registers which consists of a 16-XOR gate and a leading zero detector for 16 bits (LZD16) circuit used by [25]. The bit follow check (BFC) module is an array for checking the mode of underflow, which is explained by Fig. 9. For speedup purpose, we check 15 possible cases for the bit follow check, which means that the pos_sel $[3:0]$ of LBC selects one of the bit follow values from 15 cases based on the proper common bit position. Fig. 10 shows the internal structure for the BFC. The BFC denotes the bit follow check of two vectors, which can be implemented by a simple logic gate and an LZD circuit in Fig. 11. The corresponding bit follow value is then registered in the *Bit Follow Register File*, which is used for the output and the shift of two bounds. Three register files are employed for the *output*, *low* and *high* registers. The *low* and *high* registers are shifted left according to the values of *pos_sel* $[3:0]$ and the *bit_to_follow* register. And the output register emits the proper common bits and the underflow bits according to these registers.

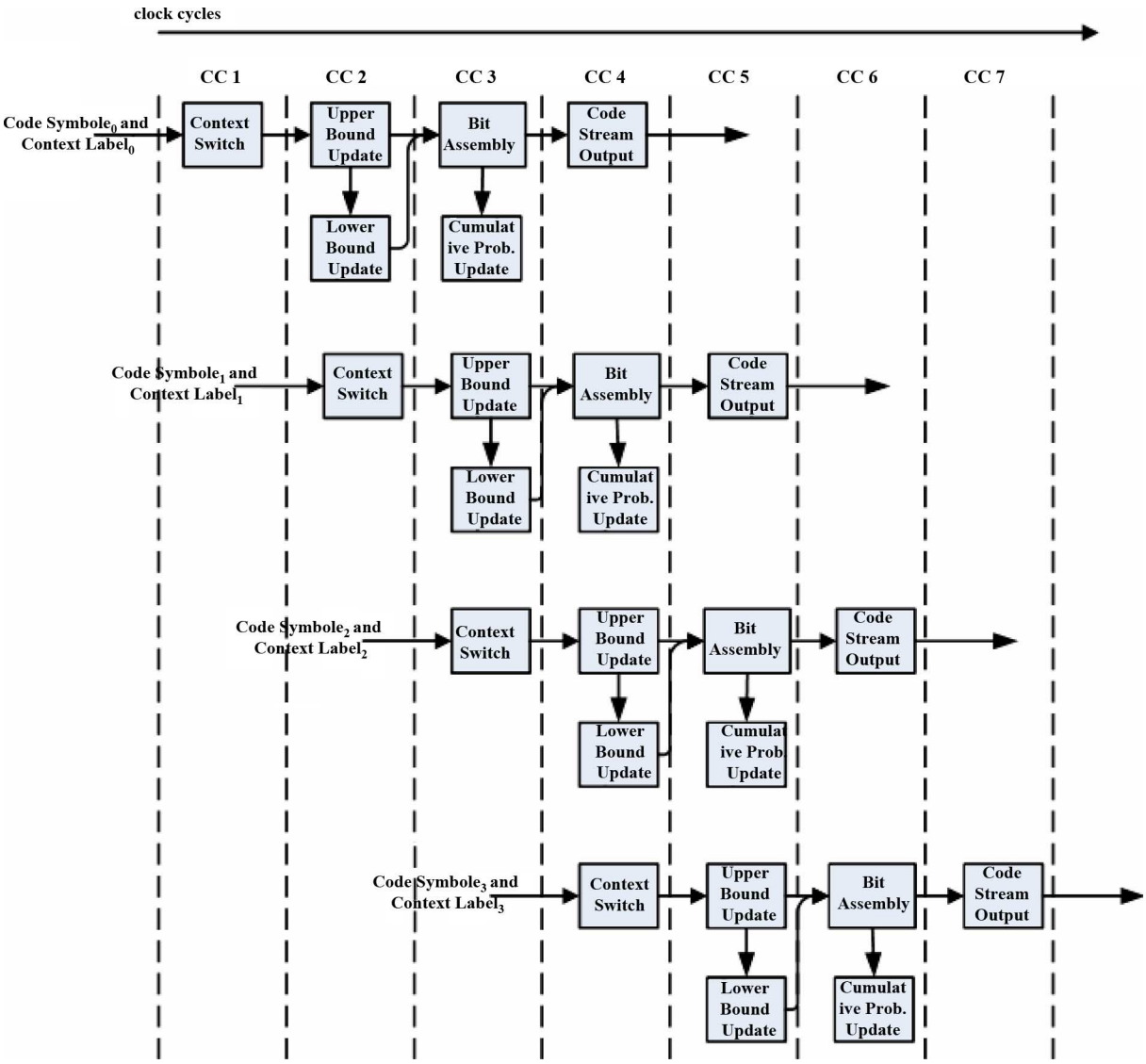

Fig. 13. Timing order of arithmetic coder.

TABLE VI
PSNR RESULTS IN dB

Image	SPIHT-AC	SPIHT-NC	SPIHT-NL	SPIHT-HW
Lena(512x512)	40.41	39.98	39.24	39.89
Barbara(512x512)	36.41	35.94	35.06	36.01
Airport(1024x1024)	33.27	32.79	32.38	32.67
Pentagon(1024x1024)	34.31	33.81	33.21	33.71
Woman(2048x2560)	38.28	37.73	36.22	37.19
Bike(2048x2560)	37.70	36.98	35.91	36.51
Café(2048x2560)	31.74	30.89	30.10	30.68

(note: SPIHT-AC means SPIHT with arithmetic coding, SPIHT-NC is SPIHT without arithmetic coding, SPIHT-NL stands for SPIHT without lists, SPIHT-HW is SPIHT with hardware arithmetic coder; The 9/7 wavelet with four levels is applied, and the compression bit rate is 1 bpp.)

For the cumulative frequency update, each cumulative frequency value behind the current code symbol i and total cumulative frequency should be added by 1 as drawn in the following formula:

$$\begin{cases} \text{cum\_freq}[j] = \text{cum\_freq}[j] + 1, & j \geq i \\ \text{cum\_freq}[0] = \text{cum\_freq}[0] + 1. \end{cases} \tag{4}$$

As for hardware, in order to avoid updating each cumulative frequency value sequentially, an independent register is used for each symbol's cumulative frequency value. Then one clock cycle can be used for all frequency values' calculations.

Fig. 12 shows the state machine for the control logic part. The state machine stays at the IDLE waiting for a new coding

TABLE VII
SYNTHESIS RESULTS OF THE WHOLE COMPRESSION SYSTEM

FPGA	SLICES	4 Input LUTs	Clock Freq. (MHz)	Critical Path(ns)	TP. (MBit/Sec)
XC2V3000	10317	14742	56.404	17.730	902.464

TABLE VIII
SYNTHESIS RESULTS OF THE ARITHMETIC PART

FPGA	SLICES	4 Input LUTs	Clock Freq. (MHz)	Critical Path(ns)	TP. (MSPS)
XC2V3000	6974	2934	71.05	14.074	284.2

TABLE IX
MEMORY CAPACITY FOR CUMULATIVE PROBABILITIES

Context type	Context number	Symbols per context	Memory Size in bits
FC_CONTEXT	4	2	16x8
FD_CONTEXT	4	2	16x8
FL_CONTEXT	4	2	16x8
FSign_CONTEXT	4	2	16x8
Total	16	8	512

symbol. When a new symbol arrives, the state machine asserts a read signal for the lookup table of the cumulative probability and jumps to the CALCULATE state. In the CALCULATE, the probability interval bounds are updated. The control signals of the *Bit Assembly and Stream Output parts* are sent by the state machine in this state. The next state is the UPDATE state which provides a write signal for the cumulative ram and starts the cumulative probability update process. After all cumulative probability values are renewed, the state machine returns to the initial IDLE state for next symbol.

Fig. 13 illustrates the timing order for the whole pipeline stages of the architecture. Due to pipelining style, the coder can consume one symbol per clock cycle. The pipeline has four stages: context switch, probability bounds update, bit emit/cumulative update, and code stream output. In the pipeline, the code symbol and context label are processed every clock at each stage based on the analysis of the architecture. After the pipeline sets up, the code stream bytes can outflow every clock.

V. EXPERIMENTAL RESULTS

A. Software Results

The experimental results come in two folds, i.e., software and hardware. First, PSNR results for typical images using different SPIHT methods are recorded, including SPIHT with arithmetic, SPIHT without arithmetic, SPIHT without lists and arithmetic and our SPIHT prototype. Table VI lists the detailed data. From the results, SPIHT-HW is slightly lower than SPIHT-AC as the precision is limited during the wavelet transform and a simple context model is involved.

B. Hardware Results Based on FPGA Device

The overall architecture including the wavelet part and the arithmetic part is synthesized and simulated by VHDL using XC2V3000 as target device. The results are reported by XILINX ISE9.1-XST and shown in the Table VII. The maximal

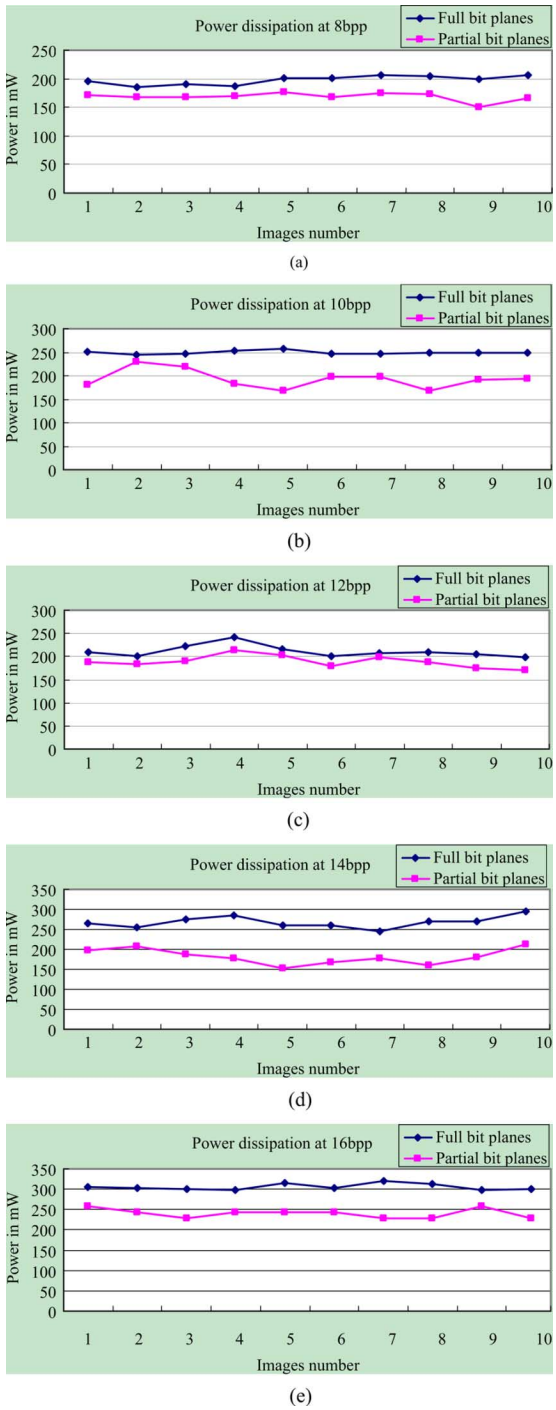

Fig. 14. Power saving curves with different pixel precisions in arithmetic coding part.

TABLE X
PERFORMANCE COMPARISONS OF DIFFERENT SPIHT CODERS ON FPGA DEVICES

Coders	FPGA device	SLICES	LUTs	Clock Freq. (MHz)	TP. (MBits/Sec)
Ritter[20]	XC4000	-	-	40	8.48
Fry*[18]	XCV2000E	-	-	75	800.00
Huang*[19]	APEX EP20K1000E	-	-	-	36.80
Li[29]	X3S1500L	2366	3416	-	-
Jyotheswar*[30]	XC4VLX25	7021	13356	35	280.00
Corsonello*[31]	XC2V1000	1637	-	100	92.96
Proposed	XC2V3000	10317	14742	56.40	902.46

(note: '*' denotes that AC is not shown in the corresponding coder; '-' denotes that data cannot be published.)

TABLE XI
PERFORMANCE COMPARISONS OF AC CODERS

AC coders	Clock Freq.(MHz)	TP.(MSPS)	Symbol bit per clock	Technology
Marks[33]	75.00	64.00	0.80	CMOS 5S 0.35 um IBM
Kuang[34]	25.00	3.00	0.12	0.8 um SPDM
Stefo[32]	32.00	256.00	8.00	ProASIC A500K FPGA
Osorio[26]	40.00	340.00	4.00	0.18 um UMC EuroPractice
Vanhoof[27]	32.00	10.67	0.33	0.5 um CMOS 3 layer metal
Printz[35]	33.33	16.67	0.50	8 Xilinx XC3090 FPGAs
Dyer-1[28]	48.85	97.70	2.00	Altera Stratix
Dyer-2[28]	211.86	423.72	2.00	0.18 um TSMC
Proposed	71.05	284.20	4.00	Xilinx XC2V3000 FPGA

clock frequency is 56.404 MHz. For hardware implementation, the input is confined to gray images with resolution of 1024×1024 and the precision of pixel from 8 bits to 16 bits. Thus the throughput of whole compression system is 56.404 frames per second (fps). Then for pixel precision of 16 bits with resolution 1024×1024, the throughput of coder can be $56.404 \times 1024 \times 1024 \times 16$ =902.464 Mb/s at its maximum. In order to test arithmetic part independently, we also synthesize this part and record the results in the Table VIII. There are four contexts processed simultaneously in the architecture. Each context uses one bit for symbols. Then four symbols can be consumed simultaneously. The throughput is $71.05 \times$ 4 MSPS/s, i.e., 284.2 MSPS/s.

C. Memory, Throughput and Power Analysis

As the memory size is important, the memory capacity of arithmetic coder in SPIHT-HW is analyzed briefly. In the arithmetic coder, a short type integer is used for the symbol probability. The main part of memory is composed by the cumulative probability storage. Table IX gives the cumulative probability memory content in the architecture. Only 512 bits are allocated for storage requirement and one block ram unit in the device is enough for this. Another two parts of memory come from the context FIFOs and the code FIFOs, which are responsible for the context buffer and the code stream buffer. The number of the context FIFOs is 12×8 and the total bits are $12 \times 8 \times 256 \times 5 =$ 122 880 bits. Each code FIFO can be set to 128×8 bits. Then the size of the code FIFOs is $8 \times 128 \times 8 = 8192$ bits. Therefore, the total size of memory used in the architecture is 128.5 kb.

For comparisons, Table X lists the results of different SPIHT image compression systems based on FPGA devices. And Table XI shows the throughput comparison with other AC coders. From the experimental results, the proposed SPIHT image compression systems and the AC coder can obtain good score in many advanced architectures.

The power saving effect is also tested by several typical images. Fig. 14 illustrates the detailed results reported from XPower Analyzer tool with compression ratio at 8:1. From the experimental results, the power saving rate can be about 20.08% on average due to the method of stopping the clock of invalid bit-planes.

VI. CONCLUSION

Arithmetic coding makes itself a standard technique for its high efficiency. However, as far as hardware implementation is

concerned, the complexity of calculation limits AC in the filed of high speed real-time coding. For improvement of throughput purpose, we propose a high speed architecture of AC used in SPIHT without lists algorithm. In the architecture, a simple context scheme is used first to reduce the memory size. Then high speed calculation units are employed for speedup purpose. Especially, a power control module can reduce the power dissipation efficiently. It is a high parallelism and calculation device that makes the speed of context processing fast. From the simulation results, our AC architecture can meet many high speed image compression requirements. And the degradation of performance incurred by the fixed point calculation is slight.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their helpful comments and revisions.

REFERENCES

[1] J. Rissanen, "Generalized kraft inequality and arithmetic coding," *IBM J. Res. Developm.*, vol. 20, no. 3, pp. 198–203, May 1976.

[2] J. Rissanen and G. G. Langdon, "Arithmetic coding," *IBM J. Res. Developm.*, vol. 23, no. 2, pp. 149–162, Mar. 1979.

[3] *ISO/IEC JTC1 Information Technology-Digital Compression and Coding of Continuous-Tone Still Images-Part 1: Requirements and Guidelines*, ISO/IEC International Standard 10918-1, ITU-T Rec. T.81, 1993.

[4] *JPEG2000 Part I Final Draft International Standard*, ISO/IEC JTC1/SC29/WG1 N1890, Sep. 2000.

[5] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Trans. Image Process.*, vol. 9, no. 7, pp. 1158–1170, Jul. 2000.

[6] B. Cao, Y.-S. Li, and K. Liu, "VLSI architecture of MQ encoder in JPEG2000," *J. Xidian Xuebao*, vol. 31, no. 5, pp. 714–718, Oct. 2004.

[7] *Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 ISO/IEC 14496-10 AVC)*, JVT-G050, Joint Video Team of ITU-T and ISO/IEC JTC 1, Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, Mar. 2003.

[8] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Trans. Circuits Syst. for Video Technol.*, vol. 13, no. 7, pp. 620–636, Jul. 2003.

[9] A. Said and W. A. Pearlman, "A new ,fast and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. Circuits Syst. for Video Technol.*, vol. 6, no. 3, pp. 243–249, Mar. 1996.

[10] Y. Wiseman, "A pipeline chip for quasi arithmetic coding," *IEICE Trans. Fundamentals*, vol. E84-A, no. 4, pp. 1034–1041, Apr. 2001.

[11] K. Andra, T. Acharya, and C. Chakrabarti, "A multi-bit binary arithmetic coding technique," in *Proc. Int. Conf. Image Process.*, Vancouver, BC, Canada, Sep. 2000, vol. 1, pp. 928–931.

[12] A. A. Kassim, N. Yan, and D. Zonoobi, "Wavelet packet transform basis selection method for set partitioning in hierarchical trees," *J. Electron. Imag.*, vol. 17, no. 3, p. 033007, Jul. 2008.

[13] M. A. Ansari and R. S. Ananda, "Context based medical image compression for ultrasound images with contextual set partitioning in hierarchical trees algorithm," *Adv. Eng. Softw.*, vol. 40, no. 7, pp. 487–496, Jul. 2009.

[14] M. Akter, M. B. I. Reaz, F. Mohd-Yasin, and F. Choong, "A modified-set partitioning in hierarchical trees algorithm for real-time image compression," *J. Commun. Technol. Electron.*, vol. 53, no. 6, pp. 642–650, Jun. 2008.

[15] F. W. Wheeler and W. A. Pearlman, "SPIHT image compression without lists," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, Istanbul, Turkey, Jun. 2000, pp. 2047–2050.

[16] J. Bac and V. K. Prasanna, "A fast and area-efficient VLSI architecture for embedded image coding," in *Proc. Int. Conf. Image Process.*, Oct. 1995, vol. 3, pp. 452–455.

[17] J. Singh, A. Antoniou, and D. J. Shpak, "Hardware implementation of a wavelet based image compression coder," in *Proc. IEEE Symp. Adv. Digit. Filter. Signal Process.*, Jun. 1998, pp. 169–173.

[18] T. W. Fry and S. A. Hauck, "SPIHT image compression on FPGAs," *IEEE Trans. Circuits Syst. for Video Technol.*, vol. 15, no. 9, pp. 1138–1147, Sep. 2005.

[19] W.-B. Huang, A. W. Y. Su, and Y.-H. Kuo, "VLSI implementation of a modified efficient SPIHT encoder," *IEICE Trans. Fundamentals*, vol. E89-A, no. 12, pp. 3613–3622, Dec. 2006.

[20] H. J. Ritter, "Wavelet based image compression using FPGAs," Ph.D. dissertation, Martin Luther Univ., Halle-Wittenberg, Germany, 2002.

[21] H. Pan, W.-C. Siu, and N.-F. Law, "A fast and low memory image coding algorithm based on lifting wavelet transform and modified SPIHT," *Signal Process.: Image Commun.*, vol. 23, no. 3, pp. 146–161, Mar. 2008.

[22] J. E. Fowler, A. G. Tescher, Ed., "Qccpack: An open-source software library for quantization, compression and coding," in *Appl. Digit. Image Process. XXIII, Proc. SPIE 4115*, Aug. 2000, pp. 294–301.

[23] I. C. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987.

[24] C. Chrysafis and A. Ortega, "Line based, reduced memory, wavelet image compression," *IEEE Trans. Image Process.*, vol. 9, no. 3, pp. 378–389, Sep. 2000.

[25] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 2, no. 1, pp. 124–128, Mar. 1994.

[26] R. R. Osorio and B. Vanhoof, "High speed 4-symbol arithmetic encoder architecture for embedded zero tree-based compression," *J. VLSI Signal Process.*, vol. 33, no. 3, pp. 267–275, Mar. 2003.

[27] B. Vanhoof, M. Peón, G. Lafruit, J. Bormans, L. Nachtergaele, and I. Bolsens, "A scalable architecture for MPEG-4 wavelet quantization," *J. VLSI Signal Process.-Syst. for Signal, Image, Video Technol.*, vol. 23, no. 1, pp. 93–107, Oct. 1999.

[28] M. Dyer, D. Taubman, and S. Nooshabadi, "Concurrency techniques for arithmetic coding in JPEG2000," *IEEE Trans. Circuits Systems I, Reg. Papers*, vol. 53, no. 6, pp. 1203–1213, Jun. 2006.

[29] L. W. Chew, W. C. Chia, L.-M. Ang, and K. P. Seng, "Very low-memorywavelet compression architecture using strip-based processing for implementation in wireless sensor networks," *EURASIP J. Embed. Syst.*, vol. 2009, p. 16, Jan. 2009.

[30] J. Jyotheswar and S. Mahapatra, "Efficient FPGA implementation of DWT and modified SPIHT for lossless image compression," *J. Syst. Arch.*, vol. 53, no. 7, pp. 369–378, Jul. 2007.

[31] P. Corsonello, S. Perri, P. Zicari, and G. Cocorullob, "Microprocessor-based FPGA implementation of SPIHT image compression subsystems," *Microprocess. Microsyst.*, vol. 29, no. 6, pp. 299–305, Aug. 2005.

[32] R. Stefo, J. L. Núñez, C. Feregrino, S. Mahapatra, and S. Jones, "FPGA-Based modelling unit for high speed lossless arithmetic coding," *Field-Program. Logic Appl. Lecture Notes Comput. Sci.*, vol. 2147/2001, pp. 643–647, 2001.

[33] K. M. Marks, "A JBIG-ABIC compression engine for digital document processing," *IBM J. Res. Developm.*, vol. 42, no. 6, pp. 753–758, Jun. 1998.

[34] S. Kuang, J. Jou, and Y. Chen, "The design of an adaptive on-line binary arithmetic coding chip," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 45, no. 7, pp. 693–706, Jul. 1998.

[35] H. Printz and P. Stubley, "Multialphabet arithmetic coding at 16 MBytes/sec," in *Proc. Data Compression Conf.*, Mar. 1993, pp. 128–137.

Kai Liu received the B.S. and M.S. degrees in computer science and the Ph.D. degree in signal processing from Xidian University, Xi'an, China, in 1999, 2002, and 2005, respectively.

Currently, he is an Associate Professor of computer science and technology with the Xidian University. His major research interests include VLSI architecture design and image coding.

Evgeniy Belyaev received the Master's (engineer) degree in automated systems of information processing and control and the Ph.D. (candidate of science) degree in technical sciences from the State University of Aerospace Instrumentation (SUAI), Saint-Petersburg, Russia, in 2005 and 2009, respectively.

He is the Research Scientist with the Laboratory of Information Technologies in Systems Analysis and Modeling, Saint-Petersburg Institute for Informatics and Automation, and Assistant Professor in SUAI. His research interests include real-time video compression and transmission, video source rate control, scalable video coding, motion estimation and arithmetic encoding.

Jie Guo received the B.S. degree in telecommunication engineering and the Ph.D. degree in information and communication engineering from Xidian University, Xi'an, China, in 2005 and 2010, respectively.

He is currently a Lecturer with the School of Telecommunication Engineering, Xidian University. His research interests include VLSI design and implementation for discrete wavelet transform and image coding.