# FPGA Implementation of VLSI Architecture For Data Compression

**Ravinder Tirupati**
Asst. Professor in ECE Department
MLRIT Hyderabad.

**CH. Neelima**
Asst. Professor in ECE Department
MLRIT Hyderabad.

***Abstract:*** This project aim is at increasing the security and compression. FPGA implementation of VLSI Architecture of Secure Arithmetic Coding improves the compression. In the traditional Secure Arithmetic Coding has no security.

Arithmetic Coding is method for lossless data compression. Arithmetic Coding is a Variable-length entropy encoding that converts a string into another representation that represents frequently used characters using fewer bits and infrequently used characters using more bits, with the goal of fewer bits in total. As opposed to other entropy encoding techniques that separate the input message into its component symbols and replace each symbol with a code word, arithmetic coding encodes the entire message to a single number. Although arithmetic coding provides no security as traditionally implemented. In this project modified scheme that offers both security and compression. The system utilizes an arithmetic coder in which the overall length within the range [0, 1) allocated to each symbol is preserved. Permutations are applied at the input of the Encoder for the security. The overall system provides a simultaneous security, compression.

*Key words: security, permutation, compression, encoding and decoding, VHDL coding, spanton 3 kid.*

## I. INTRODUCTION

Arithmetic coding has been developed extensively since its introduction several decades ago, and is notable for offering extremely high coding efficiency. While many earlier-generation image and video coding standards such as JPEG, H.263, and MPEG-2 relied heavily on Huffman coding for the entropy coding steps in compression, recent generation standards including JPEG2000 and H.264 utilize arithmetic coding. This has led to increased interest in arithmetic coding both in the context of image coding, and also more generally for other applications. While arithmetic coding is extremely efficient, and others have noted, as traditionally implemented it is not particularly secure. The issue of providing both compression and security simultaneously is growing more important given the increasing ubiquity of compressed media files in a host of applications including the Internet, digital cameras, and portable music players, and the common desire to provide security in association with these files. When both compression and security are sought, one approach is to simply use a traditional arithmetic coder in combination with a well-known encryption method such as the Advanced Encryption Standard (AES). However, while this will certainly meet both goals, it fails to take advantage of the additional design flexibility and potential computational simplifications that are available if the coding and encryption are performed jointly.

Traditional arithmetic coding provides essentially no security in the face of a chosen plaintext attack, in which an attacker has the ability to specify a sequence of input symbols and observe the corresponding output, and to repeat this process an arbitrary number of times. For example, in a binary system with two symbols A and B, it is a simple matter to choose input sequences that, in combination with their outputs, reveal the assumed probabilities of each symbol in the arithmetic coder as well as the order of the intervals. That information can then be used to decode any output from the encoder.

The issue of increasing the security of arithmetic coding has received relatively little attention in the literature. Bergen and Hogan have considered the problem of inferring the underlying symbol probabilities and partitioning of the [0,1) interval using observations of an arithmetic encoder output. Liu presented a system using table-based bit sequence substitutions to provide encryption during arithmetic coding. More recently, a randomized arithmetic coding (RAC) system based on random swapping of the two intervals in a binary arithmetic coder was described by Grangetto who utilized this approach to encrypt JPEG 2000 coded images. The systems in [7] and [8] modify the traditional arithmetic coder by randomly permuting the intervals in accordance with a key-generated shuffling sequence. The shuffling sequence consists of one bit per encoded symbol that determines whether the binary intervals are swapped or not when encoding that symbol. The authors of that paper were targeting applications to JPEG2000-encoded images in which a potential attacker would not have access to the original image nor be in a position to provide a particular image to be encoded. Thus, robustness to plaintext attacks was not a goal in [7] and [8]. Indeed, if an attacker of the RAC was granted access to the RAC encoder, removed from the larger JPEG2000 context for which it was designed, the number of trials needed to determine an N-bit shuffling sequence would be on the order of N, since the N output pairs N corresponding to inputs that differ in exactly one symbol can be compared. Such comparisons, however, would not reveal the underlying key used to generate the shuffling sequence, so if care was taken to modify the key or avoid re-initialization of the shuffling sequence in subsequent uses of the RAC encoder, substantially higher robustness would result. In [9], we specifically consider the goal of encryption, and describe an arithmetic coding (Interval Splitting AC) approach in which the intervals associated with each symbol, which are continuous in a traditional arithmetic coder, can be split according to a key known both to the encoder and decoder. This removes the constraint that the intervals corresponding to each symbol be continuous, and instead uses a more generalized constraint that the sum of the lengths of the one or more intervals associated with each symbol be equal to its probability.

The present work aims to provide an arithmetic coding system that is secure against a chosen plaintext attack. Interval splitting within the arithmetic coder and permutation steps at the input and output are utilized to construct a system with security that increases exponentially with the length of the shorter of the input block size and the key sequence. While all of the methods described here can be applied for coding of source alphabets with any size, we address the case of binary systems here to simplify the discussion and illustrations.

## II. SYSTEM DESCRIPTION

The block diagram of the secure arithmetic coding system. The system consists of a first permutation step applied to the input sequence, arithmetic coding using interval splitting, and a second permutation step applied to the bits produced by the coder. A key sequence is input to a key scheduler which in turn provides key information to both permutation steps and to the interval splitting arithmetic coder. The key scheduler utilizes information from the split AC encoder output. The permutation steps in this system are similar to the Shift Row Transformation in AES in that the rows of a block of data are shifted cyclically. The main difference is that in the system of Fig. 1 row and column shifts are shifted cyclically by different amounts according to key values, in contrast to the corresponding step in AES in which the shifts are predetermined [10].

**Data Compression Techniques:**

Data compression is the "compression ratio", or ratio of the size of a compressed file to the original uncompressed file. For example, suppose a data file takes up 100 kilobytes (KB). Using data compression software, that file could be reduced in size to, say, 50 KB, making it easier to store on disk and faster to transmit over an Internet connection. In this specific case, the data compression software reduces the size of the data file by a factor of two, or results in a "compression ratio" of 2:1.

1. Lossless data compression
2. Lossy data compression

Lossless data compression is used when the data has to be uncompressed exactly as it was before compression. Text files are stored using lossless techniques, since losing a single character can in the worst case make the text dangerously misleading. Archival storage of master sources for images, video data, and audio data generally needs to be lossless as well. However, there are strict limits to the amount of compression that can be obtained with lossless compression. Lossless compression ratios are generally in the range of 2:1 to 8:1.

Lossy compression, in contrast, works on the assumption that the data doesn't have to be stored perfectly. Much information can be simply thrown away from images, video data, and audio data, and when uncompressed such data will still be of acceptable quality. Compression ratios can be an order of magnitude greater than those available from lossless methods.

The question of which is “better”, lossless or lossy technique, is pointless. Each has its own uses, with lossless techniques better in some cases and lossy techniques better in others. In fact, as this document will show, lossless and lossy techniques are often used together to obtain the highest compression ratios.

**Introduction to Arithmetic Coding:**

Before jumping into the fray and starting with the explanation of the encoding algorithm, first we introduce some basic terms commonly used in data compression. They will be used throughout the whole paper. Our goal is to compress data, which might either be stored on a computer-readable media or be sent over some form of stream. This data could represent anything, reaching from simple text up to graphics, binary executable programs etc. However, we do not distinguish here between all those data types. We simply see them all as binary input. A group of such input bits is what we will refer to as a symbol. For instance one could think of an input stream being read byte wise, leading to 28 = 256 different input symbols. For raw text compression, it could also suffice to take an alphabet of 128 symbols only, because the ASCII code is based on a 7-byte structure.

**Foundations:**

DEFINITION 1 (ALPHABET AND SYMBOL)

We call a finite, nonempty set an ALPHABET. The LENGTH or cardinality of an alphabet A will be referred to as |A|. The elements {a1, . . . ,am} of an alphabet are called SYMBOLS.

Also we assume that *A* is an ordered set, so giving {a1, . . . ,am} a distinct order. We already mentioned above that the Arithmetic Coding algorithm works sequentially. Thus we need some notion of what the sequential input and output of the encode/decoder might look like. This leads us directly to the notion of a SEQUENCE:

DEFINITION 2 (SEQUENCE)

A series S = (s1, s2 . . . ) of symbols si from an alphabet A is called SEQUENCE. In the latter we will also use the shortcut S = s1s2 . . .

In analogy to the definition of |A|, |S| is the symbol for the length of S, provided that S is of finite length. However, |S| < ¥ will be a general assumption henceforth, since most of the corollary would otherwise make no sense. Please note that this representation of data is somehow natural, since most human-made media can be read in a sequential order. Just think of books, videos, tapes and more. Also, when looking at a sequence, one can calculate a distinct probability of each symbol of the alphabet to occur in this very sequence. This probability might be very unevenly distributed, a lot depending on the application domain. For instance consider the letter e, which is much more common than z in the English language.3 Since Arithmetic Coding depends a lot of such statistical measures in order to achieve compression, we introduce the PROBABILITY of a symbol as follows:

DEFINITION 3 (PROBABILITY)

Let S = (s1, . . . , sn) a finite-length sequence with |S| = n over A = {a1, . .,am}.

Also let $|S|_{ai}$ the frequency of ai in S. Then we define P(ai) :=|S|ai/n as the PROBABILITY of ai (in S).

From the definition, we can directly conclude that P(ai) is always contained in the interval [0,1) for any symbol, whereas the sum over all such probabilities is always $\sum_{i=1}^{m} P(ai) = 1$.

Please note that this interval is open-ended, because it would make no sense to encode a constant sequence holding only a symbol of probability 1, simply because in that case the full content of the sequence would have been known beforehand already. We will later on make use of this property in certain conclusions.

Recapturing the example of e/z however, we would like to emphasize that the probability of a symbol might heavily depend on its context. If one considers e and z as symbols for bytes in a binary executable for example, they might be rather evenly distributed. Also one could even show that certain symbols are more likely to occur in scientific text than newspaper articles and so forth. Some data is subject to interpretation: E.g. consider the sequence 1111131311. It could be interpreted as a sequence of symbols 1,3 or 11,13. At least this example proves that we need some kind of unambiguous rule of how probabilities are related to symbols. This relation between symbols of an alphabet and their probability is commonly known as a MODEL in terms of data compression.

DEFINITION 4 (MODEL)

Let A an alphabet. A MODEL M is a function

$$M : A \longrightarrow [0,1) : ai \longrightarrow PM(ai) ,$$

which maps a probability PM(ai) to each symbol ai $\in$ A.

This probability might be estimated / calculated and does not necessarily have to match the real probability of the symbol, P(ai). Indeed in most cases it does not. Please also note that an alphabet is not restricted to only hold symbols of length 1. In the example above, employing 11 and 13 as symbols we already got a picture of that. If one estimates the probability of a given symbol not only by looking at the symbol itself but also at the context given by the last n symbols seen, one speaks of an Order -n model. For instance the average probability of the letter u to occur in any German text is only about 0.0435. If one considers its probability to occur after the letter q however, this value raises to nearly 1! As one can see already now, an increased value of n might lead to better predictions of probabilities.

As already briefly mentioned above, the probability distribution that is given by the interpretation of a sequence under a certain model, matches the real probability distribution at best by chance. Usually this will not be the case. For instance there will be almost no German text fulfilling the distribution given by Table 1 exactly, but rather approximately or even worse. To distinguish the probability induced by the model from the real one, we label the former with $P_M(ai)$ in order to emphasize the dependency of the model and in order to distinguish from the latter, given by P(ai).

So we conclude that a model can be seen as an interpretation of an arbitrary dataset. A simple model could for instance be given by the probability distribution of Table 1. This table shows the probabilities of most letters of the German alphabet to occur in an average German text. Probably the clever reader can already anticipate now, that the compression ration will heavily depend on how good this model matches the reality.

This leads to the need to define some kind of measure of compression, enabling us to actually compare the efficiency of different compression approaches. A natural measure of how much information is contained in a given sequence of data is called the ENTROPY.

DEFINITION 5 (Entropy)

Let S a sequence over alphabet A = {a1, ...,am}. The ENTROPY $H_M(S)$ of the sequence S under model M is defined as

$$H_M(S) = \sum_{i=1}^{m} P(ai) \log_2 \left(\frac{1}{P(ai)}\right)$$

The unit of the entropy is [bits/symbol] because the formula only refers to probabilities as relative frequencies rather than absolute ones.

By the formula one can easily see that with our definition, the entropy of a sequence depends on the model M being used, since the $P_M(ai)$ are the probabilities under that model. Here, $\log(1/P_M(ai))$ can be interpreted as the minimal length of a binary symbol for ai, while the factor P(ai) (being the real probability of ai) can be interpreted as probability of requiring the encoder to binary encode this very symbol.

Considering a model as perfect, one obtains the correct probability distribution leading to the natural form of the entropy:

$$H(S) = \sum_{a \in A} P(ai) \log_2 \left(\frac{1}{P(ai)}\right)$$

This kind of entropy is depended on the input data only and no subject to interpretation. However the interested reader might wish to know that most of the literature about Arithmetic Coding sloppily does not distinguish between both kinds of entropy.

**Encoder and Decoder:**

DEFINITION 6 (Encoder & Decoder)

An algorithm which encodes a sequence is called an ENCODER. The appropriate algorithm decoding the sequence again is called a DECODER. In opposite to the input sequence S we refer to the encoded sequence which is output of the encoder and input for the decoder by Code(S) or C(S) for short. The application of both algorithms is referred to as ENCODING respectively DECODING. We want to emphasize that we use the notion of an algorithm in its most natural way, meaning a general sequence of steps performed by any arbitrary computer. By purpose we do not limit ourselves to a certain implementation at this stage. An encoder could be any algorithm transforming the input in such a way that there is a decoder to reproduce the raw input data. However at the end of this paper we present the full VHDL source code of a encoder/decoder pair (also referred to as CODEC), which employs Arithmetic Coding. The following code examples are taken from this reference implementation.

In the theory of data compression one often distinguishes between lossy and lossless compression algorithms. Especially analogous signals are often encoded in a lossy way because such data is in the end meant to be interpreted by some kind of human organ (eye, ear,...) and such organs are very limited in a sense that they simply do not recognize certain levels of noise or distortion at all. Of course lossy compression algorithms can reach better compression ratios by losing some accuracy. However we are not going to consider any lossy compression in this article and rather concentrate on lossless compression, that can be applied to all kinds of data in general. Thus we are only going to consider codecs that are able to reproduce the input data up to the last symbol. In a nutshell our resulting Code(S) will be proven lossless and optimal.

**Design of Secure Arithmetic Coding:**

The main object of the Secure Arithmetic Coding is to implement Security as well as Compression of the given input data for the data communication. The compression is done by Entropy Encoding based Arithmetic coding. In the Arithmetic Coding probabilities are calculated for the calculation of Entropy (H) units are bits per symbol.

**Specifications:**

- Security
- Compression(Arithmetic Coding)
- Encoding
- Decoding

**Random sequence generation:**

Input data length is 16bit data, this 16bit data is generated by the LFSR (Linear Feed Back Shift Register). In this LFSR 16 Flip-Flops and 3 xor gates are present to generate 16bit data which is used as a input data. In this LFSR D-FFs are used to store and generation of input data for the random 16bit sequence generator

**Data Security**:

In the Secure Arithmetic Coding project, Arithmetic Coding itself provides security but further security purpose permutation based operation which is shown in fig 3.2 is performed on input data. In this project input data length is 16-bit data, in order to keep input data secrecy, applying column and row circler shift on the 4x4-matrix form of input data. Circler shifting operation is based on key which is length of 8-bits.

For example

| | |
|---|---|
| Input 16 bit data | 1100000011010101 |
| Column circular shift key | D1 = 10110001 |
| Row circular shift key | D2 = 01110110 |

In the input data 4x4-matrix, 4 columns and 4 rows are present Circler shifting range maximum is 3 and minimum is 0 circler shifts, bit representation 2bit for each column and row. In the above D1 is for column shift and D2 is for row shift for the matrix.

Input data is secured by this permutation based way, this secured data is giving to Arithmetic Coding process as a input.

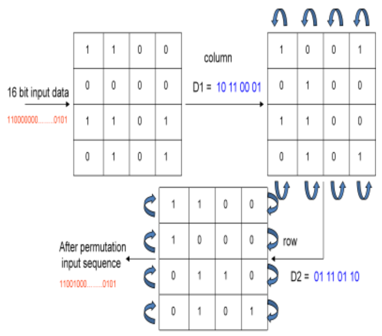


**Fig**: Permutation operation on the input data

**Data Compression:**

Data Compression comes under lossless data compression technique as already described. In this compression technique Arithmetic Coding is one of the techniques from Entropy Encoding. This Entropy Encoding is a one type which is from lossless data compression technique.

The Arithmetic Coding is completely depends on the Entropy calculation. In the Entropy calculation, input taken as 16-bit data which is come from the permutation output.

**Entropy:**

Entropy nothing but, representation of the symbol by the binary bits that means for one symbol how many bits are used to represent, the entropy units are bits/symbol. The Entropy(H(S)) calculations are done by the mathematical formula which is given in the below

$$H(S) = \sum_{i=1}^{m} P(ai) \log_2 \left(\frac{1}{P(ai)}\right)$$

Where p(ai) represents probability of the symbol that is in the given input sequence how many times the symbol repeats the probability will tell. Log base 2 represents the symbol with 2 binary bits. Summation for calculation of entropy value that is for i=1 to m.

**Symbol Identification:**

In the given input 16bit sequence 00,01,10,11 bits are represented with $z_0$, $z_L$, $z_G$, $z_1$ symbols respectively. Those given input 16bit sequence, what are the symbols present by the diagram symbol identification which is shown in fig 3.3. Here NOR, AND, NOT gates are present for identification of symbols. For this circuit 2bits are giving as input, to identify the symbol. Form 16bit sequence each 2 bits are giving as input, to identify the symbols for the entire 16 bit sequence. How many symbols are present in the 16bit sequence finding out by the symbol identification circuit.

Symbols representation

00 – $z_0$
01 – $z_L$
10 – $z_G$
11 – $z_1$

**Symbol Count:**

Symbol Count diagram which is shown in fig 3.4 counts the symbols, that is how many symbols are present in the 16bit sequence. Symbol count operation one counter for counting perpes and multiplexer is used. Multiplexer operation is to select the 2bits at a time and it gives to the symbol count. Finally the symbol count will count the symbols present in the input sequence. The results of the symbol count block are z0, zL, zG, z1 respectively.

**Probability Calculation:**

In the given sequence, occurring of symbols or repeated symbols value will tell by the calculation of probability. Probability value occurring in between 0 and 1 that is [0, 1). In the probability calculation every symbol value lies between 0 and 1. In above mentioned [0, 1) means occurring of probability value must be less than zero(0) and less than or equal to one(1). In the probability calculation design predefined the values of all 8 symbols are kept in the LUT (Look Up Table), from the LUT corresponding values for every symbol probabilities are taken respectively.

Probability calculations are done for z0, zL, zG, z1 respectively, with the below fig 3.5. In the LUT zero to seven address locations are present and in that address location 1/8, 2/8, 3/8, 4/8, 5/8, 6/8, 7/8, 8/8 values are present respectively. Whenever a symbol get repeated or unrepeated for that particular symbol get the corresponding probability value from the LUT respectively. Then the resultant values are P0, PL, PG, P1 are outputs for the probability calculation. In this diagram fig 3.5 same as previous diagram fig 3.4 but here taking the probability values extra.

All the calculated probability values are giving as input to the Entropy calculation diagram which is shown in the fig 3.6. Entropy calculation value is bits per symbol. That is for one symbol how many bits are used to represent that particular symbol. Here representation of one symbol with 2bits before Entropy calculation. After the Entropy calculation bits to represent a symbol are must be less than 2bits which are used to represent before Entropy calculation, then only compression of bits to represent a symbol is down.

**VLSI Architecture for Entropy Calculation:**

Entropy calculations are down by the formula

$$H(S) = \sum_{i=1}^{m} P(a_i) \log_2 \left(\frac{1}{P(a_i)}\right)$$

Here probabilities calculated from the fig 3.5 that is probability calculation block diagram. Here total symbols are 8, total number of input sequence is 16bits, and for one symbol representation will take 2bits as already described above.

In the Entropy calculation $\log_2 \left(\frac{1}{P(a_i)}\right)$ is calculated, and kept in the LUT. The result of $\log_2 \left(\frac{1}{P(a_i)}\right)$ and p(ai) are multiplied and the final result is Entropy value. In the fig 3.6 shows Entropy calculation value which is compressed the entire 16bit sequence into one number represented in base b for one symbol. To calculate the Entropy value to 8 symbols, then Entropy value per one symbol multiplied with 8 symbols then the resultant value is, if the resultant value is below the sequence 16bit length then that is compressed otherwise not compressed.

For example

**Developed Architecture for Compression:**

The Secure Arithmetic Coding will give the input data compression that is how many bits are reduced from the given input data sequence. Here below shown fig. is VLSI architecture for the Compression of input data sequence. After getting Entropy value multiplied by total number of symbols then the resultant value that is, it represents the bits which are present after the compression. In this arithmetic coding fixed number of bits are used at the input that input sequence length is 16bit, this 16bit input sequence length and compressed bits are subtracted then the resultant bits are the bits, which are compressed from the total number of input bits. In the above example after compression, bits are 14bits and the input bits are 16bits then these bits are subtracted resultant bits are 2bits. These operations are down by the developed architecture for compression.

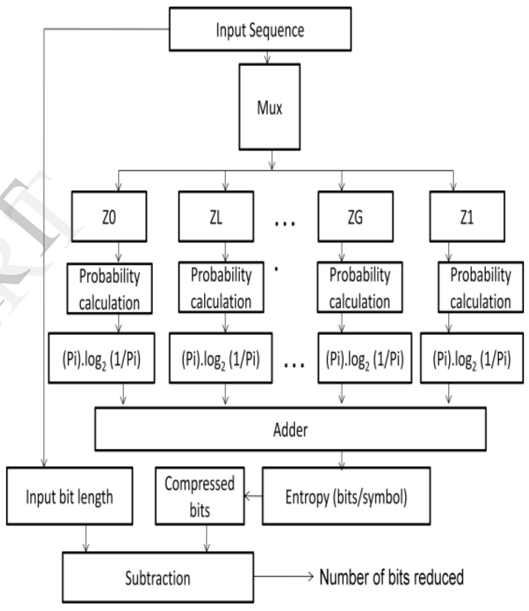


**Fig:** Developed Architecture for Compression

**Encoding:**

Arithmetic Coding uses a one-dimensional table of probabilities instead of a tree. It always encodes the whole message at once. This way it is possible to encode symbols using fragments of bits. However, one have cannot access the code word randomly. Using Huffman-coding, one can specify marks that allow decoding starting within the bit stream. Of course one can also split messages in arithmetic coding, but this limits the efficiency since use of bit-fragments on the boundaries is impossible.

In the Encoding process after getting the probabilities of the symbols, all probabilities fall into the range [0, 1) while their sum equals 1 in every case. This interval contains an infinite amount of real numbers, so it is possible to encode every possible sequence to a number in [0, 1). One partitions the interval according to the probability of the symbols. By iterating this step for each symbol in the message, one refines the interval to a unique result that represents the message. Any number in this interval would be a valid code.

The probability P($a_i$) to each symbol ai that appears in the message. Now we can split the interval [0, 1) using these values since the sum always equals 1. The size of the i−th sub-interval corresponds to the probability of the symbol ai.

For the above example S = "1111010100100101"

Let the probabilities of the symbols in the message be

$P_0 = 0.125$; $P_L = 0.5$; $P_G = 0.125$; $P_1 = 0.25$;

Now the interval [0, 1) would be split as emphasized

**Upper and lower bounds:**

The upper and lower bounds of the entire current interval called as high and low. The bounds of the sub-intervals are calculated from the cumulative probabilities:

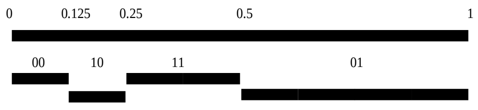


**Fig :** Creating an interval for the symbols

The values high and low change during the encoding process whereas the cumulative probabilities remain constant6. They are used to update high and low. With respect to the previous example, we get the following values: Subdivision of probabilities in the above equation for the intervals constant table containing the cumulative probabilities K($a_i$).

$$K(a_k) = \sum_{i=1}^{k} P(a_i)$$

| High 1.0 | K(0) 0.0 | K(2) 0.25 |
|---|---|---|
| Low 0.0 | K(1) 0.125 | K(3) 0.5 |

**Table:** Cumulative probabilities of the symbols

The first step in encoding is the initialization of the interval I := [low, high) by low =0 and high = 1. When the first symbol $S_1$ is read, the interval I can be resized to a new interval I′ according to the symbol. The boundaries of I′ are also called low and high. We choose I′ to equal the boundaries of $S_1$ in the model. However, how are these boundaries calculated? Let $S_1 = a_k$ be the kth symbol of the alphabet. Then the lower bound is

$$low = \sum_{i=1}^{k-1} P(a_i) = K(a_{k-1})$$

And the Upper bound is

$$high = \sum_{i=1}^{k} P(a_i) = K(a_k)$$

The new interval I′ is set to [low, high). This calculation is nothing new, it just corresponds to the mathematical method of the construction of Figure 3.8. The most relevant aspect of this method is that the sub-interval I′ becomes larger for more probable symbols $S_1$. The larger the interval the lower the number of fractional places which results in shorter code words. All following numbers generated by the next iterations will be located in the interval I′ since base interval as used [0, 1) before.

Next proceed with the second symbol $S_2 = a_j$ . However, now the problem that the partition[7] of the interval [0,1), not of I′ which was calculated in the previous step. Scaling and shifting the boundaries to match the new interval. Scaling is accomplished by a multiplication with high−low, the length of the interval. Shifting is performed by adding low. This results in the equation

$$low' = low + \sum_{i=1}^{j-1} P(a_i)(high - low) = low + K(a_{j-1})(high - low)$$

$$high' = low + \sum_{i=1}^{j} P(a_i)(high - low) = low + k(a_j)(high - low)$$

This rule is valid for all steps, especially the first one with low = 0 and high−low = 1. Since no need of the old boundaries any more for the next iterations, overwrite them with new boundaries:

$$low = low'$$

$$high = high'$$

These above low' and high' equations are calculated

**Developed Architecture of Encoder Low value:**

In the process of Encoding for each symbol upper and lower boundaries are calculated by the low' and high' equations. Those equations are developed by the below diagrams.

In the Encoder low value calculation, probability values are taken from the cumulative probabilities of the symbols which are shown in the table 2. Starting low and high values are taken as 0 and 1 respectively for the first step. And for the next steps low and high values are updated from the calculation of the low as well as high equations. In the Encoder low value diagram high and low values are subtracting, and the resulting value and the cumulative probability value are multiplied. Then the resultant value and the previous low value are added then this is the final value of low' value. Further calculation of low values for the remaining symbols, this low values is updating. In the updating process, adder having feed back with the register. Here calculated low value giving feed back to the low which is used for the subtraction with the updated value of high value.

**Developed Architecture for Encoder High:**

The same operation is done for the Encoder high value calculation which is shown in Encoder low is done for the Encoder low value.

Here also high value which is coming as output, giving as feed back to the input high value. Here also the same operations subtraction and addition feed back with register. But here the cumulative probabilities are taking differently that is depends on the upper and lower boundaries which are taking from the cumulative probabilities.

**Developed Architecture for Encoder:**

All the values of low and high values for the entire symbols, that is for each symbol its corresponding low and high values are stored in the LUT2(Look Up Table). In this Encoder low, high and LUT2 are used. Inputs to these blocks are probability, low and high values. The output of this block is Encoded value; in the LUT2 for each symbol corresponding high and low values are stored along with that symbol. All the values are stored in the LUT2 for the purpose of Decoding.

**Developed Architecture for Decoder:**

To decode a sequence, one somewhat have to apply the encoder backwards. The value V = Code(S) is given and with that V restore the original sequence *S.* assume that the message length is known and equals l. In the first iteration we compare V with each interval I′ = [K(ak −1),K(ak)) to find the one that contains V. It corresponds to the first symbol of the sequence, $S_1$. To compute the next symbol, modify the probability partition in the same way we did while encoding:

$$low' = low + K(a_{i-1})(high - low)$$

$$high' = low + K(a_i)(high - low)$$

Where I have to comply

$$low \le V \le high$$

$a_i$ is the next symbol in the encoded sequence. This time, the start case is again a special case of the general formula. The iteration is very similar to the encoder.

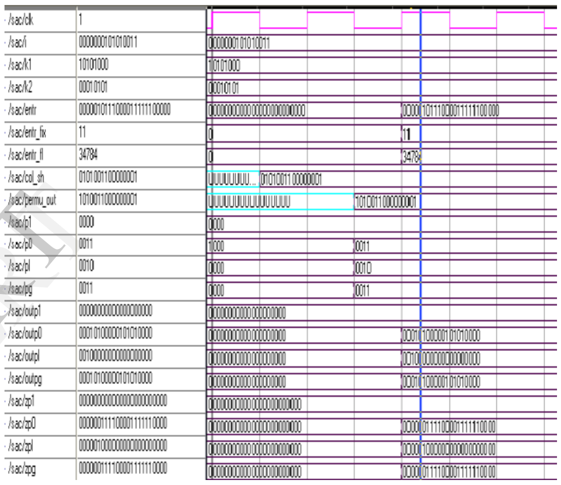
In the developed architecture for the decoder diagram, in this LUT2, comparator and assumed value(V) are present. In the LUT2 all the encoded values are stored.

Here assumed value(V) is taking as a average value of all low and high values. With V compare the low and high values which are in the LUT2 for each encoded symbol. If the V present in between the low and high value then that corresponding symbol is read back. In this manner all the encoded symbols are read back with the same procedure.

## III. Experimental Results

**Compression results:**

Here the wave form fig shows the pre-route simulation of the compression block diagram.



**Design Implementation summary of the Secure Arithmetic Coding:**

The **Table 2** shows the Design implementation summary of the Secure Arithmetic Coding (SAC). This gives a detailed analysis of the number of resources available in the FPGA. This **Table 2** shows the number of resources utilized by the proposed architecture. It consists of the Logic Utilization summary, Logic Distribution summary, timing summary. In this Logic utilization summary consists of usage of slices, slice Filp-Flops, Luts, Bonded IO's etc. The Logic Distribution summary shows the Additional JTAG gate count for IOBs. The timing summary shows the maximum frequency that can be generated by using this architecture.

**Design Implementation summary of the Secure Arithmetic Coding:**

**Table 2:** Design Implementation summary of the Secure Arithmetic Coding

| | | | | |
|---|---|---|---|---|
| **Logic Utilization**: | | | | |
| Number of Slice Flip Flops: | 20 | out of | 66560 | 1% |
| Number of 4 input LUTs: | 33 | out of | 66,560 | 1% |
| Number of IOs | 1328 | | | |
| **Logic Distribution:** | | | | |
| Number of Slices: | 19 | out of | 33,280 | 1% |
| Number of Slices containing only related Logic: | 19 | out of | 19 | 100% |
| Number of Slices containing only unrelated Logic: | 0 | out of | 19 | 0% |
| Number of 4-input LUTs: | 8265 | out of | 88192 | 9% |
| Number of bonded IOBs: | 668 | out of | 784 | 85% |
| Number of GCLKs: | 2 | out of | 8 | 25% |
| Total equivalent gate count for design: | 1,417 | | | |
| Additional JTAG gate count for IOBs: | 32,064 | | | |
| Peak Memory Usage:396 MB | | | | |
| Total REAL time to MAP completion: | 33 s | | | |
| Total CPU time to MAP completion: | 32 s | | | |
| **Timing Summary:** | | | | |
| Maximum output required time after clock | 69216 ns | | | |

## IV. CONCLUSION

In the present work developed VLSI Architecture for Secure Arithmetic Coding can be implemented using FPGA based Entropy calculation, Encoding and Decoding. The Secure Arithmetic Coding is used to compress the given input data and also providing the security to the input data. The security provided by applying permutations to the input data. Compression is done by the Entropy calculation. To get the input data back without error which is compressed, with the Encoding and Decoding operations. By this Secure Arithmetic Coding channel band width is reduced by compressing the input data. The VLSI architecture of the proposed Secure Arithmetic Coding is designed using VHDL and is implemented using Xilinx ISE navigator and modelsim using Spartan3 device.

In future, the present work can be extended for image compression, reduction in the chip area can be achieved this may required a slight increase in hard ware requirement.

## V. REFERENCES

[1] G. Langdon and J. Rissanen, “Compression of black-white images with arithmetic coding,” IEEE Trans. Commun., vol. COM-29, no. 6, pp. 858–867, Jun. 1981.

[2] D. S. Taubman and M. W. Marcellin, JPEG2000: Image Compression Fundamentals, Standards and Practice. Norwell, MA: Kluwer Academic, 2002.

[3] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, “Overview of the H.264/AVC video coding standard,” IEEE Trans. Circuits Syst. Video Technol., vol. 13, no. 7, pp. 560–576, Jul. 2003.

[4] J. Cleary, S. Irvine, and I. Rinsma-Melchert, “On the insecurity of arithmetic coding,” Comput. Secur., vol. 14, no. 2, pp. 167–180, 1995.

[5] H. Bergen and J. Hogan, “A chosen plaintext attack on an adaptive arithmetic coding algorithm,” Comput. Secur., vol. 12, no. 2, pp. 157–167, Mar. 1993.

[6] X. Liu, P. Farrell, and C. Boyd, “Unified code,” in Proc. Int. Conf. Cryptography Coding. Berlin, Germany: Springer-Verlag, 1999, vol. 1746, Lecture Notes in Computer Science, pp. 84–93.

[7] M. Grangetto, A. Grosso, and E. Magli, “Selective encryption of JPEG2000 images by means of randomized arithmetic coding,” in Proc. IEEE 6th Workshop on Multimedia Signal Processing, Siena, Italy, Sep. 2004, pp. 347–350.

[8] M. Grangetto, E. Magli, and G. Olmo, “Multimedia selective encryption by means of randomized arithmetic coding,” IEEE Trans. Multimedia, vol. 8, no. 5, pp. 905–917, Oct. 2006.

[9] J. Wen, H. Kim, and J. D. Villasenor, “Binary arithmetic coding with key-based interval splitting,” IEEE Signal Process. Lett., vol. 13, no. 2, pp. 69–72, Feb. 2006.

[10] Announcing the ADVANCED ENCRYPTION STANDARD (AES), Fed. Inf. Process. Standards Pub. 197, 26, NIST, Nov. 2001.

[11] T. Cover and J. Thomas, Elements of Information Theory. New York: Wiley, 1991.

[12] N. T. Courtois, “General principles of algebraic attacks and new design criteria for cipher components,” in Proc. AES 2004. Berlin, Germany: Springer-Verlag, 2005, vol. 3373, Lecture Notes in Computer Science, pp. 67–83.

[13] D. Marpe and T. Wiegand, “A highly efficient multiplication-free binary arithmetic coder and its application in video coding,” in Proc. ICIP 2003, Barcelona, Spain, Sep. 2003, pp. II-263–II-266. Commun., vol. 37, no. 2, pp. 93–98, Feb.1989.

[17] A. Moffat, R. M. Neal, and I. H.Witten, “Arithmetic coding revisited,” ACM Trans. Inf. Sys., vol. 16, no. 3, pp. 256–294, Jul. 1998.

[18] A. Hodjat and I. Verbauwhede, “Area-throughput trade-offs for fully pipelined 30 to 70 Gbits/s AES processors,” IEEE Trans. Comput., vol. 55, no. 4, pp. 366–372, Apr. 2006.

[14] M. Dyer, D. Taubman, and S. Nooshabadi, “Improved throughput thmetic coder for JPEG2000,” in Proc. Int. Conf. Image Process., Singapore, Oct. 2004, pp. 2817–2820.

[15] R. R. Osorio and J. D. Bruguera, “A newarchitecture for fast arithmetic coding in H.264 advanced video coder,” in Proc. 8th Euromicro Conf. Digital System Design, Porto, Portugal, Aug. 2005, pp. 298–305.

[16] J. Rissanen and K. M. Mohiuddin, “Amultiplication-free multialphabet arithmetic code,” IEEE Trans.

[19] M. Powell, 2006, The Canterbury Corpus [Online].Available:http://corpus.canterbury.ac.nz/index.html

**Ravinder Tirupati** received B.Tech degree and M.Tech degree from JNTU Hyderabad in the year of 2006 and 2010. Currently working as Asst. Professor

Research interests are Compression, VLSI based and Image Processing.

**CH. Neelima** received B.Tech degree and M.Tech degree from JNTU Hyderabad in the year of 2006 and 2012. Currently working as Asst. Professor

Research interests are Compression, VLSI based and Image Processing.