International Journal of Current Engineering and Technology
E-ISSN 2277 – 4106, P-ISSN 2347 - 5161

Available at http://inpressco.com/category/ijcet

Research Article

Design of Coder Architecture for Set Partitioning in Hierarchical Trees Encoder

Meenu Roy[Å*] and N.Kirthika[Å]

[Å]Anna university (Chennai), Sri Ramakrishna Engineering College, Coimbatore, Tamilnadu.

Accepted 01 May 2014, Available online 01 June 2014, **Vol.4, No.3 (June 2014)**

Abstract

High performance Arithmetic Coder architecture is proposed in this paper for image compression. This arithmetic coder architecture is used in Set Partitioning. In Hierarchical Trees for further compression of the Discrete Wavelet Transform decomposed images. The architecture is based on a simple context model. Simple context model results in regular access pattern during reading the wavelet transform coefficients which is convenient to the hardware implementation. The arithmetic coder contains four core's to process different contexts and there is an out-of-order execution mechanism for different types of context is proposed that helps to allocate the context symbol to the idle arithmetic coding core with different order that of input. Several dedicated circuits such as common bit detector are used in the architecture to further improve the throughput. Common bit detector can unroll the renormalization stage of the arithmetic coding. For low and high updated values, the carry look-ahead adder and fast multiplier divider are also employed in the architecture which shortens the critical path. An adaptive clock switch mechanism is used which can stop some invalid bit planes clock for the power saving purpose according to the input images. Experimental result proves that the arithmetic coder architecture with four internal cores having similar architecture gives better performance as compared with single core architecture.

Keywords: *Arithmetic Coder(AC), Set Partitioning In Hierarchical Trees(SPIHT), Discrete Wavelet Transform(DWT), Context model, Common Bit Detector(CBD), Carry Look-ahead Adder, Fast multiplier/ divider, Critical path, Adaptive clock switch.*

1. Introduction

Compression is the art of reducing the number of bits needed to store or transmit data. Compression involves encoding information using fewer bits than original representation. The compression can be either lossy or lossless. Lossless compression technique reduces bits by identifying and eliminating statistical redundancy. In lossless compression no information is lost. Lossy compression technique reduces bits by identifying unnecessary information and removing it. Process of reducing the size of a data file is popularly known as data compression, but its formal name is source coding. Compression is very useful because it helps to reduce resource usage, like data storage space or transmission capacity. All data compression algorithms consist of at least a model and a coder. Model estimates the probability distribution and coder assigns shorter codes to the more likely symbols. Compression is very useful in most situations because the compressed data will save time and the space if it compared to the unencoded information it represent.

There are different types of coding techniques available for compression. Among those here using arithmetic coding technique because it will create code word for the entire data together, instead of creating code word for each data word.

Arithmetic coding is a well-known method for lossless data compression. Arithmetic coding is mostly popular in image and video compression applications. If we have a message composed of symbols over some finite alphabet, we can generate the exact number of bits that corresponds to a symbol. Arithmetic coding bypasses the idea of replacing an input symbol with a specific code. And it replaces a stream of input symbols with a single floating point output number. Arithmetic coding offers the opportunity to create a code that exactly represents the frequency of any character [glossary data compression ppt]. Arithmetic coding is the most efficient method to code symbols according to the probability of their occurrence. In contrast to a binary Huffman code tree the arithmetic coding offers a clearly better compression rate, as it produces a single symbol rather than several separate code words. In arithmetic coding, a message is encoded as real number in an interval from one to zero. Arithmetic coder generate a unique identifier or tag for the sequence of length m to be coded and will assign a unique binary code to this tag.The main data structure of arithmetic coding is an interval which representing the string constructed so far as its initial value is [0, 1]. At each stage, the current interval [min, max] is subdivided into

*Corresponding author **Meenu Roy** is a PG student and **N.Kirthika** is working as Asst. Prof.

sub-intervals corresponding to the probability model for the next character.The interval chosen will be the one representing the actual next character.The more probable the character, the larger the interval.The coder output is a number in the final interval. Here we are going to design an arithmetic coder which is to be used in Set Partitioning of Hierarchical Trees (SPIHT) encoder for further compression of the discrete wavelet based decomposed images. After the SPIHT transformation some regularities will exist in the file. These regularities may allow us to further compress the file. With this in mind we investigated the addition of arithmetic compression to a SPIHT encoded image.

SPIHT algorithm is one of the powerful algorithm for the compression. Wavelet transform SPIHT algorithm is used to encode the coefficients of the wavelet. The SPIHT sorting is done by comparing two elements at a time which results in yes/no states. According to the sorting pass coefficients are categorizes into 3 lists: LIS (List of Insignificant Sets) are the set of coefficients having magnitude smaller than the threshold .Then LIP (List of Insignificant Pixels) are the coefficients having magnitude smaller than the threshold. The last one is LSP (List of Significant Pixels) which are the pixels those magnitude is larger than that of threshold. The arithmetic coding method can obtain optimal performance for its ability to generate codes with fractional bits and it is widely used in various image compressions (J. Rissanen, *et al*, 1976), (J. Rissanen and G. G. Langdon, 1979). Especially, the set partitioning in hierarchical trees (A. Said and W. A. Pearlman, 1996) uses an Arithmetic coding method to improve its peak signal to noise ratio. Here we are designing an arithmetic coder for SPIHT to increase the performance and further result in good compression of the SPIHT. The main contributions of this architecture can be summarized as follows:

1) Simple context model which is based on the neighbor pixels' significant states is designed for hardware implementation.
2) Different context symbols formed according to the context model by SPIHT algorithm are processed in parallel by the arithmetic coder for the speedup purpose.
3) An internal memory array is used for the cumulative probability values in order to reduce memory size. Carry look-ahead adder (CLA) circuits are employed for the update of probability variables.
4) A dedicated adaptive power management module is used to stop clocks for the invalid bit-plane, which contains no information about the wavelet coefficients for power efficient design. The memory access pattern is also compacted for power saving purpose.

The remaining part of the paper is organized as follows: Section II describes the Principle of arithmetic coding, Section III describes about the entire architecture of arithmetic coder. In that part contain architecture of arithmetic coder and its core, then CBD structure. Section IV contains the tests and results that are performed and obtained. And the Section V end up with the conclusion of the work.

2. Principle of Arithmetic Coding

Arithmetic coding (H. Printz and P. Stubley, 1993) completely bypasses the idea of replacing an input symbol with a specific code. The arithmetic coder maintains two numbers, low and high. Initially low=0 and high=1. Low and High values further depends on the formula:-

range = high – low
low = low + range * low_bound (1)
high = low + range * high_bound

The low_bound and high bound can be determined by the formula:-

Low_bound = $\sum_{i=1}^{s-1} \Pr[i]$
High_bound = $\sum_{i=1}^{s} \Pr[i]$ (2)

Here we have to code symbol *s*, where symbols are numbered from 1 to *n* and symbol *i* has probability Pr[*i*]. The range between low and high is divided between the symbols of the alphabet, according to their probabilities. Procedure for arithmetic coding is given in the Fig. 1. In Fig. 1, it encodes the message sequence 'bac' and finally got the interval between .27 and .30. So, the final any value between 0.27 and 0.30, will uniquely encode the message 'bac'.

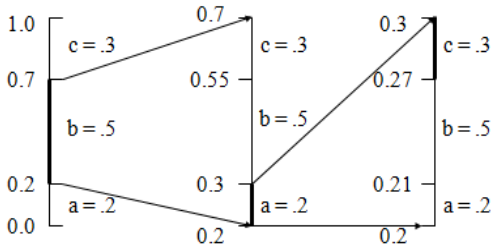

Fig.1 Arithmetic coding to encode message bac

In addition with the symbol probability, here we are taking an additional probability in normal arithmetic coding and it is called cumulative probability. Let cum_freq[i] be the cumulative counts for the symbol i, i.e., cum_freq[i] equals to $\sum_{j \le i} Pj$. Then the interval for the symbol i is [cum_freq[i-1], cum_freq[i]]. If the current probability interval is [low, high], then the update can be done by the following formula:-

Range = high-low

$$\text{High} = \text{low} + \text{range} \times \frac{\text{cum\_freq}[i-1]}{\text{cum\_freq}[0]} \quad (3)$$

$$\text{Low} = \text{low} + \text{range} \times \frac{\text{cum\_freq}[i]}{\text{cum\_freq}[0]}$$

For decreasing coding latency and avoiding registers underflowing, the normalization procedure is used as follows:-

1) If high HALF, where HALF , then a 0 bit is written into output bitstream.
2) If low HALF, then a 1 bit is written into output bitstream.

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. 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. As a result, 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. Then the context-based AC is widely used in the various fields. The context means conditions for current symbol which gives certain conditions for executing the symbol faster. 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. As a result different coefficients form different context windows using a preset model. And the different contexts will be sent to independent coding parts for updating the interval and emitting the code bits.

3. Architecture of Arithmetic Coder

The architecture of proposed arithmetic coder is shown in Fig.2. It consists of four identical cores and each core will work as an general arithmetic coder (K.Harika and K.V.Ramana Reddy, 2013) and executes parallel. When the context label and binary code symbol arrive, context switch differentiates the input context and sends the context value to the context dispatcher by different paths. And the main task of the context dispatcher is to schedule the order of the input contexts that are sent to the different cores for calculation. 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. There is a small buffer for context value which is set in the context dispatcher to implement reorganizing the processing order.

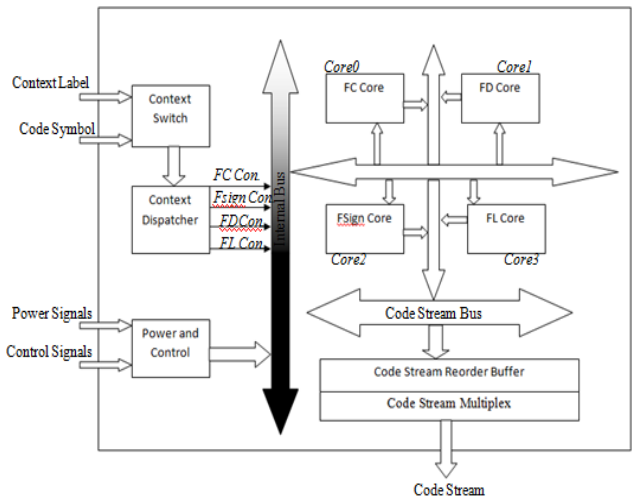

Fig.2 Architecture of proposed arithmetic coder

And each of four coding cores has its state register to indicate whether the coding core can receive new context. If there is no context in the buffer, the state of core is set to idle. When a context symbol arrives, the state of core is set to the context label which is used to block any new context. Then the dispatcher checks the states to find if there is an idle core. And 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 which is clearly shown in Fig.2. If the incoming context is blocked, then 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. And in the second clock cycle, two new contexts arrive. As one context pair is not finished, then the newly coming context pair is blocked by corresponding core to which it is being sent. Every code core works independently, which allows multi-contexts being calculated in parallel. Then the outputs of each core are connected with another bus. Code stream reorder buffer is used to sort the order of each code core as the execution order differs from the input order. And the code stream multiplex module collects all code bits emitted by the code cores and finally emitted to external by output ports.

3.1 Architecture of AC core

The internal structure of each code core is identical and its working is same as general arithmetic coder. Fig.3 shows the internal architecture of the AC core. 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. And then the correct label is transmitted to Boundary Update part. In boundary update part, the upper and lower bounds are computed by two different parts, they are, Upper Bound Update and Lower Bound Update.

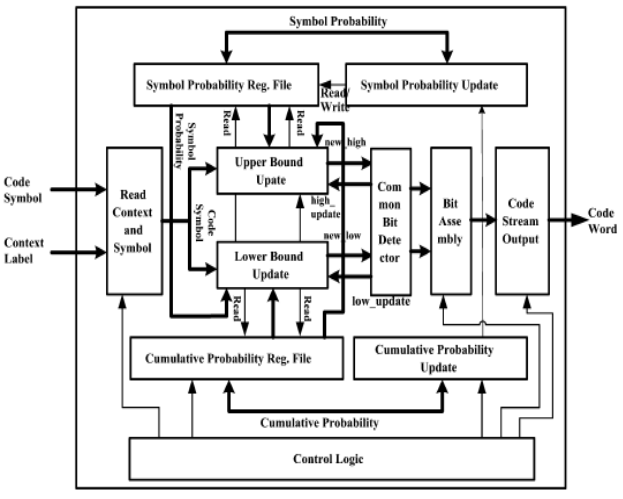

Fig.3 Internal architecture of AC core which functioning is as general AC coder

The coding symbol probability register file records the symbol probability values for the symbol. The cumulative probabilities cum_freq[i-1] and cum_freq[i] , which are stored in the cumulative probability register file, are accompanied with the old bound values high and low to compute new bounds for the probability interval. Both outputs of the update parts, i.e., new_high and new_low, are calculated by these variables based on the formula (3). These values are calculated by using carry look-ahead adder and fast multiplier/divider array (K.SivaNagi Reddy, V.Sidda Reddy and Dr.B.R.Vikram, 2012). For speedup purpose, a CLA and a fast multiplier array are employed and it helps to reduce the delay of critical path. 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. Finally the same bits are collected to form align code stream in the Bit Assembly. And these are supplied directly to the Code Stream Output part for emitting codes to the external bus. In the code stream output first the bits are stored in the buffer and then transferred to the output as code word. The Coding Control part is responsible for the whole code core's running and it sends various commands and control signals.

3.2 Common Bit Detector Structure

In Arithmetic coder, the code bits are generated by an internal loop, which is essential to the architecture design. Fig.4 shows the CBD structure which is used to unroll the internal loop. Inputs of CBD are low and high values after calculation of formula (3). The bit_valid_count signals the output bit count from the MSB to the LSB of bit_value register. And the bit_value is just a concatenation of common bits and bits_to_follow which is used for underflow. Then the low_update and high_update registers are shifted values for two bounds used for the next run.

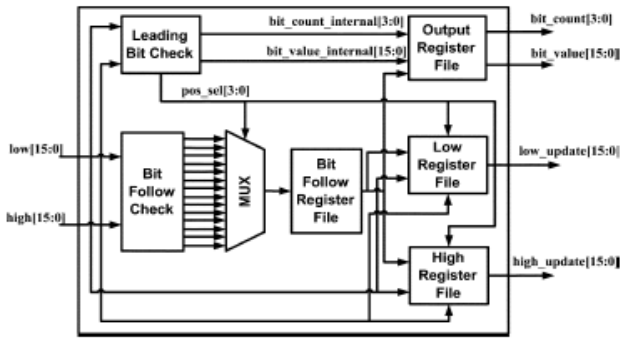

Fig.4 CBD Structure

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 (V. G. Oklobdzija, *et al*, 1994). The bit follow check (BFC) module is an array for checking the mode of underflow. In case for the speedup purpose, 15 possible cases for the bit follow check are checked, 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.5 shows the internal structure for the BFC.

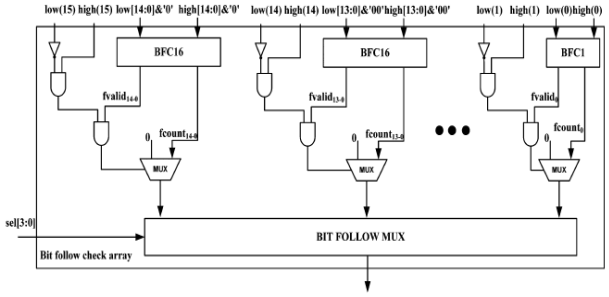

Fig.5 Internal Structure of BFC with 15 possible cases

The BFC denotes the bit follow check of two vectors that can be implemented by a simple logic gate and an LZD circuit, which is shown in Fig.6. And 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. Low and High registers are shifted left according to the values of pos_sel[3:0] and the bit_to_follow register. The output register emits the proper common bits and the underflow bits according to these registers.

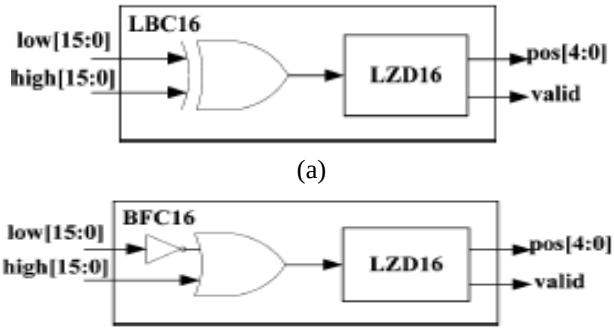

Fig.6 (a) LBC16 and (b) BFC16 Structures

4. Tests and Results

The design of arithmetic coder is described in VHDL and testing of individual module has been carried out. Each part of arithmetic coder has been simulated separately and tested the outputs. First of all the LBC16 and BFC 16 structures are design as per in Fig.6 and simulated. The simulation results of LBC16 is shown in Fig.7 and BFC16 is shown in Fig.8.

Fig.7 Simulation result of LBC16

In LBC16, the common bits are being detected and it shows the position of uncommon bits. Here in Fig.7 the

13th position of the low and high value differs and it shown in the simulation results. In BFC16, it shows the underflow bits, that means the position where the low bit is higher than the high bit. In Fig.8 it shows the position where the low bit position is higher than the high bit position. Next go for the simulation of 15 possible cases of BFC and the result is shown in Fig.9.

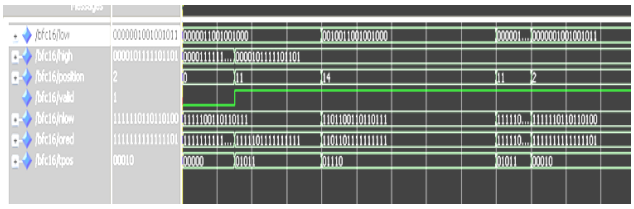

Fig.8 Simulation result of BFC16

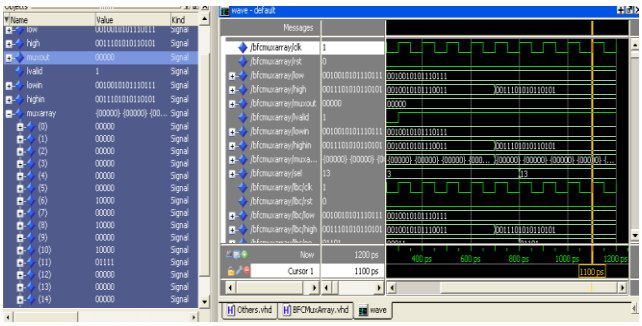

Fig.9 Simulation of internal structure of BFC

In internal structure of Bit Follow Check with 15 possible cases, the output depends upon the LBC output value. LBC output is given as the select line to the BFC mux in the BFC internal structure. And depending on that select value the output is generated. In Fig.9, the select line position is 13 so the 13th position value will be forced towards the output. The 13th position value is 0000 so it is forced towards the output value of mux and gets 0000 as output. Next we go for the simulation of core of the arithmetic coder and it is shown in Fig.10.

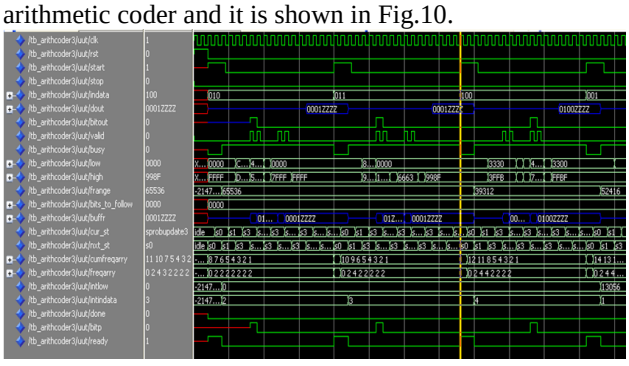

Fig.10 Simulation of Arithmetic coder's core structure

Fig.10 shows the simulation result of arithmetic coder's core structure which works as general arithmetic coder. Here the outputs are produced depending upon the bit out value. If the valid is positive, then the corresponding bit out value is sent to the buffer. The buffer stores that and produces the output in dout (output codeword). The output in buffer and final output is shown in simulation result as blue coloured line. Using this core, we are going to design a arithmetic coder with four similar core of the same above architecture. And the four cores will work in similar manner and parallel. Fig.11 shows the simulation result of arithmetic coder with four cores.

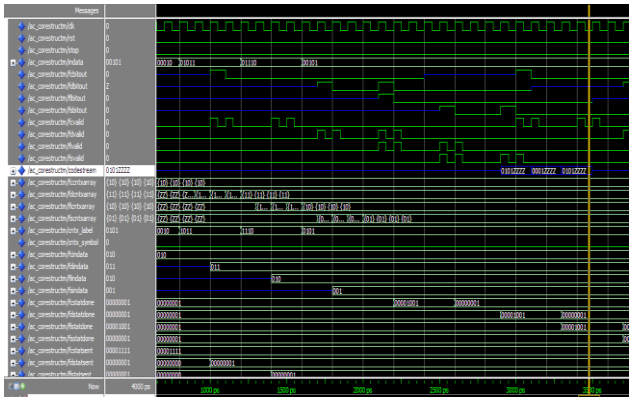

Fig.11 (a) Simulation result of Arithmetic coder

In Fig.11, the outputs from four cores are combined together and produced as one output in the code stream. It can be clearly viewed in the figure of simulation. Here the third and second bit will decide to which core it has been sent. And the zeroth, first and fourth bit will be sent to the corresponding core, which is responsible for producing the output code stream. Fig:12 shows the output produced by four cores separately.

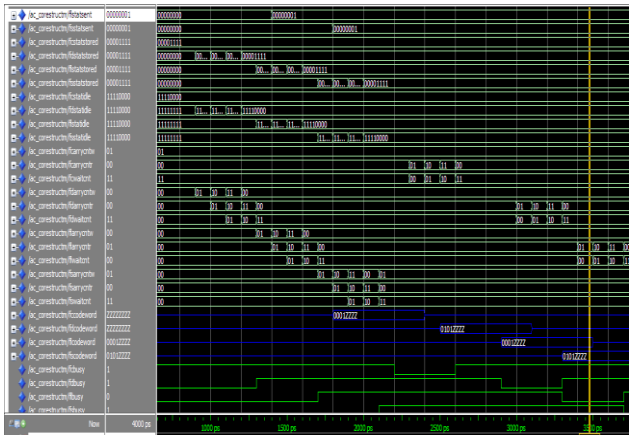

Fig.12 (b) Simulation result of arithmetic coder

In case of performing the proposed architecture with the general arithmetic coder architecture, we can find a tremendous decrease in time to produce an output in proposed AC. The time taken to produce four outputs for proposed and general architecture of arithmetic coder is shown in Table.1.

Table 1 Comparison of proposed and general AC performance

S.No	Architecture	Time Taken
1	Proposed AC	220000ps
2	General AC	720000ps

Conclusions

This paper presents implementation of arithmetic coder in VHDL. The simulation and implementation is carried out in VLSI lab of Nest Cyber Campus. The results make the following conclusions as given below:-

1) For improvement of throughput purpose, we proposed a high speed architecture of Arithmetic Coder with four cores.
2) In the proposed architecture, a simple context scheme is used to reduce memory and performance.
3) We are employed high speed calculation units for speed up 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.
4) From the simulation results, our AC architecture can meet many high speed compression requirements.
5) And comparing with general coder architecture, the performance of proposed architecture is improved.

Acknowledgement

The authors would gratefully acknowledges valuable guidance and support to carry out the work from Nest cyber campus, a branch of nest technologies. And the reviewers for their helpful comments and revisions.

References

www.binaryessence.com – Glossary Data compression.

J. Rissanen (1976), Generalized kraft inequality and arithmetic coding, *IBM J*. Res. Developm., vol. 20, no. 3, pp. 198–203.

J. Rissanen and G. G. Langdon (1979), Arithmetic coding, *IBM J*. Res. Developm., vol. 23, no. 2, pp. 149–162.

A. Said and W. A. Pearlman (1996), 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.

H. Printz and P. Stubley (1993), Multialphabet arithmetic coding at 16 MBytes/sec, *in Proc. Data Compression Conf*., pp.128–137.

K.Harika, K.V.Ramana Reddy (2013), Design and Implementation of Arithmetic Coder Used in SPIHT, *International Journal of Innovative Technology and Exploring Engineering (IJITEE)* ISSN: 2278-3075, Volume-3, Issue-3.

K.SivaNagiReddy, V.Sidda Reddy, Dr.B.R.Vikram (2012), Efficient Memory and Low Complexity Image Compression Using DWT with Modified SPIHT Encoder, *International Journal of Scientific & Engineering Research*, Vol 3, Issue 8.

V. G. Oklobdzija (1994), 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. 124–12.